ForwardCom: An open-standard instruction set for high-performance microprocessors

Agner Fog

August 7, 2021

Contents

Chapter 1

Introduction

ForwardCom stands for Forward Compatible Computer system.

This document describes a new open instruction set architecture designed for optimal performance, flexibility and scalability. The ForwardCom project includes both a new instruction set architecture and the corresponding ecosystem of software standards, application binary interface (ABI), memory management, development tools, library formats, and system functions. This project illustrates the improvements that can be obtained by a complete vertical redesign of hardware and software based on an open, collaborative process.

A short introduction to ForwardCom is provided at http://www.forwardcom.info.

This manual and all associated code is maintained at https://github.com/ForwardCom.

1.1 Highlights

- The ForwardCom instruction set is neither RISC nor CISC, but a new paradigm combining the advantages of both. ForwardCom has few instructions, but many variants of each instruction. A consistent template system with few instruction sizes combines the fast and streamlined decoding and pipeline design of RISC systems with the compactness and more work done per instruction of CISC systems.
- The instruction formats are fully orthogonal. The same instruction can be coded with different operand types, different types of register operands, memory operands, or immediate constant operands. Instructions can be coded in a compact form where the destination register is the same as the first source register, or in a non-destructive form with three or four registers. Immediate constants are compressed, if possible, to save code space.
- The ForwardCom design is scalable to support small embedded systems as well as large supercomputers and vector processors without losing binary compatibility.
- Vector registers of variable length are provided for efficient handling of large data sets.
- Array loops are implemented in a new flexible way that automatically uses the maximum vector length supported by the microprocessor in all but the last iteration of a loop. The last iteration automatically uses a vector length that fits the remaining number of elements. No extra code is needed to deal with remaining data and special cases. There is no need to compile the code separately for different microprocessor versions with different vector lengths.
- No recompilation or update of software is needed when a new microprocessor with a different vector register length becomes available. The software is guaranteed to be forward compatible and take advantage of the longer vectors of new microprocessor models without recompilation.

- Memory management is simpler and more efficient than in traditional systems. Various techniques are used for avoiding memory fragmentation. It is possible to avoid memory paging and use a memory map with a limited number of sections with variable size instead of a translation lookaside buffer (TLB) with a large number of fixed-size pages.
- There are no dynamic link libraries (DLLs) or shared objects. Instead, there is only one type of function libraries that can be used for both static and dynamic linking. Only the part of the library that is actually used is loaded and linked. The library code is kept contiguous with the main program code to improve caching and reduce memory fragmentation. Executable files can be re-linked to replace or update library functions and plug-ins and to support multiple user interface frameworks.
- A mechanism for calculating the required stack size is provided. This can prevent stack overflow in most cases without making the data stack bigger than necessary.
- A mechanism for optimal register allocation across program modules and function libraries is provided. This makes it possible to keep most variables in registers without spilling to memory. Vector registers can be saved in an efficient way that stores only the part of the register that is actually used.
- Strong security features are fundamental parts of the hardware and software design.
- Standards for software tools, ABI, file formats, system libraries, etc. are defined in order to establish compatibility between different programming languages and different platforms. It is possible to code different parts of a program in different programming languages.

The ForwardCom design can be useful for many purposes where performance is important, where large vectors are desired, where security is important, or where the copyright and license restrictions of proprietary microprocessor systems is an obstacle.

The ForwardCom design is also useful as a sandbox for university projects and experiments aiming at improving many different aspects of computer design, as discussed at http://www.forwardcom.info.

1.2 Background

An instruction set architecture is a standardized set of machine instructions that a computer can run. There are many instruction set architectures in use.

Some commonly used instruction sets are poorly designed from the beginning. These systems have been augmented many times with extensions and patches. One of the worst cases is the widely used x86 instruction set and its many extensions. The x86 instruction set is the result of a long history of short-sighted extensions and patches. The result of this development history is a very complicated architecture with thousands of different instruction codes, which is very difficult and costly to decode in a microprocessor. We need to learn from past mistakes in order to make better choices when designing a new instruction set architecture and the software that supports it.

The design should be based on an open process. Krste Asanović and David Patterson (2014) have presented compelling arguments for why an open instruction set should be preferred. Openness can be crucial for the success of a technical design. For example, the original IBM PC in the early 1980's had an advantage over competing computers because the open architecture allowed other hardware and software producers to make compatible equipment. IBM lost their market dominance when they switched to the proprietary Micro Channel Architecture in 1987. The successes of open source software are well known and need no further discussion here. The only thing that is missing for a complete computer ecosystem based on open standards is

an open microprocessor architecture. This will open the market also for smaller microprocessor producers and niche products.

This project is based on discussions in various Internet forums. The specifications are preliminary. The development of a new standard should benefit from a long experimental phase, and it would be unwise to make it a fixed standard at this initial stage.

1.3 Design goals

Previously published open instruction sets are suitable for small, cheap microprocessors for embedded systems, system-on-a-chip designs, FPGA implementations for scientific experiments, etc. The proposed ForwardCom architecture takes the idea further and aims at a design that can outperform common high-end processors.

The ForwardCom instruction set architecture is based on the following priorities:

- The instruction set should have a simple and consistent modular design.
- The instruction set represents a suitable compromise between the RISC principle that enables fast decoding, and the CISC principle that makes it possible to do more work per instruction and to use the code cache more efficiently.
- The design should be extensible so that new instructions and extensions can be added in a consistent and predictable way.
- The design should be scalable so that it is suitable for both small computers with on-chip RAM and large supercomputers with very long vectors.
- The design should be competitive over current commercial designs with a focus on the high-end applications of tomorrow rather than the low-end applications of yesterday.
- Vector support and other features that have proven essential for high performance should be a fundamental part of the design, not a clumsy appendix.
- Security should be a fundamental part of the design, not patches added ad hoc.
- The instruction set should be designed through an open process with the participation of the international hardware and software community, similar to the standardization work in other technical areas.
- The entire vertical design should be non-proprietary and allow anybody to make compatible software, hardware, and equipment for test, debugging and emulation.
- Decisions about instructions and extensions should not be determined by the short term marketing considerations of an oligopolistic microprocessor industry but by the long term needs of the entire hardware and software community.
- The design should allow the construction of forward compatible software that will run optimally without recompilation on future processors with larger vector registers.
- The design should allow application-specific extensions.
- The basic aspects of the entire ecosystem of ABI standard, assembler, compilers, function libraries, system functions, user interface framework, etc. should also be standardized for maximum compatibility.

A new instruction set will not easily get success on a commercial market, even if it is better than legacy systems, because the market prefers backward compatibility with existing software and hardware. It is unlikely that the ForwardCom instruction set will make a successful commercial

product within a short time frame, but the discussion about what an ideal instruction set, microprocessor design, and software ecosystem might look like is always useful. The ForwardCom project has already generated so many important new ideas that it is worth pursuing further, even if we do not know where this process will end. The present work can be useful if the need for introducing a new instruction set architecture should arise for other reasons. It will be particularly useful for large vector processors, for applications where security is important, for real-time operating systems, for FPGA soft cores, as well as for projects where the patent and license restrictions of other architectures would be an obstacle.

The ideas in this document will also be useful as a source of inspiration and for scientific experiments. Many of the ideas are independent of the design details and may be implemented in other systems.

1.4 Problems addressed by ForwardCom

The design of ForwardCom was prompted by many years of frustration with existing systems. The design is trying to address and solve a lot of problems with existing CPU designs as well as the surrounding ecosystems of development tools, ABI standard, and operating systems. This list provides an overview of problems that the ForwardCom design is trying to solve:

- RISC vs. CISC. The consistent template design of ForwardCom instructions aims at obtaining the efficient instruction decoding and smooth pipeline design of RISC systems combined with the more work done per instruction of CISC systems. RISC systems typically have a fixed instruction size of 32 bits that makes it impossible to include larger addresses, constants, and option bits in a single instruction. ForwardCom allows instructions to have a size of one, two, or three 32-bit words. This provides space for larger addresses, constants, and option bits to allow each instruction to contain more information and to have many different variants. Common CISC systems such as x86, on the other hand, have a variable instruction size that is so difficult to decode that decoding has become a serious bottleneck. ForwardCom avoids this problem by indicating the instruction size with just two bits.

- Forward compatibility. Current SIMD designs have been made with little foresight of future extensions with larger vectors. It is impossible in most other systems to save and restore a vector register in a way that can accommodate future extensions to the vector length. This has caused a lot of problems and awkward patches in current systems. Software has to be recompiled for every new extension. The ForwardCom design with variable vector lengths makes the software automatically use the maximum vector length of the CPU it is running on with no need for recompilation.

- Vector loops. Current SIMD designs have a problem with vector loops when the loop count is not certain to be a multiple of the vector length. The new design with variable vector lengths solves this problem in an elegant and very efficient way.

- Position independent code. All code addresses are relative to the instruction pointer. All writeable data are addressed relative to a data pointer. Code and data can be relocated independently of each other.

- Data coherency. The ForwardCom design makes it possible to store constant data in instruction codes instead of constants scattered in static data memory. This reduces cache misses.

- Suitable for out-of-order execution. ForwardCom has no global status flags or status register that would complicate parallel and out-of-order execution. The efficiency of out-of-order

scheduling is also improved by avoiding instructions that modify a partial register and leave the rest of the register unchanged.

- No microcode. Complex instructions in x86 and other architectures use microcode. This makes decoding inefficient. ForwardCom avoids microcode by using only instructions that fit into the pipeline design. A few more complex instructions can be implemented with state machines or application-specific FPGA modules.

- Function libraries. There is only one kind of function library which serves the purposes of both static libraries, dynamic libraries, shared objects, and program plug-in modules. The library code is linked in a way that makes it contiguous with the program code it serves. Executable program files can be relinked to update or replace a linked library. Problems with missing or incompatible library versions are avoided.

- Stack size calculation. Stack overflow can be prevented by calculation of the maximum stack size during the link process if the program has no recursive functions.

- Avoid memory fragmentation. The design of function libraries, stack size calculation, position independent code, and other efforts are able to reduce memory fragmentation to a level where the TLB (translation lookaside buffer) can be replaced by a memory map with a limited number of variable-size memory blocks in most cases.

- Error tracking and exception handling. The design has no traps for numerical exceptions and no status register. Instead, floating point errors are indicated in propagating NAN payloads. Integer overflow can be indicated in propagating extra vector elements if needed. This makes out-of-order parallelism and SIMD parallelism simpler and more efficient.

- Avoid register spilling. Object files contain information about which registers each function is using. This makes it possible to keep most or all variables in registers without ever spilling to memory.

- Function calling convention. Call stack and data stack are separate. The function calling convention is safe and efficient. Function parameters are transferred in registers, not on the stack. Tail calls are always possible.

- User friendly assembly syntax. The ForwardCom assembler gets out of the habitual thinking that assembly syntax must be obscure and complicated. Adding two registers is as simple as `int32 r1 = add(r2, r3)`, or even `int r1 = r2 + r3`. This is easily intelligible to high-level language programmers and leaves no doubt about which operands are source and destination. Branches and loops can be coded with C-style syntax such as `for (int r1 = 0; r1 < r2; r1++) { }`

- Security. A lot of security features are part of the basic design. See page 109

- Free and open. A noncommercial development process and a free license improves the possibilities for synergy between different hardware and software developers and university scientists. Commercial CPU vendors have often produced suboptimal designs due to the priority of short-term marketing goals. This is avoided with an open development process.

1.5 Comparison with other open instruction sets

A few other open instruction sets have been proposed, most notably RISC-V and OpenRISC. Both are pure RISC designs with mostly fixed 32-bit instruction word sizes. These instruction sets are suitable for small systems where the use of silicon space is economized, but they are not designed for high performance superscalar processors and they do not focus on details that

are critical for achieving maximum performance in bigger systems. The ForwardCom system is thought as the next step towards making an open instruction set that is actually more efficient than the best commercial instruction sets today.

A typical RISC design with the instruction size limited to 32 bits leaves only limited space for immediate constants and addresses of memory operands. A medium-size program will need 32-bit relative addresses of static memory operands to avoid overflow during the relocation process in the linker. A 32-bit relative address requires several instructions in the pure RISC designs. For example, to add a memory operand to the value of a register, you typically need five instructions in a RISC design with only 32-bit instruction words: (1) load the lower part of the 32-bit address offset, (2) add the upper part of the 32-bit address offset, (3) add the reference pointer or instruction pointer to this value, (4) read the memory operand from the calculated address, (5) do the desired addition. The ForwardCom design does all this in a single instruction with double word size. The speed advantage is obvious. The address calculation, load, and execution are done at each their stage in the pipeline in order to achieve a smooth throughput of one instruction per clock cycle in each pipeline.

Another important difference is that the previous RISC designs have limited support for vector operations. The ForwardCom design introduces a new system of variable-length vector registers that is more efficient and flexible than the best current commercial designs. Efficient vector operations are essential for obtaining maximum performance, and this has been an important priority in the design of the ForwardCom architecture proposed here.

1.6 References and links

- Krste Asanović and David Patterson: “The Case for Open Instruction Sets. Open ISA Would Enable Free Competition in Processor Design”. Microprocessor Report, August 18, 2014. www.linleygroup.com/newsletters/newsletter_detail.php?num=5210
- RISC-V: The Free and Open RISC Instruction Set Architecture riscv.org
- OpenRISC: openrisc.io
- Open Cores: opencores.org
- Agner Fog: Proposal for an ideal extensible instruction set, 2015. A blog discussion thread that initiated the ForwardCom project. www.agner.org/optimize/blog/read.php?i=421
- Agner Fog: Stop the instruction set war, 2009. Blog post about the problems with the x86 instruction set. www.agner.org/optimize/blog/read.php?i=25
- Darek Mihocka: Standard Need To Be Forward Looking, 2007. Blog post criticizing the x86 instruction set standard. www.emulators.com/docs/nx02_standards.htm. See also the following pages.
- Agner Fog: Floating point exception tracking and NAN propagation, 2020. www.agner.org/optimize/nan_propagation.pdf

Chapter 2

Basic architecture

This chapter gives an overview of the most important features of the ForwardCom instruction set architecture. Details are given in the subsequent chapters.

2.1 A fully orthogonal instruction set

The ForwardCom instruction set is fully orthogonal in all respects. Where other instruction sets have a large number of different instructions for different register types, operand types, operand sizes, addressing modes, etc., ForwardCom has fewer instructions, but many variants of each instruction. This modular design makes the hardware implementation much simpler. The same instruction can use integer operands of all sizes and floating point operands of all precisions. It can use register operands, memory operands or immediate operands. It can use many different addressing modes. Instructions can be coded in short forms with two operands where the same register is used for destination and source operand, or longer forms with three operands. It can work with scalars or vectors of any size. It can have predication or masks for conditional execution, and it can have optional flag inputs for determining rounding mode, exception control and other details, where appropriate. Data constants of all types can be included in the instructions and compressed in various ways to reduce the instruction size.

Rationale

The orthogonality is implemented by a standardized modular design that makes the hardware implementation simpler. It also makes compilation simpler and more flexible and makes it easier for the compiler to convert linear code to vector code.

The support for immediate constants of all types is an improvement over current systems. Most current systems store floating point constants in a data segment and access them through a 32-bit address in the instruction code. This is a waste of data cache space and causes many cache misses because the data are scattered around in different sections. Replacing a 32-bit address with a 32-bit immediate constant makes the code more efficient without increasing the code size. Extensions to allow 64-bit immediate constants are possible at the cost of having instructions with triple size.

2.2 Instruction size

The ForwardCom instruction set uses a 32-bit word size for code. An instruction can consist of one, two, or three 32-bit words. It is possible to add future extensions with instruction sizes of four or more words, but there is currently no need for this.

Rationale

A CISC architecture with many different instruction sizes is inefficient in superscalar processors where we want to execute several instructions per clock cycle. The decoding front end is often a bottleneck, especially in the x86 architecture. The decoder has to determine the length of the first instruction before it knows where the next instruction begins. The “instruction length decoding” is a fundamentally serial process which makes it difficult to decode multiple instructions per clock cycle. Modern x86 microprocessors have an extra “micro-operations cache” after the decoder in order to circumvent this bottleneck.

Here, it is desired to have as few different instruction sizes as possible and to make it easy to determine the length of each instruction. We want a small instruction size for the most common simple instructions, but we also need a larger instruction size in order to accommodate things like a larger register set, instructions with multiple operands, vector operations with advanced features, 32-bit address offsets, and large immediate constants. This proposal is a compromise between code compactness, easy decoding, and space for advanced features. The instruction size is indicated by only two bits. A decoder can find the instruction boundaries in n words by means of a simple Boolean function of 2n inputs.

2.3 Register set

There are 32 general purpose registers (r0–r31) of 64 bits each, and 32 vector registers (v0–v31) of variable length. The maximum vector length is different for different hardware implementations. The general purpose registers can be used for integers of up to 64 bits as well as for pointers. The vector registers can be used for scalars and vectors of integers and floating point numbers.

The following special registers are defined and visible at the application program level. All have 64 bits:

- Instruction pointer (IP)
- Data section pointer (DATAP)
- Thread data pointer (THREADP)
- Stack pointer (SP)
- Numeric control register (NUMCONTR)

The stack pointer is identical to r31. The other special registers cannot be accessed as ordinary registers.

There is no dedicated flags register. Registers r0–r6 and v0–v6 can be used for masks, predicates and floating point option flags to control attributes such as rounding mode and exception control.

The unused part of a register is always set to zero. This means that integer operations with an operand size smaller than 64 bits and vector operations with a vector length smaller than the maximum will always set the unused bits of the destination register to zero.

Rationale

The number of registers is a compromise between code density and flexibility. The cost of spilling registers to memory is usually important only in the critical innermost loop, which is unlikely to need more than 32 registers.

We can avoid false dependencies on the previous value of a register by setting all unused register bits to zero rather than leaving them unchanged. The hardware can save power by disabling the unused parts of execution units and data buses.

A dedicated flags or status register is unfeasible for vector processing, parallel processing, out-of-order processing, and instruction scheduling.

The reason for handling floating point scalars in the vector registers rather than in separate registers is to make it easy for a compiler to convert scalar code including function calls to vector code. Floating point code often contains calls to mathematical library functions. A library function with variable-length vectors as input and output can be used for both scalars and vectors, and the compiler can easily vectorize code that contains such library function calls.

2.4 Vector support

A vector register can contain signed or unsigned integers of 8, 16, 32, 64, and optionally 128 bits, or floating point numbers of single and double precision. There is limited support for floating point numbers in half precision and optional support for quadruple precision. All elements of a vector must have the same type. The elements of a vector are processed in parallel. For example, a vector addition will produce the sum of two vectors in a single operation.

The vector registers have variable length. Each vector register has extra bits for storing the length of the vector. The maximum vector length depends on the hardware. For example, if the hardware supports a maximum vector length of 64 bytes and a particular application needs only 16 bytes, then the vector length is set to 16.

Some instructions need to specify the length of a vector explicitly, for example when reading a vector from memory. These instructions use a general purpose register for specifying the vector length. The length is usually indicated as the number of bytes, not the number of vector elements.

The maximum length supported by the processor must be a power of 2. The actual length specified does not need to be a power of 2. If the specified length is longer than the maximum length, then the maximum length is used.

The contents of a vector register can arbitrarily be interpreted as any of the types and element sizes supported. For example, the hardware does not prevent the application of integer instructions on a vector that contains floating point data. It is the responsibility of the programmer that the code makes sense.

2.5 Vector loops

A special addressing mode is provided to make vector loops more compact and efficient. It uses a pointer P to the end of an array, and a negative index J, and calculates the address of a memory operand as P-J, where P and J are general purpose registers. This makes it possible to make a loop through an array as illustrated by the following pseudocode:

```
P = address of array
J = size of array (in bytes)
L = maximum vector length (depends on processor)
X = a vector register
P += J; // point to end of array
while (J > 0) {
    X = whatever_operation(X, [P-J], vector_length = J)
    J -= L;
}
```

This loop works in the following way: P points to the end of the array. J is the remaining number of array bytes; counting down until the loop is finished. The loop reads one vector at a time from the array at the address (P-J). J is larger than the maximum vector length L in all but the last iteration of the loop. This makes the processor use the maximum vector length. If the array size is not divisible by the maximum vector length then the last iteration of the loop will use a smaller vector length that fits the remaining number of elements. Obviously, the loop can contain any number of vector read, vector write, and vector arithmetic instructions, using the same principle.

This loop will work on different processors with different maximum vector lengths *without knowing the maximum vector length at compile time*. Thus, the same piece of software will work on different microprocessors with different vector lengths without the need to compile separately for each microprocessor.

A further advantage is that no extra code is needed after the loop to handle remaining elements in the case that the array size is not divisible by the vector length. The loop overhead can be reduced to a single instruction (sub_maxlen/jump_pos) which subtracts L from J and jumps back if the result is positive.

Rationale

Most current systems have fixed vector lengths. If different processors have different vector lengths then you have to compile the code separately for each vector length. Every time a new processor with longer vectors comes on the market, you have to compile a new version of the code for the new vector length, using newly defined extensions to the instruction set. It usually takes several years for the new software to be developed and to penetrate the mainstream market. It is so costly for software producers to develop, test, and maintain different versions of their code for each vector length that this is rarely done.

A further problem with current systems is that it is impossible to save a vector register in a way that is guaranteed to be compatible with future processors with longer vectors. This is no problem with the ForwardCom design because the vector length is stored in the vector register itself. Instructions are provided for saving and restoring vectors of variable length and for storing only the part of a vector register that is actually used.

The ForwardCom design makes it possible to take advantage of a new processor with longer vector registers immediately without recompiling the code. The loop method described above makes this easy and very efficient. You do not need different versions of the code for different processors.

It is possible to obtain the same effect without the special negative addressing mode by inverting the sign of J and allowing a negative value in the register that specifies the vector length while using the absolute value for the actual vector length. This solution is less elegant and more confusing, but it may possibly be included in other instruction sets by allowing negative values when specifying a vector length.

Loop unrolling is generally not necessary. The loop overhead is already reduced to a single instruction and a superscalar processor will execute multiple iterations in parallel if dependency chains are not too long. Loop unrolling with multiple accumulators may be useful for hiding a loop-carried dependency. In this case, you will either insert a loop control instruction after each section in the unrolled code or calculate the loop iteration count before the loop.

The ForwardCom design has no practical limit to the vector length that a microprocessor can support. A large microprocessor with very long vectors can be useful for calculations with a high amount of data parallelism. Other solutions to obtain high performance on parallel data processing have been discussed, such as rolling register stacks and software pipelining, but it was concluded that long vectors is the method that can be implemented most efficiently in the microprocessor as well as in the compiler.

2.6 Maximum vector length

The maximum length of vector registers will be different for different processors. The maximum length must be a power of 2. It can be as large as desired and should be at least 16 bytes. Each instruction can use a smaller length, which does not need to be a power of 2.

The maximum length may be different for different element sizes. For example, the maximum length for 32-bit integers can be 32 bytes to contain eight integers, while the maximum length for 8-bit integers could be 16 bytes to contain 16 smaller numbers. However, the maximum length must be the same for different types with the same element size. For example, the maximum length for double precision floating point numbers must be the same as for 64-bit integers because loops are likely to contain both types when integer vectors are used as masks for floating point vectors. The maximum length for a 32-bit element size cannot be less than for any other element size or operand type. This rule guarantees that it is possible to save a complete vector using a 32-bit operand type.

The maximum vector length should generally be the same for all instructions for the same data type, but there may be exceptions for instructions that are particularly expensive to implement.

It is possible for an application program or the operating system to reduce the maximum vector length. This can be useful if a smaller vector length is more appropriate for a particular purpose.

It is also possible to increase the apparent maximum vector length for purposes of emulation. Virtual vector registers that are bigger than what the hardware supports may be emulated through traps (synchronous interrupts) in order to verify the functionality of a program on processors with a longer maximum vector length than is currently available.

When an instruction specifies a longer vector than the maximum, then the maximum length is used (unless the emulation of larger vectors is activated). This is necessary for the efficient implementation of vector loops as described above on page 13. If the specified vector length is zero or negative then the result will be a vector of zero length.

2.7 Instruction masks

Most instructions can have a mask register which can be used for conditional execution and for specifying various options. Instructions with general purpose registers use one of the registers r0–r6 as a mask register or predicate. Bit 0 of the mask register indicates whether the operation is executed or not.

The instruction will produce the normal result when bit 0 of the mask is one, and a fallback value when this bit is zero. The fallback value can be the value of the first source operand, a separate register, or zero.

This mechanism can be vectorized. Instructions with vector registers use one of the vector registers v0–v6 as mask register. The calculation of each vector element is conditional on the corresponding element in the mask register.

An arithmetic operation with a mask of zero can never generate an error condition. A memory read or write with an illegal address and a mask of zero may or may not generate an error trap.

Additional bits in the mask register are used for various options, overriding the values in the numeric control register. See page 28 for details.

2.8 Addressing modes

All memory addressing is relative to a base pointer. Memory operands are addressed in this general form:

```
Address = Base pointer + Index * Scale + Offset
```

Where Base pointer is a 64-bit base pointer, Index is a 64-bit index register, Scale is a scale factor, and Offset is a constant. A base pointer is always present; the other elements are optional.

The base pointer can be a general purpose register or it can be the instruction pointer (IP), data section pointer (DATAP), thread data pointer (THREADP), or stack pointer (SP).

The index register can be one of the registers r0–r30. A value of 31 in the index register field means no index register.

A limit can be applied to the index register in the form of an integer constant. A trap is generated if the index register is bigger than the limit in an unsigned comparison.

The scale factor is equal to the operand size (in bytes) for scalar operands and broadcasts. The scale factor is 1 for vector operands. A special addressing mode with Scale = -1 is also available, as explained on page 13.

The offset is a sign-extended integer of 8, 16, or 32 bits. 8-bit offsets are multiplied by the operand size. Offsets of 16 and 32 bits have no multiplier.

Memory operands in vector instructions can load a vector of a specified length, a scalar, or a broadcast scalar. The length of the loaded or broadcast vector is specified by a general purpose register. The specified length is the number of bytes. The number of vector elements is the number of bytes divided by the operand size. Register r31, which is the stack pointer, cannot be used for specifying vector length. Instead, a value of 31 in the length register field will give a scalar.

Jumps and calls specify a target address relative to the instruction pointer. The relative address is specified with a signed offset of 8, 16, 24, or 32 bits, multiplied by the code word size which is 4. This will cover an address range of $\pm$ 8 gigabytes with the 32-bit offset.

Rationale

A 64-bit flat address space is used. Relative addressing is used in order to avoid 64-bit addresses in the instruction code. In the rare case that a 64-bit absolute address is needed, it must be loaded into a register which is then used as a pointer.

Addressing with an index scaled by the operand size is useful for arrays. A limit can be applied to the index so that array bounds can be checked without any extra instructions.

Addressing with a negative index is useful for the efficient implementation of vector loops described on page 13.

The addressing modes specified here will cover all common applications, including arrays, vectors, structures, classes, and stack frames.

Support for addressing modes with both base pointer, index, and direct offset is optional because this requires two adders in the address-calculation stage in the pipeline which might limit the maximum clock frequency.

Chapter 3

Instruction formats

3.1 Formats and templates

All instructions use one of the general format templates shown below (the most significant bits are shown to the left). The basic layout of the 32-bit code word is shown in template A. Template B, C and D are derived from template A by replacing 8, 16, or 24 bits, respectively, with immediate data. Double-size and triple-size instructions can be constructed by adding one or two 32-bit words to one of these templates. For example, template A with an extra 32-bit word containing data is called A2. Template E2 is an extension to template A where the second code word contains an extra register field, extra opcode bits, mode bits, option bits, and data.

Bits	2	3	6	5	1	2	5	3	5
Field	IL	Mode	OP1	RD	M	OT	RS	Mask	RT
Template A. Has three operand registers and a mask register.									

Bits	2	3	6	5	1	2	5	8
Field	IL	Mode	OP1	RD	M	OT	RS	IM1
Template B. Has two operand registers and an 8-bit immediate constant.								

Bits	2	3	6	5	8	8
Field	IL	Mode	OP1	RD	IM2	IM1
Template C. Has one operand register two 8-bit immediate constants.						

Bits	2	3	3	24
Field	IL	Mode	OP1	IM2
Template D. Has no register and a 24-bit immediate constant.				

Bits	2	3	6	5	1	2	5	3	5
Field	IL	Mode	OP1	RD	M	OT	RS	Mask	RT
Field	IM2								
Template A2. 2 words. As A, with an extra 32-bit immediate constant.									

Bits	2	3	6	5	1	2	5	8
Field	IL	Mode	OP1	RD	M	OT	RS	IM1
Field	IM2							
Template B2. As B, with an extra 32-bit immediate constant.								

Bits	2	3	6	5	8	8
Field	IL	Mode	OP1	RD	IM2	IM1
Field	IM3					
Template C2. As C, with an extra 32-bit immediate constant.						

Bits	2	3	6	5	1	2	5	3	5
Field	IL	Mode	OP1	RD	M	OT	RS	Mask	RT
Bits	3	5	2	6	16				
Field	Mode2	RU	OP2	IM3	IM2				
Template E2. Has 4 register operands, mask, a 16-bit immediate constant, and extra bits for mode, opcode, and options.									

Bits	2	3	6	5	1	2	5	3	5
Field	IL	Mode	OP1	RD	M	OT	RS	Mask	RT
Field	IM2								
Field	IM3								
Template A3. 3 words. As A, with two extra 32-bit immediate constants.									

Bits	2	3	6	5	1	2	5	8
Field	IL	Mode	OP1	RD	M	OT	RS	IM1
Field	IM2							
Field	IM3							
Template B3. As B, with two extra 32-bit immediate constants.								

Bits	2	3	6	5	1	2	5	3	5
Field	IL	Mode	OP1	RD	M	OT	RS	Mask	RT
Bits	3	5	2	6	16				
Field	Mode2	RU	OP2	IM3	IM2				
Field	IM4								
Template E3. As E2, with an extra 32-bit immediate constant.									

The meaning of each field is described in the following table.

Table 3.1: Fields in instruction templates

Field name	Meaning	Values
IL	Instruction length	0 or 1: 1 word = 32 bits 2: 2 words = 64 bits 3: 3 words (possibly more in future extensions if mode > 3)
Mode	Format	Determines the format template and the use of each field. Extended with the M bit when needed. See details below.
Mode2	Format	Extension to Mode.
OT	Operand type and size (OS)	0: 8 bit integer, OS = 1 byte 1: 16 bit integer, OS = 2 bytes 2: 32 bit integer, OS = 4 bytes 3: 64 bit integer, OS = 8 bytes 4: 128 bit integer, OS = 16 bytes (optional) 5: single precision float, OS = 4 bytes 6: double precision float, OS = 8 bytes 7: quadruple precision float, OS = 16 bytes (optional) The OT field is extended with the M bit when needed.
M	Operand type or mode	Extends the mode field when bit 1 and bit 2 of Mode are both zero (general purpose registers). Extends the OT field otherwise (vector registers).
OP1	Opcode	Decides the operation, for example add or move.
OP2	Opcode	Opcode extension for single-format instructions. May also be used as an extension to IM3.
RD	Destination register	r0 – r31 or v0 – v31. Also used for first source operand and fallback if the instruction format does not specify enough operands.
RS	Source register	r0 – r31 or v0 – v31. Source register, pointer, or fallback.
RT	Source register	r0 – r31 or v0 – v31. Source register, index, or vector length.
RU	Source register	r0 – r31 or v0 – v31. Source register or fallback.
Mask	mask register	0-6 means that a general purpose register or vector register is used for mask and option bits. 7 means no mask.
IM1 IM2 IM3 IM4	Immediate data	8, 16, 24, or 32 bits immediate operand or address offset or option bits. Adjacent IM fields can be merged to make a larger constant.

Instructions have several different formats, defined by the IL and mode bits, according to table 3.2 below. The different formats specify different sizes of immediate data or memory operands with different addressing modes.

Instructions can have up to three source operands (input), one destination operand (output), and a mask. The destination operand always uses the RD field, except where the destination is a memory operand. The source operands are using the available operand fields according to the following algorithm: The required source operands are assigned to the available operand fields defined by table 3.2 in the following order of priority: immediate data field, memory operand, RT,

RS, RU, RD. The operands are assigned in reverse order so that the last operand gets the field that comes first in this order of priority. For example, the instruction r1 = r2 - r3 using template A will be RD = RS - RT. RD is used for both destination and the first source operand only if there are no other vacant register fields.

The coding of instructions with two or three source operands is indicated in the table in the following way:
RD = f2(RS,RT) means that instructions with two input operands (f2) use the register specified in RD as destination operand and RS and RT as source operands.
RD=f3(RD, RU, [RS+RT*OS+IM2]) means that instructions with three input operands (f3) use the register specified in RD as both destination and the first source operand. The second source operand is RU. The third source operand is a memory operand with RS as base pointer, RT as index scaled by the operand size, and the constant IM2 as offset.
Instructions with only one input operand are coded as f2 with the first source operand omitted.

Table 3.2: List of instruction formats

Format name	IL	Mode. Mode2	Template	Use
0.0	0	0	A	Three general purpose register operands. RD = f2(RS, RT). RD = f3(RD, RS, RT).
0.1	0	1	B	Two general purpose registers and 8-bit immediate operand. RD = f2(RS, IM1). RD = f3(RD, RS, IM1).
0.2	0	2	A	Three vector register operands. RD = f2(RS, RT). RD = f3(RD, RS, RT).
0.3	0	3	B	Two vector registers and a broadcast 8-bit immediate operand. RD = f2(RS, IM1). RD = f3(RD, RS, IM1).
0.4	0	4	A	One vector register and memory operand. Vector length specified by general purpose register. RD = f2(RD, [RS]). length=RT.
0.5	0	5	A	One vector register and a memory operand with base pointer and negative index. This is used for vector loops as explained on page 13. RD = f2(RD, [RS-RT]). length=RT.
0.6	0	6	A	One vector register and a scalar memory operand with base pointer and scaled index. RD = f2(RD, [RS+RT*OS]).
0.7	0	7	B	One vector register and a scalar memory operand with base pointer and 8-bit offset. RD = f2(RD, [RS+IM1*OS]).
0.8	0	0 M=1	A	One general purpose register and a memory operand with base pointer and scaled index. RD = f2(RD, [RS+RT*OS]).
0.9	0	1 M=1	B	One general purpose register and a memory operand with base pointer and 8-bit offset. RD = f2(RD, [RS+IM1*OS]).
1.0	1	0	A	Single-format instructions. Three general purpose register operands. RD = f2(RS, RT). RD = f3(RD, RS, RT).

1.1	1	1	C	Single-format instructions. One general purpose register and a 16-bit immediate operand. RD = f2(RD, IM1-2).
1.2	1	2	A	Single-format instructions. Three vector register operands. RD = f2(RS, RT). RD = f3(RD, RS, RT).
1.3	1	3	B	Single-format instructions. Two vector registers and a broadcast 8-bit immediate operand. RD = f2(RS, IM1). RD = f3(RD, RS, IM1).
1.4	1	4	C	Single-format instructions. One vector register and a broadcast 16-bit immediate operand. RD = f2(RD, IM1-2).
1.5	1	5		Vacant. May be used for application-specific vector instructions.
1.6 A	1	6	A	Multiway jump instructions and system calls with three register operands.
1.6 B	1	6	B	Jump instructions with two register operands and 8 bit offset.
1.7C	1	7	C	Jump instructions with one register operand, 8 bit constant (IM2) and 8 bit offset (IM1).
1.7D	1	7	D	Jump instructions with no register and 24 bit offset.
1.8	1	0 M=1	B	Single-format instructions. Two general purpose registers and an 8-bit immediate operand. RD = f2(RS, IM1). RD = f3(RD, RS, IM1).
1.9				There is no format 1.9 because 1.1 has no M bit.
2.0.0	2	0.0	E2	Three general purpose registers and a memory operand with base and 16 bit offset. RD = f2(RT, [RS+IM2]). RD = f3(RU, RT, [RS+IM2]).
2.0.1	2	0.1	E2	Two general purpose registers and a memory operand with base, index and optional 16 bit offset, no scale. RD = f2(RU, [RS+RT+IM2]). RD = f3(RD, RU, [RS+RT+IM2]).
2.0.2	2	0.2	E2	Two general purpose registers and a memory operand with base, scaled index, and optional 16 bit offset. RD = f2(RU, [RS+RT*OS+IM2]). RD = f3(RD, RU, [RS+RT*OS+IM2]).
2.0.3	2	0.3	E2	Two general purpose registers and a memory operand with base, scaled index, and 16-bit limit. Optional. RD = f2(RU, [RS+RT*OS]). RD = f3(RD, RU, [RS+RT*OS]). Limit RT $\leq$ IM2 (unsigned). Support for this format is optional.

2.0.5	2	0.5	E2	One general purpose register and a memory operand with base, scaled index, 16-bit offset, and an 8-bit immediate operand using IM3 extended with OP2. Optional. RD = f2([RS+RT*OS+IM2], IM3). RD = f3(RU, [RS+RT*OS+IM2], IM3).
2.0.6	2	0.6	E2	Four general purpose registers. RD = f2(RS, RT). RD = f3(RU, RS, RT).
2.0.7	2	0.7	E2	Three general purpose registers and a 16-bit integer with left shift. RD = f2(RT, IM2). RD = f3(RS, RT, IM2). IM2 (signed) is shifted left by the 6-bit unsigned value of IM3, or whithout shift if IM3 is used for other purposes.
2.1	2	1	A2	Two general purpose registers and a memory operand with base and 32 bit offset (IM2). RD = f2(RT, [RS+IM2]). RD = f3(RD, RT, [RS+IM2]).
2.2.0	2	2.0	E2	Two vector registers and a broadcast scalar memory operand with base and 16 bit offset. RD = f2(RU, [RS+IM2]). RD = f3(RD, RU, [RS+IM2]). Broadcast to length RT.
2.2.1	2	2.1	E2	Two vector registers and a memory operand with base and 16 bit offset. RD = f2(RU, [RS+IM2]). RD = f3(RD, RU, [RS+IM2]). Length=RT.
2.2.2	2	2.2	E2	Two vector registers and a scalar memory operand with base and scaled index. RD = f2(RU, [RS+RT*OS+IM2]). RD = f3(RD, RU, [RS+RT*OS+IM2]).
2.2.3	2	2.3	E2	Two vector registers and a scalar memory operand with base, scaled index, and 16-bit limit. Optional. RD = f2(RU, [RS+RT*OS]). RD = f3(RD, RU, [RS+RT*OS]). Limit RT $\leq$ IM2 (unsigned).
2.2.4	2	2.4	E2	Two vector registers and a memory operand with base and negative index. RD = f2(RU, [RS-RT+IM2]). RD = f3(RD, RU, [RS-RT+IM2]). Length=RT.
2.2.5	2	2.5	E2	One vector register and a memory operand with base, 16-bit offset, and an 8-bit immediate operand using IM3 extended with OP2. Optional. RD = f2([RS+IM2], IM3). RD = f3(RU, [RS+IM2], IM3). Length=RT.

2.2.6	2	2.6	E2	Four vector registers. RD = f2(RS, RT). RD = f3(RU, RS, RT).
2.2.7	2	2.7	E2	Three vector registers and a broadcast immediate half-precision float or 16-bit integer with left shift. RD = f2(RT, IM2). RD = f3(RS, RT, IM2). Floating point operands: IM2 is half precision. Integer operands: IM2 (signed) is shifted left by the 6-bit unsigned value of IM3, or whithout shift if IM3 is used for other purposes.
2.3	2	3	A2	Three vector registers and a broadcast 32-bit immediate operand. RD = f2(RT, IM2). RD = f3(RS, RT, IM2).
2.4	2	4	A2	One vector register and a memory operand with base and 32 bit offset. RD = f2(RD, [RS+IM2]). length=RT.
2.5	2	5	A2, B2, C2	Jump instructions for OP1 $<$ 8. Single format instructions with memory operands or mixed register types for OP1 $\geq$ 8.
2.6	2	6	A2	Single-format instructions. Three vector registers and a 32-bit immediate operand. RD = f2(RT, IM2). RD = f3(RS, RT, IM2).
2.7	2	7		Currently unused.
2.8	2	0 M=1	A2	Three general purpose registers and a 32-bit immediate operand. RD = f2(RT, IM2). RD = f3(RS, RT, IM2).
2.9	2	1 M=1	A2	Single-format instructions. Three general purpose registers and a 32-bit immediate operand. RD = f2(RT, IM2). RD = f3(RS, RT, IM2).
3.0.0	3	0.0	E3	Three general purpose registers and a memory operand with base and 32 bit offset. RD = f2(RT, [RS+IM4]). RD = f3(RU, RT, [RS+IM4]).
3.0.2	3	0.2	E3	Two general purpose registers and a memory operand w. base, scaled index, and 32 bit offset. RD = f2(RU, [RS+RT*OS+IM4]). RD = f3(RD, RU, [RS+RT*OS+IM4]).
3.0.3	3	0.3	E3	Two general purpose registers and a memory operand with base, scaled index, and 32-bit limit. Optional. RD = f2(RU, [RS+RT*OS]). RD = f3(RD, RU, [RS+RT*OS]). Limit RT $\leq$ IM4 (unsigned).

3.0.5	3	0.5	E3	One general purpose register and a memory operand with base, scaled index, 16-bit offset, and a 32-bit immediate operand. Optional. RD = f2([RS+RT*OS+IM2], IM4). RD = f3(RU, [RS+RT*OS+IM2], IM4).
3.0.7	3	0.7	E3	Three general purpose registers and a 32-bit integer with left shift. RD = f2(RS, IM4 $<<$ IM2). RD = f3(RS, RT, IM4 $<<$ IM2). IM4 (signed) is shifted left by the unsigned value of IM2.
3.1	3	1	A3, B3	Jump instructions for OP1 $<$ 8. Single format instructions with memory operands or mixed register types for OP1 $\geq$ 8.
3.2.0	3	2.0	E3	Two vector registers and a broadcast scalar memory operand with base and 32 bit offset. RD = f2(RU, [RS+IM4]). RD = f3(RD, RU, [RS+IM4]). Broadcast to length RT.
3.2.1	3	2.1	E3	Two vector registers and a memory operand with base and 32 bit offset. RD = f2(RU, [RS+IM4]). RD = f3(RD, RU, [RS+IM4]). Length=RT.
3.2.2	3	2.2	E3	Two vector registers and a scalar memory operand w. base, scaled index, and 32-bit offset. Optional. RD = f2(RU, [RS+RT*OS+IM4]). RD = f3(RD, RU, [RS+RT*OS+IM4]).
3.2.3	3	2.3	E3	Two vector registers and a scalar memory operand with base, scaled index, and 32-bit limit. Optional. RD = f2(RU, [RS+RT*OS]). RD = f3(RD, RU, [RS+RT*OS]). Limit RT $\leq$ IM4 (unsigned).
3.2.5	3	2.5	E3	One vector register and a memory operand with base, 16-bit offset, and a 32-bit immediate operand. Optional. RD = f2([RS+IM2], IM4). RD = f3(RU, [RS+IM2], IM4). Length=RT.
3.2.7	3	2.7	E3	Three vector registers and a broadcast single precision float or 32-bit integer with left shift. RD = f2(RT, IM4). RD = f3(RS, RT, IM4). Floating point operands: IM4 is single precision. Integer operands: IM4 (signed) is shifted left by the unsigned value of IM2.
3.3	3	3	A3	Three vector registers and a broadcast 64-bit immediate operand. RD = f2(RT, IM3:IM2). RD = f3(RS, RT, IM3:IM2).

3.8	3	0 M=1	A3	Three general purpose registers and a 64-bit immediate operand. RD = f2(RT, IM3:IM2). RD = f3(RS, RT, IM3:IM2).
3.9				There is no format 3.9 because 3.1 uses the M bit.
4.x	3	4-7		Reserved for future 4-word instructions and longer.

3.2 Coding of operands

Operand type

The type and size of operands is determined by the OT field as indicated above. The operand type is 32 bit integer if there is no OT field unless otherwise specified. The operand size (OS) is the size in bytes of a scalar operand or a vector element. This is equal to the number of bits divided by 8.

Register type

The instructions can use either general purpose registers or vector registers. General purpose registers are used for source and destination operands and for masks if the Mode field is 0 or 1 (with M = 0 or 1). Vector registers are used for source and destination operands and for masks if Mode is 2-7. Jump instructions use vector registers if M = 1. A few single-format instructions deviate from this rule and use mixed register types.

Pointer register

Instructions with a memory operand always use an address relative to a base pointer. The base pointer can be a general purpose register, the data section pointer, the thread data pointer, the instruction pointer, or the stack pointer. The pointer is determined by the RS field. This field is interpreted as follows.

Single-size instructions with a memory operand (formats 0.4 - 0.9) can use any of the registers r0-r31 as base pointer. r31 is the stack pointer.

Larger instructions with a memory operand and an offset field of at least 16 bits (formats 2.0.x, 2.1, 2.2.x, 2.4, 2.9, 3.0.x, 3.2.x) can use the same registers, except r28 - r30, which are replaced by the thread pointer (THREADP), data section pointer (DATAP), and instruction pointer (IP), respectively.

The instruction pointer may be used for addressing data in a read-only data section. This works in the following way. The address of the end of the current instruction is used as a reference point. This is the same as the address of the next instruction. The reason for using the end of the instruction as reference point is that it makes relocation in the linker independent of the instruction length in most cases. This address is multiplied by 4 when used as a data address because the instruction pointer is addressing 32 bit word units while data pointers are addressing byte units.

Index register

Instruction formats with an index can use r0 - r30 as index in the RT field. A value of 31 in the index field means no index. The signed index is multiplied by the operand size (OS) for formats

0.6, 0.8, 2.0.2, 2.0.3, 2.0.5, 2.2.3, 3.0.3, 3.2.3; by 1 for format 2.0.1; or by -1 for format 0.5 and 2.2.2. The result is added to the address given by the base pointer.

Offsets

Offsets can be 8, 16, or 32 bits. The value is sign-extended to 64 bits. An 8-bit offset is multiplied by the operand size OS, as given by the OT field. An offset of 16 or 32 bits is not scaled. The result is added to the address given by the base pointer and the index.

Support for addressing modes with both index and offset is optional (format 2.0.1, 2.0.2, 2.0.5, 2.2.2, 2.2.4, 3.0.2, 3.0.5, 3.2.2). Hardware implementations where the use of two additions in the address calculation would cause timing problems may allow having an index with a offset of zero or an offset with no index (RT = 31).

Limit on index

Formats 2.0.3, 2.2.3, 3.0.3, and 3.2.3 have a 16-bit or 32-bit limit on the index register. This is useful for checking array limits. A trap is generated if the value of the index register, interpreted as unsigned, is bigger than the unsigned limit. This feature is optional.

Vector length

The vector length of memory operands is specified by r0-r30 in the RT field for formats with a vector memory operand. A value of 31 in the RT field indicates a scalar with the same length as the operand size (OS).

The value of the vector length register indicates the vector length in bytes (not the number of elements). If the value is bigger than the maximum vector length then the maximum vector length is used. If the indicated vector length is zero or negative then the resulting vector will be empty and nothing will be read or written.

The vector length must be a multiple of the operand size OS, as indicated by the OT field. If the vector length is not a multiple of the operand size then the partial vector element will be zero.

The vector length for source operands in vector registers is stored in the register itself.

Combining vectors with different lengths

The length of the destination register of a vector instruction will be the same as the vector length of the first source operand.

A consequence of this is that the length of the result is determined by the order of the operands when vectors of different lengths are combined.

If the source operands have different lengths then the lengths will be adjusted as follows. If a vector source operand is too long then the extra elements will be ignored. If a vector source operand is too short then the missing elements will be zero.

A scalar memory operand is not broadcast but treated as a short vector. It is padded with zeroes to the vector length of the destination.

A broadcast memory operand will use the vector length given by the vector length register. If this is less than the length of the destination then it is padded with zeroes.

An immediate operand will be broadcast to the vector length of the destination.

Immediate constants

Immediate constants can be 8, 16, 32, and 64 bits. Immediate fields are aligned to natural addresses. They are interpreted as follows.

If OT specifies an integer type then the field is interpreted as an integer. If the field is smaller than the operand size then it is sign-extended to the appropriate size. If the field is larger than the operand size then the superfluous upper bits are ignored. The truncation of a too large immediate operand will not trigger any overflow condition.

If OT specifies a floating point type then the field is interpreted as follows. Immediate fields of 8 bits are interpreted as signed integers and converted to floating point numbers of the desired precision. A 16-bit field is interpreted as a half precision floating point number (subnormal numbers are supported for float16). A 32-bit field is interpreted as a single precision floating point number. It is converted to the desired precision if necessary. A 64-bit field is interpreted as a double precision floating point number. A 64-bit field is not allowed with a single precision operand type.

Some instruction formats allow immediate integer constants with a left shift. Large integer constants with a limited number of significant bits can be represented with fewer bits in this way. Format 2.0.7 and 2.2.7 allow a 16-bit immediate constant in IM2 to be shifted left by the unsigned value of IM3 to give a 64-bit signed value, except for instructions that use IM3 for other purposes. Format 3.0.7 and 3.2.7 allow a 32-bit immediate constant in IM4 to be shifted left by the unsigned value of IM2. Any overflow beyond 64 bits is ignored.
Some single-format instructions also use shifted constants.

An instruction can be made compact by using the smallest size that fits the actual value of the constant.

Mask register and fallback register

The 3-bit mask field in formats with templates of type A or E indicates a mask register. Register r0-r6 can be used as masks if the destination is a general purpose register. Vector register v0-v6 can be used as masks if the destination is a vector register. A value of 7 in the mask field means no mask and unconditional execution using the options specified in the numeric control register.

If the mask is a vector register then it is interpreted as a vector with the same element size as indicated by the OT field. Each element of the mask register is applied to the corresponding element of the result.

The mask has multiple purposes. The primary purpose is for conditional execution. An instruction is not executed if bit 0 of the mask is zero. In this case, the destination will get a fallback value instead of the result of the calculation, and any numerical error condition will be suppressed. Vector instructions are executed conditionally for each vector element separately, so that each vector element is enabled if bit 0 of the corresponding vector element of the mask register is 1.

The fallback value is taken from an extra register if the instruction has less than three source operands and the format has a vacant register field, or from the first source register operand otherwise. The fallback cannot be different from the first source register if the instruction has three source operands, even if there is a vacant register field. If the instruction format has more than one vacant register field, then the field that would be used for the first source operand if the instruction had three source operands is used for the fallback register.

Register r31 (stack pointer) and v31 cannot be used as fallback register. Instead, the fallback value will be zero if a register number of 31 is indicated. Register r31 and v31 should not be used as first source register if it is also used as feedback because this would cause ambiguity about the fallback value. (The fallback value will not be zero in this case).

A memory write has no fallback register. Instead, the value of the memory operand will be unchanged if the mask has a zero in bit 0.

The remaining bits of the mask are used for specifying various options. The meanings of these mask bits are described in the next section.

3.3 Coding of masks

A mask register can be a general purpose register r0-r6 or a vector register v0-v6. A value of 7 in the mask field means no mask.

The bits in the mask register are coded as follows.

Table 3.3: Bits in mask register and numeric control register

Bit number	Meaning
0 1	Predicate or mask. The operation is executed only if this bit is one. Guaranteed to be ignored.
2-7 2 3 4 5	Numerical exception control. See page 106. Floating point division by zero generates NAN Floating point overflow generates NAN Floating point underflow generates NAN Floating point inexact generates NAN Bits 2-7 may also be used for controlling integer division by zero and integer overflow
10-12	Floating point rounding mode: 000 = nearest or even 001 = down 010 = up 011 = towards zero This feature is optional.
13	Support subnormal numbers in single and higher precision. (Subnormal numbers are always supported for half precision). This feature is optional.
18-23	Instruction-specific option bits.
26 - 30	Possible use for enabling numerical traps. Not used in the standard version.
31	Constant execution time. This bit makes instructions take the same number of clock cycles regardless of the values of mask and operands. The guarantee provided by this bit is useful for cryptographic applications. This feature is optional.

Bits 8, 16, 24, etc. in a vector mask register can be used like bit 0 for 8-bit and 16-bit operand sizes. All other bits are reserved for future use.

Vector instructions treat the mask register as a vector with the same element size (OS) as the operands. Each element of the mask vector has the bit codes as listed above. The different vector elements can have different mask bits.

The numeric control register (NUMCONTR) is used as mask when the mask field is 7 or absent. The NUMCONTR register is broadcast to all elements of a vector, using as many bits of NUMCONTR as indicated by the operand size, when an instruction has no mask register. The number of bits in NUMCONTR is implementation dependent (usually 16 or more). Any missing bits will

be zero. The same NUMCONTR value is applied to all vector elements. Bit 0 of NUMCONTR is always 1.

The instruction-specific option bits (bit 18-23) may be used for various options in specific instructions. The option bits in the mask are considered zero in vector operands with an 8-bit or 16-bit operand type because each mask element has too few bits in this case.

3.4 Format for jump, call and branch instructions

Most branches in software are based on the result of an arithmetic or logic instruction (ALU). The ForwardCom design combines the ALU instruction and the conditional jump into a single instruction. For example, a loop control can be implemented with a single instruction that counts down and jumps until it reaches zero or counts up until it reaches a certain limit.

The jumps, calls, branches, and multiway branches will use the following formats.

Table 3.4: List of formats for control transfer instructions

Format	IL	Mode	OP1	Template	Description
1.6 A	1	6	OPJ	B	Multiway jump and calls with three register operands.
1.6 B	1	6	OPJ	B	Short jump with two register operands (RD, RS) and 8 bit offset (IM1).
1.7 C	1	7	OPJ	C	Short jump with one register operand (RD), an 8-bit immediate constant (IM2) and 8 bit offset (IM1).
1.7 D	1	7	0-15	D	Jump or call with 24-bit offset.
2.5.0	2	5	3	A2	Double size jump with three register operands (RD, RS, RT), and a 24-bit address offset (IM2). OPJ in upper 8 bits of IM2.
2.5.1	2	5	1	B2	Double size jump with a register destination operand, a register source operand, a 16-bit immediate operand (IM2 lower half), and a 16-bit jump offset (IM2 upper half). OPJ in IM1.
2.5.2	2	5	2	B2	Double size jump with one register operand (RD), a memory operand with base RS and 16-bit address offset (IM2 lower half), and a 16-bit jump offset (IM2 upper half). OPJ in IM1. Optional.
2.5.4	2	5	4	C2	Double size jump with one register operand (RD), one 8-bit immediate constant (IM2) and 32 bit offset (IM3). OPJ in IM1.
2.5.5	2	5	5	C2	Double size jump with one register operand (RD), an 8-bit offset (IM2) and a 32-bit immediate constant (IM3). OPJ in IM1.
2.5.7	2	5	7	C2	Double size system call, 16 bit constant (IM1,IM2) and 32-bit constant (IM3). No OPJ.

3.1.0	3	1	0	A3	Triple size jump with two register operands (RD, RT), a 24-bit jump offset (IM2), and a memory operand with base RS and 32-bit address offset (IM3). OPJ in last byte of IM2. Optional.
3.1.1	3	1	1	B3	Triple size jump with a register destination operand, a register source operand (RS), a 32-bit jump offset (IM2), and a 32-bit immediate operand (IM3). OPJ in IM1. Optional.

The jump, call, and branch instructions have signed offsets of 8, 16, 24, or 32 bits relative to the instruction pointer. Or, more precisely, relative to the end of the instruction. This offset is multiplied by the instruction word size (= 4 bytes) to cover an address range of $\pm$ 512 bytes for short conditional jumps with 8 bits offset, $\pm$ 128 kilobytes for jumps and calls with 16 bits offset, $\pm$ 32 megabytes for 24 bits offset, and $\pm$ 8 gigabytes for 32 bits offsets.

The OPJ field defines the operation and jump condition. This field has 6 bits in the single size version and 8 bits in the longer format versions. The two extra bits in the longer versions are reserved for future use.

The versions with template C and C2 have no OT field. The operand type is 32-bit integer when there is no OT field, unless otherwise noted. It is not possible to use formats with template C or C2 with other operand types.

The instructions will use vector registers when there is an OT field and M = 1. In other words, the combined ALU-and-branch instructions will use vector registers only when a floating point type is specified (or 128-bit integer type, if supported). General purpose registers are used in all other cases. Only the first element of a vector register is used. Logical instructions will interpret the value in a vector register as an integer, when a floating point type is specified. Only the compare instructions interpret the operands as floating point when a floating point type is indicated. Branch instructions with addition and subtraction cannot use floating point operands. The codes that these instructions would use are used for floating point compare instructions instead.

The combined ALU and conditional jump instructions can be coded in the formats listed above. Subtraction with a constant cannot be coded in format 1.7 C. The assembler will replace subtraction with a small immediate constant by addition with the negative constant. The code space that would have been used by subtraction in format 1.7 C is instead used for coding direct jump and call instructions with a 24-bit offset using format 1.7 D, where the lower three bits of OP1 are used as part of the 24-bit offset.

Unconditional and indirect jumps and calls use the formats indicated above, where unused fields must be zero. Bit 0 of the OPJ field is zero for unconditional jump instructions and one for call instructions.

See page 46 for a list of OPJ condition codes.

3.5 Assignment of opcodes

The opcodes and formats for new instructions can be assigned according to the following rules.

- Multi-format instructions. Often-used instructions that need to support many different operand types, addressing modes, and formats use all or most of the following formats: 0.0 - 0.9, 2.0.x, 2.1, 2.2.x, 2.3, 2.4, 2.8, 3.0.x, 3.2.x, 3.3, and 3.8. The same value of OP1 is used in all these formats. OP2 must be 0, except in formats 2.0.5 and 2.2.5 that use OP2 for other purposes. Instructions with few source operands should have the lowest values of OP1.

Available OP1 values is a limited resource that should be economized. Instructions for integers only and instructions for floating point only may share the same OP1 value.

- Control transfer instructions, i. e. jumps, branches, calls and returns, can be coded as short instructions with IL = 1, mode = 6 - 7, and OP1 = 0 - 63 or as double-size instructions with IL = 2, mode = 5, OP1 = 0 - 7, and optionally as triple-size instructions with IL = 3, mode = 1, OP1 = 0-7. See page 29.
- Short single-format instructions with general purpose registers. Use mode 1.0, 1.1, and 1.8, with any value of OP1. Mode 1.0 is currently unused and may be reserved for future purposes.
- Short single-format instructions with vector registers. Use mode 1.2, 1.3, and 1.4 with any value of OP1.
- Double-size single-format instructions with general purpose registers can use mode 2.9 with any value of OP1, and mode 2.0.x (except 2.0.5) with any value of OP1 and OP2 $\neq 0$ (give similar instructions the same value of OP2). If more combinations are needed then use IM3 for further subdivision of the code space.
- Double-size single-format instructions with vector registers can use mode 2.6 with with any value of OP1, and mode 2.2.x (except 2.2.5) with any value of OP1 and OP2 $\neq 0$ (give similar instructions the same value of OP2). If more combinations are needed then use IM3 for further subdivision of the code space.
- Double-size single-format instructions with mixed vector and general purpose registers or with memory operands can use mode 2.5 with OP1 in the range 8-63.
- Triple-size single-format instructions with general purpose registers can use mode 3.0.x with with any value of OP1 and OP2 $\neq 0$.
- Triple-size single-format instructions with vector registers can use mode 3.2.x with with any value of OP1 and OP2 $\neq 0$.
- Triple-size single-format instructions with mixed register types can use mode 3.1 with with OP1 in the range 8-63.
- Possible future instructions longer than three 32-bit words should be coded with IL = 3, mode = 4-7.
- New options or other modifications to existing instructions can use IM3 bits in template E or mask register bits.
- New addressing modes and formats may be implemented as single-format read and write instructions. Template E formats use Mode2 for distinguishing between different formats. Other single-format templates may be divided into groups of eight consecutive OP1 values with the same format. New addressing modes or other formats that apply to all multi-format instructions can use vacant values of Mode2 with E templates.
- Format 1.0 is intended for single-format instructions with three general purpose registers. There are currently no such instructions. Therefore, format 1.0 A or B may be used for application-specific single-size instructions or for other purposes. Note that the M bit is not available in format 1.0 because this bit is used for distinguishing format 1.8 from 1.0. This means that format 1.0 cannot be used for vector instructions without violating the general coding scheme.
- Format 1.5 is vacant to use for single-format instructions with vector registers.

Application-specific instructions may preferably use E template formats with OP2 $\neq 0$. There are many vacant opcodes in these formats. General multi-purpose instructions may use some of the more crowded formats.

Unused register fields may have the same value as the first source register operand in order to avoid false dependences. Unused mask fields have the value 7 in instructions that can have a mask. All other unused fields must be zero. The instructions with the fewest input operands should preferably have the lowest OP1 codes.

The file forwardcom_sourcecode_documentation has a checklist of what to do when making or modifying instructions.

Chapter 4

Instruction lists

The ForwardCom instructions are listed in a comma-separated file instruction_list.csv. This file is intended for use by assemblers, disassemblers, debuggers and emulators. The list is preliminary and subject to possible changes. Please remember to keep the lists in this document and the list in the instruction_list.cvs file synchronized.

The instruction list file has the following fields:

Table 4.1: Fields in instruction list file

Field	**Meaning**
Name	Name of instruction as used by assembler.
Category	1: single format instruction, 2: unused, 3: multi-format instruction, 4: jump instruction.
Formats	See table 4.2 below.
Template	Hexadecimal number: 0xA - 0xE for template A - E, 0x0 for multiple templates.

Variant	D0: No destination operand, no operand type. D1: No destination operand, but operand type specified. D2: Operand type ignored. D3: Destination register used for other purpose. F0: Can have mask register, but not fallback register. F1: Can have fallback register without mask register. I2: Immediate source operand is integer regardless of specified operand type. M0: Memory operand is destination. On: n bits of IM3 in E template format used for options (IM3 can be used for shift count only if it is not used for options). R0: Destination is a general purpose register. R1: First source operand is a general purpose register. R2: Second source operand is a general purpose register. RL: RT is a general purpose register specifying vector length. U0: Integer operands are unsigned. U3: Integer operands are unsigned if option bit 3 is set. (compare instruction). H0: Half precision floating point instruction. X0: Source register can be a special pointer (threadp, datap, ip). X1: Source register is special register. X2: Source register is capabilities register. X3: Source register is performance monitor register. X4: Source register is system register. Y0-4:Destination register is one of the above.
Source operands	Number of source operands, including register, memory and immediate operands, but not including mask, option bits, vector length, and index.
OP1	Operation code OP1.
OP2	Additional operation code OP2. Zero if none.
Operand types general purpose registers	Hexadecimal number indicating required and optional support for each operand type with general purpose registers. See table 4.3 below for meaning of each bit.
Operand types scalar	Hexadecimal number indicating required and optional support for each operand type for scalar operations in vector registers. See table 4.3 below for meaning of each bit.
Operand types vector	Hexadecimal number indicating required and optional support for each operand type for vector operations. See table 4.3 below for meaning of each bit.
Immediate operand type	Type of immediate operand for single-format instructions. See table 4.4 below.
Description	Description of the instruction and comments.

Table 4.2: Meaning of formats field in instruction list file

Category	Interpretation of formats field
1. Single format instruction	Number with three hexadecimal digits. The leftmost digit is the value of the IL field (0-3). The middle digit is he value of mode field or the combined M+mode field (0-9). The rightmost digit is the sub-mode defined by OP2 in E template modes or OP1 in mode 2.5.x. Zero otherwise. For example 0x223 means format 2.2.3.
3. Multi-format instruction	Hexadecimal number composed of one bit for each format supported: 0x0000001 Format 0.0: three general purpose registers. 0x0000002 Format 0.1: two general purpose registers, 8-bit immediate. 0x0000004 Format 0.2: Three vector registers. 0x0000008 Format 0.3: Two vectors, 8-bit immediate. 0x0000010 Format 0.4: One vector, memory operand. 0x0000020 Format 0.5: One vector, memory operand with negative index. 0x0000040 Format 0.6: One vector, scalar memory operand with index. 0x0000080 Format 0.7: One vector, scalar memory operand with 8-bit offset. 0x0000100 Format 0.8: One g. p. register, memory operand with index. 0x0000200 Format 0.9: One g. p. register, memory operand with 8-bit offset. 0x0001000 Format 2.8: Three g. p. registers, 32-bit immediate. 0x0002000 Format 2.1: Two g. p. registers, memory with 32-bit offset. 0x0004000 Format 2.3: Three vector registers, 32-bit immediate. 0x0008000 Format 2.4: One vector register, memory with 32-bit offset. 0x0010000 Format 2.0.0: Three g. p. reg., memory with 16-bit offset. 0x0020000 Format 2.0.1: Two g. p. reg., memory with unscaled index. 0x0040000 Format 2.0.2: Two g. p. reg., memory with scaled index. 0x0080000 Format 2.0.3: Two g. p. reg., memory with index and limit. 0x0400000 Format 2.0.6: Four g. p. reg. 0x0800000 Format 2.0.7: Three g. p. registers, 16-bit shifted immediate. 0x1000000 Format 2.2.0: Two vector reg., scalar memory w. 16-bit offset. 0x2000000 Format 2.2.1: Two vector reg., memory with 16-bit offset. 0x4000000 Format 2.2.2: Two vector reg., memory with negative index. 0x8000000 Format 2.2.3: Two vector reg., scalar memory w. index and limit.

	0x40000000	Format 2.2.6: Four vector reg.
	0x80000000	Format 2.2.7: Three vector registers, 16-bit shifted immediate.
	0x100000000	Format 3.8: Three g. p. registers, 64-bit immediate.
	0x40000 0000	Format 3.3: Three vector registers, 64-bit immediate.
	0x100000 0000	Format 3.0.0: Three g. p. reg., memory with 32-bit offset.
	0x800000 0000	Format 3.0.3: Two g. p. reg., memory with index and 32-bit limit.
	0x2000000 0000	Format 3.0.5: One g. p. reg., memory with index and 16-bit offset, 32-bit immediate.
	0x8000000 0000	Format 3.0.7: Three g. p. registers, 32-bit shifted immediate.
	0x10000000 0000	Format 3.2.0: Two vector reg., scalar memory w. 32-bit offset.
	0x20000000 0000	Format 3.2.1: Two vector reg., memory with 32-bit offset.
	0x80000000 0000	Format 3.2.3: Two vector reg., scalar memory index and 32-bit limit.
	0x200000000 0000	Format 3.2.5: One vector reg., memory with 16-bit offset, and 32-bit immediate.
	0x800000000 0000	Format 3.2.7: Three vector registers, float or 32-bit shifted immediate.
4. Jump instruction	Hexadecimal number composed of one bit for each format supported:	
	0x00001	Format 1.6.0 B: Two registers, 8 bit offset.
	0x00002	Format 1.7.1 C: One register, 8 bit immediate, 8 bit offset.
	0x00010	Format 2.5.0 A: Three registers, 24 bit offset.
	0x00020	Format 2.5.1 B: Two registers, 16 bit immediate, 16 bit offset.
	0x00040	Format 2.5.2 B: One register, memory operand with 16 bit address, 16 bit offset.
	0x00080	Format 2.5.3 B: Unused.
	0x00100	Format 2.5.4 C: One register, 8 bit immediate, 32 bit offset.
	0x00200	Format 2.5.5 C: One register, 32 bit immediate, 8 bit offset.
	0x01000	Format 3.1.0 A: Two registers, memory operand w 32 bit address, 24 bit offset.
	0x02000	Format 3.1.1 B: Two registers, 32 bit immediate, 32 bit offset.
	0x10000	Format 1.6.1 B: Memory operand with 8 bit offset.
	0x20000	Format 1.6.2 A: Reg. and memory w. scaled index.
	0x40000	Format 1.6.3 A: Three registers.
	0x100000	Format 1.7.0 D: No register, 24 bit address.
	0x400000	Format 1.7.3 C: One register.
	0x800000	Format 1.7.4 C: 16 bit immediate.
	0x1000000	Format 1.7.5 C: 16 bit fixed immediate.

	0x2000000	Format 1.7.A C: Format 1.7 with 64 bit operand size.
	0x10000000	Format 2.5.1 X: Two registers, 2x16 bit immediate.
	0x20000000	Format 2.5.2 X: One register, memory operand with 32 bit offset.
	0x40000000	Format 2.5.4 X: 64 bit operand size.
	0x80000000	Format 2.5.5 X: Conditional trap.
	0x100000000	Format 2.5.7 C: System call, 16 bit function, 32 bit module.
	0x1000000 0000	Format 3.1.1 X: System call, 32 bit function, 32 bit module.

Table 4.3: Indication of operand types supported for general purpose registers, scalars in vector registers, or vectors. The value is a hexadecimal number composed of one bit for each operand type supported

0x0001	8-bit integer supported.
0x0002	16-bit integer supported.
0x0004	32-bit integer supported.
0x0008	64-bit integer supported.
0x0010	128-bit integer supported.
0x0020	single precision floating point supported.
0x0040	double precision floating point supported.
0x0080	quadruple precision floating point supported.
0x0100	8-bit integer optionally supported.
0x0200	16-bit integer optionally supported.
0x0400	32-bit integer optionally supported.
0x0800	64-bit integer optionally supported.
0x1000	128-bit integer optionally supported.
0x2000	single precision floating point optionally supported.
0x4000	double precision floating point optionally supported.
0x8000	quadruple precision floating point optionally supported.

Table 4.4: Immediate operand type for single-format instructions

0	none or multi-format.
2	8-bit signed integer.
3	16-bit signed integer.
4	32-bit signed integer.
5	64-bit signed integer.
6	8-bit signed integer shifted by specified count.
7	16-bit signed integer shifted by specified count.
8	16-bit signed integer shifted by 16.
9	32-bit signed integer shifted by 32.
18	8-bit unsigned integer.
19	16-bit unsigned integer.
20	32-bit unsigned integer.
21	64-bit unsigned integer.
24	two 8-bit unsigned integers.

25	two 8-bit and one 6-bit unsigned integers.
26	two 16-bit unsigned integers.
27	one 16-bit and one 32-bit unsigned integer.
28	two 32-bit unsigned integers.
29	one 16-bit and two 8-bit unsigned integers.
34	8-bit signed integer converted to float.
35	16-bit signed integer converted to float.
64	half precision floating point.
65	single precision floating point.
66	double precision floating point.
100	determined by operand type.
in	a number prefixed by 'i' indicates an implicit value. The implicit immediate operand with this value does not need to be written in the assembly code.

Jump instructions are listed on page 46. All other categories of instructions are listed in the following tables.

4.1 List of multi-format instructions

The following list covers general instructions that can be coded in most or all of the formats assigned to multi-format instructions.

Table 4.5: List of multi-format instructions

Instruction	OP1	Source operands	Description
nop	0	0	No operation.
store	1	1	Store value to memory.
move	2	1	Copy value.
prefetch	3	1	Prefetch from memory.
sign_extend	4	1	Sign-extend smaller integer to 64 bits.
sign_extend_add	5	2	Sign-extend smaller integer to 64 bits and add 64-bit register.
compare	7	2	Compare. Uses condition codes, see p. 65.
add	8	2	src1 + src2.
sub	9	2	src1 - src2.
sub_rev	10	2	src2 - src1.
mul	11	2	src1 · src2.
mul_hi	12	2	(src1 · src2) >> OS, signed (integer only).
mul_hi_u	13	2	(src1 · src2) >> OS, unsigned (integer only).
div	14	2	src1 / src2, signed division (optional for integer vectors).
div_u	15	2	src1 / src2, unsigned integer division (optional for vectors).
div_rev	16	2	src2 / src1, signed division (optional for integer vectors).
rem	18	2	Modulo or remainder, signed (optional for integer vectors).
rem_u	19	2	Modulo or remainder, unsigned (optional for integer vectors).
min	20	2	Signed minimum.
min_u	21	2	Minimum. unsigned for integers, abs for f.p.
max	22	2	Signed maximum.
max_u	23	2	Maximum. unsigned for integers, abs for f.p.
and	26	2	src1 & src2.

or	27	2	src1 | src2.
xor	28	2	src1 ^ src2.
mul_2pow	32	2	src1 * 2^{src2}. Multiply by integer power of 2. Floating point only.
shift_left	32	2	src1 $<<$ src2. Shift left. Integer only.
rotate	33	2	Rotate left if src2 positive, right if negative.
shift_right_s	34	2	src1 $>>$ src2. Integer shift right with sign extension.
shift_right_u	35	2	src1 $>>$ src2. Integer shift right with zero extension.
clear_bit	36	2	Clear bit. src1 & ~ (1 $<<$ src2).
set_bit	37	2	Set bit. src1 | (1 $<<$ src2).
toggle_bit	38	2	Toggle bit. src1 ^ (1 $<<$ src2).
test_bit	39	2	Test single bit. (src1 $>>$ src2) & 1.
test_bits_and	40	2	Test if all indicated bits are 1. (src1 & src2) == src2
test_bits_or	41	2	Test if at least one indicated bit is 1. (src1 & src2) != 0
add	44	2	src1 + scr2 (float16. optional).
sub	45	2	src1 - scr2 (float16. optional).
mul	46	2	src1 * scr2 (float16. optional).
mul_add	48	3	$\pm$ src1 $\cdot$ src2 $\pm$ src3 (float16. optional).
mul_add	49	3	$\pm$ src1 $\cdot$ src2 $\pm$ src3 (optional).
mul_add2	50	3	$\pm$ src1 $\cdot$ src3 $\pm$ src2 (optional).
add_add	51	3	$\pm$ src1 $\pm$ src2 $\pm$ src3 (optional).
select_bits	52	3	src1 & src3 | src2 & ~src3
funnel_shift	53	3	Concatenate src1 and src2 and shift right by src3.
userdef56 - userdef62	56-62	2	Reserved for user-defined instructions.
undef	63	2	Undefined code. Generates trap.

4.2 List of single-format instructions

These instructions are mostly available in only one or a few formats.

Table 4.6: List of single-format instructions with general purpose registers

Instruction	Format	OP1	Description
move	1.1 C	0	Move 16-bit sign-extended constant to 32-bit general purpose register.
move	1.1 C	1	Move 16-bit sign-extended constant to 64-bit general purpose register.
move	1.1 C	3	Move 16-bit zero-extended constant to 64-bit general purpose register.
move	1.1 C	4	RD = IM2 « IM1. Sign-extend IM2 to 32 bits and shift left by the unsigned value IM1.
move	1.1 C	5	RD = IM2 « IM1. Sign-extend IM2 to 64 bits and shift left by the unsigned value IM1.
add	1.1 C	6	Add 16-bit sign-extended constant to 32-bit general purpose register..
mul	1.1 C	8	Multiply 32-bit general purpose register by 16-bit sign-extended constant.
add	1.1 C	10	RD += IM2 « IM1. Sign-extend IM2 to 32 bits, shift left by the unsigned value IM1, add to RD.
add	1.1 C	11	RD += IM2 « IM1. Sign-extend IM2 to 64 bits, shift left by the unsigned value IM1, add to RD.

and	1.1 C	12	RD &= IM2 « IM1. Sign-extend IM2 to 32 bits, shift left by the unsigned value IM1, AND with RD.
and	1.1 C	13	RD &= IM2 « IM1. Sign-extend IM2 to 64 bits, shift left by the unsigned value IM1, AND with RD.
or	1.1 C	14	RD |= IM2 « IM1. Sign-extend IM2 to 32 bits, shift left by the unsigned value IM1, OR with RD.
or	1.1 C	15	RD |= IM2 « IM1. Sign-extend IM2 to 64 bits, shift left by the unsigned value IM1, OR with RD.
xor	1.1 C	16	RD ^= IM2 « IM1. Sign-extend IM2 to 32 bits, shift left by the unsigned value IM1, XOR with RD.
xor	1.1 C	17	RD ^= IM2 « IM1. Sign-extend IM2 to 64 bits, shift left by the unsigned value IM1, XOR with RD.
add	1.1 C	18	RD += (IM1,IM2) « 16. Shift 16-bit zero-extended constant left by 16 and add to 32-bit general purpose register.
abs	1.8 B	0	Absolute value of integer. IM1 determines handling of overflow: 0: wrap around, 1: saturate, 2: zero.
bitscan	1.8 B	2	Bit scan forward or reverse. Find index to first or last set bit.
roundp2	1.8 B	3	Round up or down to nearest power of 2.
popcount	1.8 B	4	Count the number of bits that are 1.
read_spec	1.8 B	32	Read special register RS into g. p. register RD.
write_spec	1.8 B	33	Write g. p. register RS to special register RD.
read_capabilities	1.8 B	34	Read capabilities register RS into g. p. register RD.
write_capabilities	1.8 B	35	Write g. p. register RS to capabilities register RD.
read_perf	1.8 B	36	Read performance counter.
read_perfs	1.8 B	37	Read performance counter, serializing.
read_sys	1.8 B	38	Read system register RS into g. p. register RD.
write_sys	1.8 B	39	Write g. p. register RS to system register RD.
push	1.8 B	56	Push g. p. register RS to stack with pointer RD.
pop	1.8 B	57	Pop g. p. register RS from stack with pointer RD.
input	1.8 B	62	Read RD from input port with address IM1 or RS. (privileged instruction)
output	1.8 B	63	Write RD to output port with address IM1 or RS. (privileged instruction)
truth_tab3	2.0.6 E	8.1	Boolean function of three inputs, given by a truth table.
move_bits	2.0.7 E	0.1	Replace one or more contiguous bits at one position of RS with contiguous bits from another position of RT. Optional.
move	2.9 A	0	Load 32-bit constant into the high part of a general purpose register. The low part is zero. RD = IM2 « 32.
insert_hi	2.9 A	1	Insert 32-bit constant into the high part of a general purpose register, leaving the low part unchanged. RD = (RT & 0xFFFFFFFF) | (IM2 « 32).
add	2.9 A	2	Add zero-extended 32-bit constant to general purpose register.
sub	2.9 A	3	Subtract zero-extended 32-bit constant from general purpose register.
add	2.9 A	4	Add 32-bit constant to high part of general purpose register. RD = RT + (IM2 « 32).

and	2.9 A	5	AND high part of general purpose register with 32-bit constant. RD = RT & (IM2 « 32).
or	2.9 A	6	OR high part of general purpose register with 32-bit constant. RD = RT | (IM2 « 32).
xor	2.9 A	7	XOR high part of general purpose register with 32-bit constant. RD = RT ^ (IM2 « 32).
address	2.9 A	32	RD = RT + IM2, RT can be THREADP (28), DATAP (29) or IP (30).

Table 4.7: List of single-format instructions with vector registers and mixed register types

Instruction	**Format**	**OP1. OP2**	**Description**
get_len	1.2 A	0	Get length of vector register RT into general purpose register RD.
get_num	1.2 A	1	Get length of vector register RT divided by the operand size.
set_len	1.2 A	2	RD = vector register RS with length changed to value of RT.
set_num	1.2 A	3	Change the length of vector register RS to RT·OS.
insert	1.2 A	4	Replace one element in vector RD, starting at offset RT·OS, with scalar RS.
extract	1.2 A	5	Extract one element from vector RS, starting at offset RT·OS, with size OS into scalar in vector register RD.
broad	1.2 A	6	Broadcast first element of vector RS into all elements of RD with length RT bytes.
compress_ sparse	1.2 A	8	Compress sparse vector elements indicated by mask bits into contiguous vector. (optional).
ex- pand_sparse	1.2 A	9	Expand contiguous vector into sparse vector with positions indicated by mask bits. RT = length of output vector. (optional).
bits2bool	1.2 A	12	The lower n bits of RT are unpacked into a boolean vector RD with length RS, with one bit in each element, where n = RS / OS.
shift_expand	1.2 A	16	Shift vector RS up by RT bytes and extend the vector length by RT. The lower RT bytes of RD will be zero.
shift_reduce	1.2 A	17	Shift vector RS down RT bytes and reduce the length by RT. The lower RT bytes are lost.
shift_up	1.2 A	18	Shift elements of vector RS up RT elements. The lower RT elements of RD will be zero, the upper RT elements are lost.
shift_down	1.2 A	19	Shift elements of vector RS down RT elements. The upper RT elements of RD will be zero, the lower RT elements are lost.
div_ex	1.2 A	24	Divide vector of double-size signed integers RS by signed integers RT. RS has element size 2·OS. These are divided by the even numbered elements of RT with size OS. The truncated results are stored in the even-numbered elements of RD. The remainders are stored in the odd-numbered elements of RD. (Optional for vectors).
div_ex_u	1.2 A	25	Same, with unsigned integers. (Optional for vectors).

mul_ex	1.2 A	26	Multiply even-numbered signed integer vector elements to double size result.
mul_ex_u	1.2 A	27	Multiply even-numbered unsigned integer vector elements to double size result.
sqrt	1.2 A	28	Square root (floating point, optional).
add_ss	1.2 A	32	Add integer vectors, signed with saturation (optional).
add_us	1.2 A	33	Add integer vectors, unsigned with saturation (optional).
sub_ss	1.2 A	34	Subtract integer vectors, signed with saturation (optional).
sub_us	1.2 A	35	Subtract integer vectors, unsigned with saturation (optional).
mul_ss	1.2 A	36	Multiply integer vectors, signed with saturation (optional).
mul_us	1.2 A	37	Multiply integer vectors, unsigned with saturation (optional).
add_oc	1.2 A	38	add with overflow check (optional).
sub_oc	1.2 A	39	subtract with overflow check (optional).
mul_oc	1.2 A	40	multiply with overflow check (optional).
div_oc	1.2 A	41	divide with overflow check (optional).
add_c	1.2 A	42	Add with carry. Vector has two elements. The upper element is used as carry on input and output (optional).
sub_b	1.2 A	43	Subtract with borrow. Vector has two elements. The upper element is used as borrow on input and output (optional).
read_spev	1.2 A	56	read special vector register. Length RT.
read_call_stack	1.2 A	58	read internal call stack. RD = vector register destination of length RS, RT-RS = internal address (privileged instruction).
write_call_stack	1.2 A	59	write internal call stack. RD = vector register source of length RS, RT-RS = internal address (privileged instruction).
read_memory_ map	1.2 A	60	read memory map. RD = vector register destination of length RS, RT-RS = internal address (privileged instruction).
write_memory_ map	1.2 A	61	write memory map. RD = vector register source of length RS, RT-RS = internal address (privileged instruction).
input	1.2 A	62	read from input port. RD = vector register, RT = port address, RS = vector length (privileged instruction).
output	1.2 A	63	write to output port. RD = vector register source operand, RT = port address, RS = vector length (privileged instruction).
gp2vec	1.3 B	0	Move value of general purpose register RS to scalar in vector register RD.
vec2gp	1.3 B	1	Move value of first element of vector register RS to general purpose register RD.
make_sequence	1.3 B	3	Make a vector with RS sequential numbers. First value is IM1.
insert	1.3 B	4	Replace one element in vector RD, starting at offset IM1·OS, with first element in RS.
extract	1.3 B	5	Extract one element from vector RS, starting at offset IM1·OS into a scalar in vector register RD.

compress	1.3 B	6	Compress vector to half the length and half the element size. Double precision → single precision, 64-bit integer → 32-bit integer, etc.
expand	1.3 B	7	Expand vector to the double length and the double element size. Half precision → single precision, 32-bit integer → 64-bit integer, etc.
float2int	1.3 B	12	Conversion of floating point to integer with the same operand size. The rounding mode is specified in IM1.
int2float	1.3 B	13	Conversion of integer to floating point with same operand size.
round	1.3 B	14	Round floating point to integer in floating point representation. The rounding mode is specified in IM1.
round2n	1.3 B	15	Round to nearest multiple of 2^n. RD = $2^n \cdot$ round($2^{-n} \cdot$ RS). n is a signed integer constant in IM1 (optional).
abs	1.3 B	16	Absolute value of integer. IM1 determines handling of overflow: 0: wrap around, 1: saturate, 2: zero.
fp_category	1.3 B	17	Check if floating point numbers belong to the categories indicated by constant.
broad	1.3 B	18	Broadcast 8-bit constant into all elements of RD with length RS (31 in RS field gives scalar output).
broadcast_max	1.3 B	19	Broadcast 8-bit constant into all elements of RD with maximum vector length.
byte_reverse	1.3 B	20	Reverse the order of bytes in each element of vector.
bit_reverse	1.3 B	20	Reverse the order of bits in each element of vector (optional).
bitscan	1.3 B	21	Bit scan forward or reverse. Find index to lowest set bit.
popcount	1.3 B	22	Count the number of bits that are 1 (optional for vectors).
bool2bits	1.3 B	25	A boolean vector with n elements is packed into the lower n bits of RD, taking bit 0 of each element. The length of RD is at least sufficient to contain n bits.
bool_reduce	1.3 B	26	An integer vector is reduced by combining bit 0 of all elements. The output is a scalar integer where bit 0 is the AND combination of all the bits, and bit 1 is the OR combination of all the bits. The remaining bits are reserved for future use.
category_reduce	1.3 B	26	A floating point vector is reduced to a scalar integer where each bit indicates that the source vector contains at least one element in a certain category, such as NAN, zero, normal positive, etc.
push	1.3 B	56	Push vector register RS to stack with pointer RD.
pop	1.3 B	57	Pop vector register RS from stack with pointer RD.
clear	1.3 B	58	Clear vector register RS.
move	1.4 C	0	Move 16 bit integer constant to 16-bit scalar (optional).
add	1.4 C	1	Add broadcasted 16 bit constant to 16-bit vector elements (optional).
and	1.4 C	2	AND broadcasted 16 bit constant with 16-bit vector elements (optional).
or	1.4 C	3	OR broadcasted 16 bit constant with 16-bit vector elements (optional).
xor	1.4 C	4	XOR broadcasted 16 bit constant with 16-bit vector elements (optional).

move	1.4 C	8	RD = IM2 « IM1. Sign-extend IM2 to 32 bits and shift left by the unsigned value IM1 to make 32 bit scalar (optional).
move	1.4 C	9	RD = IM2 « IM1. Sign-extend IM2 to 64 bits and shift left by the unsigned value IM1 to make 64 bit scalar (optional).
add	1.4 C	10	RD += IM2 « IM1. Add broadcast shifted signed constant to 32-bit vector elements (optional).
add	1.4 C	11	RD += IM2 « IM1. Add broadcast shifted signed constant to 64-bit vector elements (optional).
and	1.4 C	12	RD &= IM2 « IM1. AND broadcast shifted signed constant with 32-bit vector elements (optional).
and	1.4 C	13	RD &= IM2 « IM1. AND broadcast shifted signed constant with 64-bit vector elements (optional).
or	1.4 C	14	RD |= IM2 « IM1. OR broadcast shifted signed constant with 32-bit vector elements (optional).
or	1.4 C	15	RD |= IM2 « IM1. OR broadcast shifted signed constant with 64-bit vector elements (optional).
xor	1.4 C	16	RD ^= IM2 « IM1. XOR broadcast shifted signed constant with 32-bit vector elements (optional).
xor	1.4 C	17	RD ^= IM2 « IM1. XOR broadcast shifted signed constant with 64-bit vector elements (optional).
move	1.4 C	32	Move converted half precision floating point constant to single precision scalar (optional).
move	1.4 C	33	Move converted half precision floating point constant to double precision scalar (optional).
add	1.4 C	34	Add broadcast half precision floating point constant to single precision vector (optional).
add	1.4 C	35	Add broadcast half precision floating point constant to double precision vector (optional).
mul	1.4 C	36	Multiply broadcast half precision floating point constant with single precision vector (optional).
mul	1.4 C	37	Multiply broadcast half precision floating point constant with double precision vector (optional).
add_h	1.4 C	40	add constant to half precision vector (optional).
mul_h	1.4 C	41	multiply half precision vector with constant (optional).
concatenate	2.2.6 E	0.1	A vector RU of length RT and a vector RS of length RT are concatenated into a vector RD of length 2·RT.
permute	2.2.6 E	1.1	The vector elements of RU are permuted within each block of size RT bytes, using indices in RS. Each index is relative to the beginning of a block. An index out of range produces zero. The maximum block size is implementation dependent.
interleave	2.2.6 E	2.1	Interleave elements of vectors RU and RS of length RT/2 to produce vector RD of length RT. Even-numbered elements of the destination come from RU and odd-numbered elements from RS. (optional).
truth_tab3	2.2.6 E	8.1	Boolean function of three inputs, given by a truth table.
move_bits	2.2.7 E	0.1	Replace one or more contiguous bits at one position of RS with contiguous bits from another position of RT. Optional
mask_length	2.2.7 E	1.1	Make mask with true in the first RT bytes. Option bits in IM2.

repeat_block	2.2.7 E	8.1	Repeat a block of data to make a longer vector. RS is input vector containing data block to repeat. IM2 is length in bytes of the block to repeat (must be a multiple of 4). RT is the length of destination vector RD. (optional).
repeat_within_blocks	2.2.7 E	9.1	Broadcast the first element of each block of data in a vector to the entire block. RS is input vector containing data blocks. IM2 is length in bytes of each block (must be a multiple of the operand size). RT is length of destination vector RD. The operand size must be at least 4 bytes. (optional).
load_hi	2.6 A	0	Make vector of two elements. dest[0] = 0, dest[1] = IM2.
insert_hi	2.6 A	1	Make vector of two elements. dest[0] = src1[0], dest[1] = IM2.
make_mask	2.6 A	2	Make vector where bit 0 of each element comes from bits in IM2, the remaining bits come from RT.
replace	2.6 A	3	Replace elements in RT by constant IM2.
replace_even	2.6 A	4	Replace even-numbered elements in RT by constant IM2.
replace_odd	2.6 A	5	Replace odd-numbered elements in RT by constant IM2.
broad	2.6 A	6	Broadcast 32-bit or float32 constant into all elements of RD with length RT (31 in RT field gives scalar output).
permute	2.6 A	8	The vector elements of RS are permuted within each block of size RT bytes. The 4·n bits of IM2 are used as index with 4 bits for each element in blocks of size n. The same pattern is used in each block. The number of elements in each block, n = RT / OS $\leq$ 8.
replace	3.1 A	32	Replace elements in RT by constant IM2,IM3.
broad	3.1 A	33	Broadcast 64-bit or float64 constant into all elements of RD with length RT (31 in RT field gives scalar output).

Table 4.8: List of single-format instructions with memory operands.

Instruction	**Format**	**OP1, OP2**	**Description**
store	2.5 B	8	Store 32-bit constant IM2 to memory operand [RS+IM1] (optional).
fence	2.5 B	16	Memory fence at address [RS+IM2]. read, write or full indicated by IM1.
compare_swap	2.5 B	18	Atomic compare and exchange with address [RS+IM2].
read_insert	2.5 A	32	Replace one element in vector RD, starting at offset RT·OS, with scalar memory operand [RS+IM2] (optional).
extract_store	2.5 A	40	Extract one element from vector RD, starting at offset RT·OS, with size OS into memory operand [RS+IM2] (optional).

4.3 List of control transfer instructions

Table 4.9: Condition codes for control transfer instructions with integer operands in general purpose registers

OPJ	bit 0 of OPJ	Instruction	Comment
0-7	part of offset	Unconditional jump with 24-bit offset (jump)	Format 1.7 D. Bit 0-2 of OPJ are part of offset
8-15	part of offset	Unconditional call with 24-bit offset (call)	Format 1.7 D. Bit 0-2 of OPJ are part of offset
0-1	invert	sub/jump_zero, sub/jump_nzero	Not format 1.7. Not floating point
2-3	invert	sub/jump_neg, sub/jump_nneg	Not format 1.7. Not floating point
4-5	invert	sub/jump_pos, sub/jump_npos	Not format 1.7. Not floating point
6-7	invert	sub/jump_overfl, sub/jump_noverfl	Not format 1.7. Not floating point
8-9	invert	sub/jump_borrow, sub/jump_nborrow	Not format 1.7. Not floating point
10-11	invert	and/jump_zero and/jump_nzero	Not format 1.7
12-13	invert	or/jump_zero or/jump_nzero	Not format 1.7
14-15	invert	xor/jump_zero, xor/jump_nzero	Not format 1.7
16-17	invert	add/jump_zero, add/jump_nzero	Not floating point
18-19	invert	add/jump_neg, add/jump_nneg	Not floating point
20-21	invert	add/jump_pos, add/jump_npos	Not floating point
22-23	invert	add/jump_overfl, add/jump_noverfl	Not floating point
24-25	invert	add/jump_carry, add/jump_ncarry	Not floating point
26-27	invert	test_bit/jump_true, test_bit/jump_false	
28-29	invert	test_bits_and/jump_true, test_bits_and/jump_false	
30-31	invert	test_bits_or/jump_true, test_bits_or/jump_false	
32-33	invert	compare/jump_equal, compare/jump_nequal	
34-35	invert	compare/jump_sbelow, compare/jump_saboveeq	
36-37	invert	compare/jump_sabove, compare/jump_sbeloweq	
38-39	invert	compare/jump_ubelow, compare/jump_uaboveeq	

40-41	invert	compare/jump_uabove, compare/jump_ubeloweq	
42-47	invert	Reserved for future use.	
48-49	invert	increment_compare/jump_below, /jump_aboveeq	
50-51	invert	increment_compare/jump_above, /jump_beloweq	
52-53	invert	sub_maxlen/jump_pos, sub_maxlen/jump_npos	
54-57		Reserved for future use.	
58-59	0 jump 1 call	Indirect jump or call with memory operand.	Format 1.6 B and 2.5.2.
58-59	0 jump 1 call	Unconditional direct jump or call	2.5.4, and 3.1.1.
60-61	0 jump_ relative 1 call_ relative	Jump or call with relative address in memory, table index, and arbitrary reference point	Format 1.6 A and 2.5.2
60-61	0 jump 1 call	Indirect jump or call to value of register	Format 1.7 C
62	0	return	Format 1.6 C
62	0	sys_return	Format 1.7 C
63	1	sys_call. ID in register	Format 1.6 A
63	1	sys_call. ID in constants	Format 2.5.7 and 3.1.1.
63	1	trap or filler	Format 1.7 C
63	1	Conditional traps	Format 2.5.5.

Table 4.10: Condition codes for control transfer instructions with floating point operands in vector registers

OPJ	bit 0 of OPJ	Instruction	Comment
32-33	invert	compare/jump_equal, compare/jump_nequal	false if unordered
0-1	invert	compare/jump_equal_uo, compare/jump_nequal_uo	true if unordered
34-35	invert	compare/jump_below, compare/jump_aboveeq	false if unordered
2-3	invert	compare/jump_below_uo, compare/jump_aboveeq_uo	true if unordered
36-37	invert	compare/jump_above, compare/jump_beloweq	false if unordered
4-5	invert	compare/jump_above_uo, compare/jump_beloweq_uo	true if unordered
38-39	invert	compare/jump_abs_below, compare/jump_abs_aboveeq	false if unordered
6-7	invert	compare/jump_abs_below_uo, compare/jump_abs_aboveeq_uo	true if unordered
40-41	invert	compare/jump_abs_above, compare/jump_abs_beloweq	false if unordered
8-9	invert	compare/jump_abs_above_uo, compare/jump_abs_beloweq_uo	true if unordered

24-25	invert	fp_category/jump_true, fp_category/jump_false	
The following instructions treat floating point operands as integers in vector registers:			
10-11	invert	and/jump_zero and/jump_nzero	
12-13	invert	or/jump_zero or/jump_nzero	
14-15	invert	xor/jump_zero, xor/jump_nzero	
26-27	invert	test_bit/jump_true, test_bit/jump_false	
28-29	invert	test_bits_and/jump_true, test_bits_and/jump_false	
30-31	invert	test_bits_or/jump_true, test_bits_or/jump_false	

See page 86 for detailed descriptions of control transfer instructions.

Chapter 5

Description of instructions

Data move and conversion instructions

broad

format	op-code	operands
1.2 A	6	vector and g.p. register
1.3 B	18	g.p. register, and 8-bit signed constant
2.6	6	g.p. register, and 32-bit signed or float constant
3.1	33	g.p. register, and 64-bit signed or double constant

float v0 = broad(v1, r2)
float v0 = broad(r2, 2.5)

Broadcast a constant or the first element of a source vector into all elements of the destination vector with the length in bytes indicated by a general purpose register.

This instruction can have a mask but not a fallback register. The fallback value is zero. (This instruction is not called broadcast because that is a reserved keyword).

broadcast_max

format	op-code	operands
1.3 B	19	vector and 8-bit signed constant

float v0 = broadcast_max(1)

Broadcast a small constant to all elements of a vector with maximum length.

compress

format	op-code	operands
1.3 B	6	vectors

double v0 = compress(v1, 0)

All the elements of a vector are converted to half the element size. The length of the output vector will be half the length of the input vector. The OT field specifies the operand type of the input vector. Double precision floating point numbers are converted to single precision. Integer elements are converted to half the size. Support for the following conversions are optional: single precision float to half precision, quadruple precision to double precision, 8-bit integer to 4-bit.

Overflow options and rounding mode are specified in IM1 as follows:

IM1 bits	meaning
bit 0-2	Floating point exception control: 000 = exceptions are controlled by NUMCONTR. See page 106 001 = overflow generates NAN code 010 = underflow generates NAN code 011 = overflow and underflow generate NAN code 100 = underflow and inexact generate NAN code 101 = overflow, underflow, and inexact generate NAN code 111 = no conditions generate NAN code
bit 0-2	Integer overflow control: 000 = integer overflow wraps around 100 = signed integer overflow gives zero 101 = signed integer overflow gives signed saturation 110 = unsigned integer overflow gives zero 111 = unsigned integer overflow gives unsigned saturation
bit 3-5	Floating point rounding mode: 000 = rounding mode determined by NUMCONTR 001 = odd if not exact 100 = nearest or even 101 = down 110 = up 111 = towards zero

The rounding mode ”odd if not exact” works in the following way: Truncate the superfluous mantissa bits. If the result is not exact then set the least significant bit to 1. This rounding mode is needed to avoid double rounding errors when rounding in multiple steps. Use odd rounding mode except in the last step. For example, to convert from double precision to half precision, use the odd rounding mode in the first step from double to single precision, then use ”nearest or even” in the last step from single to half precision.

Overflow in integer conversion can be detected by doing the conversion twice, using an ”overflow gives zero” option and the corresponding saturation option. Overflow has occurred if the two results are different.

NANs are converted by preserving the least significant bits of the payload and the quiet bit. This differs from most other microprocessors, which preserve the most significant bits of binary floating point NAN payloads.

compress_sparse

format	op-code	operands
1.2 A	8	vectors. Optional

int32 v0 = compress_sparse(v1), mask = v2

Compress sparse vector elements indicated by mask bits into contiguous vector.

The algorithm of this instruction is: Start with a zero-length destination vector. For each element in the mask vector that is true, take an element from the corresponding position in the source vector and append it to the destination vector. The length of the destination vector will be the number of true mask elements times the element size.

This instruction cannot have a fallback register.

concatenate

format	op-code	operands
2.2.6	0.1	vectors

float v0 = concatenate(v1, v2, r3)

A vector v1 of length r3 bytes and a vector v2 of length r3 bytes are concatenated into a result vector of length 2·r3, with v2 in the high end.

This instruction cannot have a mask.

expand

format	op-code	operands
1.3 B	7	vectors

float v0 = expand(v1, 0)

This is the opposite of compress. The length of the output vector is double the length of the input vector if the maximum vector length is not exceeded.

The OT field specifies the operand type of the output vector. Single precision floating point numbers are converted to double precision. Integers are converted to the double size by sign-extension or zero-extension. Support for the following conversions are optional: half precision float to single precision, double precision to quadruple precision, 4-bit integer to 8-bit.

Options are specified in IM1:

IM1 bits	meaning
bit 0-1	integer options: 00 = sign extension 10 = zero extension

expand_sparse

format	op-code	operands
1.2 A	9	vectors. Optional

int32 v0 = expand_sparse(v1, r2), mask = v3

This is the opposite of compress_sparse.
Expand a contiguous vector into a sparse vector with positions indicated by mask bits.
The second operand is a general purpose register indicating the length in bytes of the output vector.

The algorithm of this instruction is:
Set an index i1 to position zero in the source vector.
Let another index i2 loop through the elements of the mask vector. For each i2 do:

```
   if mask[i2] then
      destination[i2] = source[i1]; increment i1
   else
      destination[i2] = 0
end for
```

The length of the destination vector will be the number of true mask elements times the element size. This instruction cannot have a fallback register.

extract

format	op-code	operands
1.2 A	5	vectors
1.3 B	5	vectors

float v0 = extract(v1, r2)
float v0 = extract(v1, 5)

Extract one element from the source vector at the given position and broadcast it into all elements of vector register RD with same length and operand size as the source vector. The index can be a constant or a general purpose register. This index indicates which vector element to extract. The size of the vector elements must match the operand type.

An index out of range will produce zero. An operand size of 128 bits can be used, even if this size is not otherwise supported. This instruction cannot have a mask.

float2int

format	op-code	operands
1.3 B	12	vectors

int32 v0 = float2int(v1, 0)

Conversion of floating point values to integers with the same operand size.
float16 is converted to int16. float32 is converted to int32. float64 is converted to int64.

The bits in IM1 specify rounding mode and error control, according to the following table:

IM1 bit	Meaning
0-2	overflow control: 000 = integer overflow wraps around 100 = signed integer overflow gives zero 101 = signed integer overflow gives signed saturation 110 = unsigned integer overflow gives zero 111 = unsigned integer overflow gives unsigned saturation
3-4	rounding mode: 00 = nearest or even 01 = down 10 = up 11 = truncate towards zero
5	0: NAN gives 0. 1: NAN gives MIN_INT

To check for overflow: Compare the results for overflow gives zero and overflow gives saturation.
To check if the result is exact: Compare the results for round down and round up.

get_len

format	op-code	operands
1.2 A	0	vectors

Get length in bytes of vector register RT into general purpose register RD.

This instruction cannot have a mask.

get_num

format	op-code	operands
1.2 A	1	vectors

Get the number of elements in vector register RT into general purpose register RD. This is equal to the length divided by the operand size. The result is a 64-bit integer.

This instruction cannot have a mask.

gp2vec

format	op-code	operands
1.3 B	0	g.p register in, vector register out

int64 v0 = gp2vec(r1)

Move integer value of general purpose register RS to scalar in vector register RD.

insert

format	op-code	operands
1.2 A	4	vectors
1.3 B	4	vectors

float v0 = insert(v0, v1, r2)
float v0 = insert(v0, v1, 5)

Replace one element in the first vector with the first element of the second vector. The index to the position of replacement can be a constant or a general purpose register. This index indicates which vector element to replace. The size of the vector elements must match the operand type. The destination register must be the same as the first source operand.

An index out of range will leave the vector unchanged. An operand size of 128 bits can be used, even if this size is not otherwise supported.

This instruction cannot have a mask.

insert_hi

format	op-code	operands
2.9	1	general purpose register, 32-bit immediate constant
2.6	1	vector register, 32-bit immediate constant

int64 r0 = insert_hi(r1, 2)
float v0 = insert_hi(v1, 2.1)

Insert 32-bit constant into the high part of a general purpose register, leaving the low part unchanged.
dest = (src1 & 0xFFFFFFFF) | (IM2 << 32).

Make a vector of two elements. A constant is inserted into the second element, leaving the first element unchanged.
dest[0] = src1[0], dest[1] = IM2.

int2float

format	op-code	operands
1.3 B	13	vectors

int64 v0 = int2float(v1, 0)

Conversion of signed or unsigned integers to floating point numbers with same operand size. int16 is converted to float16. int32 is converted to float32. int64 is converted to float64.

Options are coded in IM1:

IM1 bit number	Meaning
0	The integer is unsigned
2	Inexact result gives NAN. See page 106.

interleave

format	op-code	operands
2.2.6	2.1	vectors. Optional

float v0 = interleave(v1, v2, r3)

Interleave the inputs from two vectors, v1 and v2, so that the even-numbered elements come from v1 and the odd-numbered elements come from v2. The length in bytes of the destination vector is indicated by a general purpose register, r3. The length of each input vector is half the indicated value.

This instruction can have a mask but not a fallback register. The fallback value is zero.

load_hi

format	op-code	operands
2.5	0	vector. 32 bit immediate constant

float v0 = load_hi(1.2)

Make vector of two elements. dest[0] = 0, dest[1] = IM2.

move

format	op-code	operands
multi	2	all types
1.1 C	0	32-bit register = 16-bit sign-extended constant
1.1 C	1	64-bit register = 16-bit sign-extended constant
1.1 C	3	64-bit register = 16-bit zero-extended constant
1.1 C	4	32-bit register = 8-bit sign-extended constant with left shift
1.1 C	5	64-bit register = 8-bit sign-extended constant with left shift
1.4 C	0	vector register 16-bit scalar = 16-bit constant. Optional
1.4 C	8	vector register 32-bit scalar = 8-bit sign extended constant with left shift. Optional
1.4 C	9	vector register 64-bit scalar = 8-bit sign extended constant with left shift. Optional
1.4 C	32	vector register single precision scalar = half precision immediate constant. Optional
1.4 C	33	vector register double precision scalar = half precision immediate constant. Optional

Copy A value from a register, memory operand or immediate constant to a register. If the destination is a vector register and the source is an immediate constant then the result will be a scalar. The value will not be broadcast because there is no other input operand that specifies the vector length. If a vector is desired then use the broadcast instruction instead.

The move instruction with an immediate operand is the preferred method for setting a register to zero.

permute

format	op-code	operands
2.2.6	1.1	vectors
2.6	8	vectors and 32 bit immediate constant

float v0 = permute(v1, v2, r3)
float v0 = permute(v1, r3, 5)

This instruction permutes the elements of a vector v1. The vector is divided into blocks of size r3 bytes each. The block size must be a power of 2 and a multiple of the operand size. Elements can be moved arbitrarily between positions within each block, but not between blocks. Each element of the output vector is a copy of an element in the input vector, selected by the corresponding index in an index vector v2 or a constant. The indexes are relative to the start of the block they belong to, so that an index of zero will select the first element in the block of the input vector and insert it in the corresponding position of the output vector. The same element in the input vector can be copied to multiple elements in the output vector. An index out of range will produce a zero. The indexes are interpreted as integers regardless of the operand type.

The permute instruction has two versions. The first version specifies the indexes in a vector with the same length and element size as the input vector.

The second version specifies the indexes as a 32-bit immediate constant with 4 bits per element. This constant is split into a maximum of 8 elements with 4 bits in each, where the least significant four bits is index for the first element in the block. If the blocks have more than 8 elements each then the sequence of 8 elements is repeated to fill a block. The same pattern of indexes will be applied to all blocks in the second version of the permute instruction.

The maximum block size for the permute instruction is implementation-dependent and given by a special register. The reason for this limitation of block size is that the complexity of the hardware grows quadratically with the block size. A full permutation is possible if the vector length does not exceed the maximum block size. A trap is generated if r3 is bigger than the maximum block size.

The outputs of multiple permute instructions can be combined by using indexes out of range to produce zeroes for unused outputs and then combine the outputs of multiple permutes by bitwise OR. The fallback value is zero if a mask is used.

Permute instructions are essential for a vector processor because it is often necessary to rearrange data to facilitate the vector processing. These instructions are useful for reordering data, for transposing a matrix, etc.

Permute instructions can also be used for parallel table lookup when the block size is big enough to contain the entire table.

Finally, permute instructions can be used for gathering and scattering data within an area not bigger than the vector length or the block size.

read_insert

format	op-code	operands
2.5 A	32	vectors. Optional

int32 v0 = read_insert(v0, r1, [r2+0x8, scalar])

Replace one element in vector RD, starting at offset RT·OS, with scalar memory operand [RS+IM2]. (OS = operand size).

repeat_block

format	op-code	operands
2.2.7	8.1	vectors. Optional

float v0 = repeat_block(v1, r2, 8)

Repeat a block of data to make a longer vector. This is the same as broadcast, but with a larger block of data. v1 is an input vector containing a data block to repeat. A constant (IM2) is the length in bytes of the block to repeat. This must be a multiple of 4. r2 is the length in bytes of the result vector. This instruction is useful for matrix multiplication.

This instruction cannot have a mask.

repeat_within_blocks

format	op-code	operands
2.2.7	9.1	vectors. Optional

float v0 = repeat_within_blocks(v1, r2, 8)

This divides a vector into blocks and broadcasts the first element of each block to the rest of the block. The block size is given by a constant (IM2). This must be a multiple of the operand size, and at least 4 bytes. There may be a maximum limit to the block size. r2 is the length in bytes of the resulst vector. This instruction is useful for matrix multiplication.

For example, if the input vector contains (0,1,2,3,4,5,6,7,8) and the block size is 3 times the operand size, then the result will be (0,0,0,3,3,3,6,6,6).

This instruction cannot have a mask.

replace

format	op-code	operands
2.6	3	vectors and 32-bit immediate constant
3.1	32	vectors and 64-bit immediate constant. Optional

int32 v0 = replace(v1, 1), mask=v2, fallback=v3
double v0 = replace(v1, 2.3)

All elements of src1 are replaced by the integer or floating point constant src2.

When used without a mask, the constant is simply broadcast to make a vector of the same length as src1. This is useful for broadcasting a constant to all elements of a vector. Only the length of src1 (in bytes) is used, not its contents, when this instruction is used without a mask.

When used with a mask, the elements of src1 are selectively replaced. Elements that are not selected by the mask will be taken from a fallback register.

replace_even

format	op-code	operands
2.6	4	vectors and 32-bit immediate constant

Same as replace. Only even-numbered vector elements are replaced.

replace_odd

format	op-code	operands
2.6	5	vectors and 32-bit immediate constant

Same as replace. Only odd-numbered vector elements are replaced.

set_len

format	op-code	operands
1.2	2	vectors

v1 = set_len(v2, r3)

Sets the length of a vector register to the number of bytes specified by a general purpose register. If the specified length is more than the maximum length for the specified operand type then the maximum length will be used.

If the output vector is longer than the input vector then the extra elements will be zero. If the output vector is shorter than the input vector then the extra elements will be discarded.

This instruction cannot have a mask.

set_num

format	op-code	operands
1.2	3	vectors

v1 = set_num(v2, r3)

The length of a vector register is changed to the value of general purpose register. The length is indicated as number of elements. If the length is increased then the extra elements will be zero. If the length is decreased then the superfluous elements are lost.

This instruction differs from set_len by multiplying the length by the operand size. This instruction cannot have a mask.

shift_down

format	op-code	operands
1.2	19	vectors

int32 v0 = shift_down(v1, r2)

Shift elements of a vector down by the number of elements (n) indicated by general purpose register. The upper n elements of the result will be zero, the lower n elements are lost. The length of the vector is not changed.

This instruction differs from shift_reduce by indicating the shift count as a number of elements rather than a number of bytes, and by not changing the length of the vector.

This instruction cannot have a mask.

shift_expand

format	op-code	operands
1.2	16	vectors

int32 v0 = shift_expand(v1, r2)

The length of a vector is expanded by the specified number of bytes by adding zero-bytes at the low end and shifting all bytes up. If the resulting length is more than the maximum vector length for the specified operand type then the upper bytes are lost.

This instruction cannot have a mask.

shift_reduce

format	op-code	operands
1.2	17	vectors

int32 v0 = shift_reduce(v1, r2)

The length of a vector is reduced by the specified number of bytes by removing bytes at the low end and shifting all bytes down. If the resulting length is less than zero then the result will be a zero-length vector. The specified operand type is ignored.

This instruction cannot have a mask.

shift_up

format	op-code	operands
1.2	18	vectors

int32 v0 = shift_up(v1, r2)

Shift elements of a vector up by the number of elements (n) indicated by general purpose register. The lower n elements of RD will be zero, the upper n elements are lost. The length of the vector is not changed.

This instruction differs from shift_expand by indicating the shift count as a number of elements rather than a number of bytes, and by not changing the length of the vector.

This instruction cannot have a mask.

sign_extend

format	op-code	operands
multi	4	general purpose and integer scalar

int8 r0 = sign_extend(r1) // result is 64 bits
int8 v0 = sign_extend(v1) // lower 8 bits of each 64-bit element is extended to 64 bits
int8 v0 = sign_extend([r1, scalar]) // memory operand is 8 bits, result is 64 bits scalar

Sign-extend smaller integer to 64 bits.

The input can be an 8-bit, 16-bit or 32-bit integer. This integer is sign-extended to produce a 64-bit output in a general purpose register or a scalar in a vector register. If the input is a vector then only the first element in each 64-bit block of the input vector is used. Floating point types cannot be used.

sign_extend_add

format	op-code	operands
multi	5	general purpose registers

int8 r0 = sign_extend_add(r1, r2)
int32 r0 = sign_extend_add(r1, [r2]), options = 2

src2 is an integer of the specified size, often a memory operand. This integer is sign-extended to produce a 64-bit integer. The sign-extended value is optionally shifted left by a value of 1 .. 3, specified in the options. The result is added to the 64-bit integer in src1 and the result is stored in the 64-bit destination register.

This instruction is useful for converting relative pointers to absolute pointers, where the reference point is in src1. The relative pointer may be scaled by a factor of 1, 2, 4, or 8, corresponding to a shift count or 0, 1, 2, or 3, respectively. Support for larger scale factors is optional.

This instruction does not sign-extend when the operand size is 64 bits, but it can still add and shift 64-bit integers.

This instruction will not generate traps in case of signed or unsigned overflow.

vec2gp

format	op-code	operands
1.3 B	1	vector register in, g.p. register out

int64 r0 = vec2gp(v1)

Copy value of first element of vector register RS to general purpose register RD. Integers are sign-extended. Single precision floating point values are zero-extended.

Data read and write instructions

address

format	op-code	operands
2.9 A	32	general purpose register

int64 r1 = address([mydata])

Gives the address of a data object in static memory.

The value must be shifted two places to the right if used as the target for a jump or call instruction, because code addresses are based on 32-bit words rather than bytes.

clear

format	op-code	operands
1.3 B	58	vector. Optional

clear(v5) // clear one vector register
clear(v5, 8) // clear vector registers v5 - v8

Clear one or more vector registers by setting the length to zero. A cleared register is regarded as unused.

It may be advantageous to clear vector registers after use. This will mean that there is less data to save during a task switch.

extract_store

format	op-code	operands
2.5 A	40	vector. Optional

int32 [r3+8, scalar] = extract_store(v1, r2)

Extract one element from vector RD, starting at offset RT·OS, with size OS into memory operand [RS+IM2].
(OS = operand size).

fence

format	op-code	operands
2.5 B	16	memory operand and immediate. Optional

int32 fence([r1], 2)

Memory fence at address [RS+IM2].

Options indicated by IM1:

IM1 value	meaning
1	read fence
2	write fence
3	read and write fence

move

The move instruction, described at page 55 can read a register from a memory operand.

pop

format	op-code	operands
1.8 B 1.3 B	57 57	general purpose registers. Optional vector registers. Optional

```
pop(r5)          // pop 64-bit register r5 off the stack
pop(r1, r2, 6)   // pop registers r2-r6 from stack pointed to by r1
pop(v5)          // pop vector register v5 off the stack
pop(v5, 9)       // pop vector registers v5-v9 off the stack
```

The pop instruction can pop one or more registers from a stack. The registers are popped in reverse order.

An optional first register (RD) indicates a stack pointer. The default stack pointer (SP) is used if not specified. An optional last operand is an index of the last register to pop. The syntax for the POP instruction has no equal sign. The operand size is 64 bits by default. A different operand type is allowed only for general purpose registers.

The stack is growing backwards by default. The last register is read from the address pointed to by the stack pointer. Then the stack pointer is incremented by the amount that was occupied by the register. This is 8 bytes by default for a general purpose register or a variable amount for a vector register. This process is repeated if multiple registers are popped. Registers are pushed in forward order and popped in reverse order.

It is possible to make a forward-growing stack for general purpose registers by adding 0x80 to the last operand. A stack containing vector registers cannot grow forwards because the pop instruction needs to read the vector length stored at the beginning of each field before it can read the rest of the vector.

See the push instruction on page 62 for more details.

prefetch

format	op-code	operands
multi	3	memory operand. Optional

Prefetch memory operand into cache for later read or write. Different variants (not yet defined) can be specified by option bits in IM3 for formats with E template.

push

format	op-code	operands
1.8 B	56	general purpose register. Optional
1.3 B	56	vector register. Optional

```
push(r5)            // push 64-bit register r5 on the stack
push(r1, r2, 6)     // push registers r2-r6 on stack pointed to by r1
push(r1, r2, 0x86) // push registers r2-r6 on forward growing stack r1
push(v5, 9)         // push vector registers v5-v9 on the stack
```

The push instruction can push one or more registers on a stack.

An optional first register (RD) indicates a stack pointer. The default stack pointer (SP) is used if not specified. An optional last operand is an index of the last register to push. The syntax for the PUSH instruction has no equal sign. The operand size is 64 bits by default. A different operand type is allowed only for general purpose registers.

The stack is growing backwards by default. The stack pointer is decremented by the amount that will be occupied by the register. The first register is then stored to the address pointed to by the stack pointer. This size is 8 bytes for a full general purpose register or a variable amount for a vector register. This process is repeated if multiple registers are pushed.

It is possible to make a forward-growing stack for general purpose registers by adding 0x80 to the last operand. This may be used as an increment-pointer-and-store instruction. A stack containing vector registers cannot grow forwards because a later pop instruction needs to read the vector length stored at the beginning of each field before it can read the rest of the vector.

Note that vector registers are stored in an implementation-dependent way by the push instruction. The microprocessor may compress the data or it may insert extra space for optimal alignment of memory access. The programmer should make no assumption about how the vector elements are stored. A pushed vector register can only be restored by a pop instruction on the same or an identical microprocessor that pushed it. If the memory image is moved before restoring, it must be moved by a multiple of the maximum vector lenth. The maximum amount of memory occupied by a pushed vector register is 8 bytes plus the maximum vector length.

See also the pop instruction on page 61 for more details.

store

format	op-code	operands
multi	1	memory operand and g.p. or vector register
2.5 B	8	memory operand and 32-bit constant. Optional

int32 [r0+r1*4] = r1
float [r0, length = r1] = v2
float [r0 + 0x10] = 2.5

Write the value of a register or constant to a memory operand.

The size of the memory operand is determined by the operand size OS when a scalar memory operand is specified, or by the vector length register in RS when a vector operand is specified.

An immediate constant cannot be bigger than 32 bits. A 64 bit integer constant can only be used if it fits into a 32-bit signed integer. A float64 constant can only be used if it can be represented as single precision without loss of precision.

The hardware must be able to handle memory operand sizes that are not powers of 2 without touching additional memory (read and rewrite beyond the memory operand is not allowed unless access from other threads is blocked during the operation and any access violation is suppressed). It is allowed for the hardware to write the operand in a piecemeal fashion.

Masked operation with a mask of zero will leave the corresponding memory element untouched. An explicit fallback value cannot be specified.

General arithmetic instructions

abs

format	op-code	operands
1.8 B	0	g.p. registers
1.3 B	16	vector registers

int32 r0 = abs(r1, 1)

Absolute value of signed number.

Signed integers can overflow when the input is the minimum value. The handling of overflow for signed integers is controlled by the constant IM1 as follows:

IM1	result when input is INT_MIN
0	INT_MIN (wrap around)
1	INT_MAX (saturation)
2	zero

add

format	op-code	operands
multi	8	all standard types
multi	44	float16. Optional
1.1 C	6	32-bit register and 16-bit sign-extended constant
1.1 C	10	32-bit register and 8-bit sign-extended constant shifted left by another constant.
1.1 C	11	64-bit register and 8-bit sign-extended constant shifted left by another constant.
1.1 C	18	32-bit register and 16-bit zero-extended constant shifted left by 16
2.9	2	g.p. register and 32-bit zero-extended constant
2.9	4	g.p. register and 32-bit constant shifted left by 32
1.4 C	1	vector of 16-bit integer elements and broadcast 16 bit integer constant. Optional
1.4 C	10	vector of 32-bit integer elements and broadcast 8-bit sign-extended constant shifted left by another constant. Optional
1.4 C	11	vector of 64-bit integer elements and broadcast 8-bit sign-extended constant shifted left by another constant. Optional
1.4 C	34	single precision floating point vector and broadcast half precision floating point constant. Optional
1.4 C	35	double precision floating point vector and broadcast half precision floating point constant. Optional
1.4 C	40	half precision floating point vector and broadcast half precision floating point constant. Optional

int32 r0 = r1 + r2
int32 r0 = r1 + 2
int32+ r0 += 4
int32+ r0++
float v0 = v1 + [r2 + 8, length = r5]

Addition.

If you want to add a 64-bit constant to a general purpose register, and triple size instructions are not supported, then add the lower half first using the zero-extended version, and then add the upper half using the shifted version.

add_add

format	op-code	operands
multi	51	all types. Optional

This gives two additions in one instruction:

dest = ± src1 ± src2 ± src3

For optimal precision with floating point operands, the intermediate sum of the two numerically largest operands should preferably be calculated first with extended precision.

The signs of the operands can be inverted as indicated by the following option bits:

Table 5.3: Control bits for add_add

Option bits	Meaning
bit 0	change sign of src1
bit 1	change sign of src2
bit 2	change sign of src3

There is no sign change if there are no option bits.

This instruction may be supported for integer operands or floating point or both.

compare

format	op-code	operands
multi	7	all types

Examples:
int8 r0 = r1 $>$ r2
uint8 r0 = r1 $>$ r2
float v0 = v1 $<=$ 2.3
int32 r0 = compare(r1, 2), mask=r3, fallback=r4, options=0b1001

The compare instruction compares two source operands and generates a boolean scalar or vector where bit 0 indicates the result. This instruction can do different compare operations depending on option bits 0-4 defined according to the following table:

Table 5.4: Condition codes for compare instruction

Bit 3-2-1-0	Meaning for integer	Meaning for floating point
_ 0 0 0	a $=$ b	a $=$ b
_ 0 0 1	a $\neq$ b	a $\neq$ b
_ 0 1 0	a $<$ b	a $<$ b
_ 0 1 1	a $\geq$ b	a $\geq$ b
_ 1 0 0	a $>$ b	a $>$ b
_ 1 0 1	a $\leq$ b	a $\leq$ b
_ 1 1 0		abs(a) $<$ abs(b)
_ 1 1 1		abs(a) $\geq$ abs(b)
0 _ _ _	compare as signed	unordered gives 0
1 _ _ _	compare as unsigned	unordered gives 1

Option bit 3 indicates how to threat floating point NAN inputs. A compare operation is considered unordered if at least one floating point input operand is NAN. The translation of high level language operators to ordered and unordered compare operations are listed on page 91.

The result is indicated in bit 0 of the destination register. It is 1 for true and 0 for false. The remaining bits are copied from a mask register, or zero if there is no mask register. The number of mask bits available is implementation dependent.

The condition code is zero (indicating compare for equal) if there are no option bits.

A fallback register can be used as operand for an extra boolean operation, with or without a mask. Only bit 0 of the fallback register is used. This option is controlled by option bits 4-5:

Table 5.5: Alternative use of fallback register

bit 5 bit 4	Output with mask	Output without mask
0 0	mask ? result : fallback	result
0 1	mask && result && fallback	result && fallback
1 0	mask && (result || fallback)	result || fallback
1 1	mask && (result ^ fallback)	result ^ fallback

div

format	op-code	operands
multi	14	all types. Optional for integer vectors

int32 r0 = r1 / r2
int32 r0 = div(r1, r2), options = 4
float v0 = v1 / [r2 + 8, length = r5]

Signed division.

This instruction has multiple rounding modes. The rounding mode for integer operands is controlled by option bits (IM3) as follows:

Table 5.6: division instructions

Option bits 0-3	Meaning
0 0 0 0	Truncate towards zero (default)
0 1 0 0 0 1 0 1 0 1 1 0 0 1 1 1	Nearest or even Down Up Truncate towards zero
other values	Not allowed

Truncation is always used with integer operands when there are no option bits.

The rounding mode for floating point operands is controlled by the mask or numeric control register. Option bits must be zero for floating point operands.

Division of floating point operands by zero gives ±INF (or NAN if exceptions are enabled).
Division of integer operands by zero gives INT_MAX or INT_MIN.
Overflow occurs by division of INT_MIN by -1. The result will wrap around to give INT_MIN.

div_rev

format	op-code	operands
multi	16	all types. Optional for integer vectors

int32 r0 = 10 / r2
int32 v0 = div_rev(v1, v2), options = 4

Same as div, with the two source operands swapped.
The rounding mode is controlled in the same way as for the div instruction.

div_u

format	op-code	operands
multi	15	all integer types. Optional for integer vectors

uint32 r0 = r1 / r2
uint32 v0 = div_u(v1, v2), options=4

Unsigned integer division.
The rounding mode is controlled in the same way as for the div instruction, see page 66

Division by zero gives UINT_MAX.

div_ex

format	op-code	operands
1.2 A	24	Integer vectors. Optional for more than one element

Divide vector of double-size signed integers RS by signed integers RT. RS has element size 2·OS. These are divided by the even numbered elements of RT with size OS. The truncated results are stored in the even-numbered elements of RD. The remainders are stored in the odd-numbered elements of RD. (OS = operand size).

div_ex_u

format	op-code	operands
1.2 A	25	Integer vectors. Optional for more than one element

Divide vector of double-size unsigned integers RS by unsigned integers RT. RS has element size 2·OS. These are divided by the even numbered elements of RT with size OS. The truncated results are stored in the even-numbered elements of RD. The remainders are stored in the odd-numbered elements of RD. (OS = operand size).

max

format	op-code	operands
multi	22	all types

int32 r0 = max(r1, r2)
float v0 = max(v1, v2)

Get the maximum of two numbers:
max(src1,src2) = src1 > src2 ? src1 : src2

Integer operands are treated as signed.

The handling of floating point NAN operands follows the definition of the maximum function in the 2019 revision of the IEEE floating point standard 754, which guarantees the propagation of NANs, unlike the 1985 and 2008 versions of the standard.

max_abs

format	op-code	operands
multi	23	all floating point types

float v0 = max_abs(v1, v2)

Gives the maximum of the absolute values of two floating point numbers.

max_abs(src1, src2) = max(abs(src1), abs(src2))

NAN values are treated in the same way as for the max instruction.

max_u

format	op-code	operands
multi	23	all integer types

uint32 r0 = max_u(r1, r2)

Gives the maximum of two unsigned integers.

max_u(src1,src2) = src1 > src2 ? src1 : src2

min

format	op-code	operands
multi	20	all types

int32 r0 = min(r1, r2)
float v0 = min(v1, v2)

Get the minimum of two numbers:
min(src1,src2) = src1 < src2 ? src1 : src2

Integer operands are treated as signed.

The handling of floating point NAN operands follows the definition of the minimum function in the 2019 revision of the IEEE floating point standard 754, which guarantees the propagation of NANs, unlike the 1985 and 2008 versions of the standard.

min_abs

format	op-code	operands
multi	21	all floating point types

float v0 = min_abs(v1, v2)

Gives the minimum of the absolute values of two floating point numbers.

min_abs(src1, src2) = min(abs(src1), abs(src2))

NAN values are treated in the same way as for the min instruction.

min_u

format	op-code	operands
multi	21	all integer types

uint32 r0 = min_u(r1, r2)

Gives the minimum of two unsigned integers.

min_u(src1,src2) = src1 < src2 ? src1 : src2

mul

format	op-code	operands
multi	11	all standard types
multi	46	float16. Optional
1.1 C	8	general purpose register and 16-bit sign-extended integer constant
1.4 C	36	single precision floating point vector and broadcast half-precision floating point constant. Optional
1.4 C	37	double precision floating point vector and broadcast half-precision floating point constant. Optional
1.4 C	41	half precision floating point vector and broadcast half-precision floating point constant. Optional

int32 r0 = r1 * r2
float v0 *= 5.0

Multiplication.

The same instruction can be used for signed and unsigned integers.

mul_add, mul_add2

format	op-code	operands
multi	49	mul_add: dest = $\pm$ src1 $\cdot$ src2 $\pm$ src3. All types. Optional
multi	50	mul_add2: dest = $\pm$ src1 $\cdot$ src3 $\pm$ src2. All types. Optional
multi	48	mul_add. float16. Optional

Fused multiply and add.

The fused multiply-and-add instruction can often improve the performance of floating point code significantly. The intermediate product is calculated with extended precision according to the IEEE 754-2008 standard.

The signs of the operands can be inverted as indicated by the following option bits

Table 5.7: Control bits for mul_add and mul_add2

Option bits	Meaning
bit 0	change sign of product in even-numbered vector elements
bit 1	change sign of product in odd-numbered vector elements

bit 2	change sign of addend in even-numbered vector elements
bit 3	change sign of addend in odd-numbered vector elements

These option bits make it possible to do multiply-and-add, multiply-and-subtract, multiply-and-reverse-subtract, etc. It can also do multiply with alternating add and subtract, which is useful in calculations with complex numbers. There is no sign change if there are no option bits.

Support for integer operands is optional. Support for floating point operands is optional but desired.

mul_ex

format	op-code	operands
1.2 A	26	integer vectors

int32 v0 = mul_ex(v1, v2)

Extended multiply, signed.

Multiply even-numbered signed integer vector elements to double size result. The result extends into the next odd-numbered vector element.

mul_ex_u

format	op-code	operands
1.2 A	27	integer vectors

uint32 v0 = mul_ex_u(v1, v2)

Extended multiply, unsigned.

Multiply even-numbered unsigned integer vector elements to double size result. The result extends into the next odd-numbered vector element.

mul_hi

format	op-code	operands
multi	12	integer vectors

int32 r0 = mul_hi(r1, r2)
int32 v0 = mul_hi(v1, 2)

High part of signed integer product.

dest = (src1 $\cdot$ src2) $>>$ OS
(Signed, OS = operand size in bits).

mul_hi_u

format	op-code	operands
multi	13	integer vectors

uint32 r0 = mul_hi_u(r1, r2)

High part of unsigned integer product.

dest = (src1 · src2) $>>$ OS
(Unsigned, OS = operand size in bits).

mul_2pow

format	op-code	operands
multi	32	all floating point types

Multiply by power of 2.
dest = src1 * 2^{src2}
src1 and dest are floating point vectors, while src2 is interpreted as a signed integer vector with the same element size as src1 and dest.

Overflow will produce infinity. The result will be zero rather than a subnormal number in case of underflow, regardless of control bits in the mask or numeric control register. The reason for this is that speed has priority here. This instruction will typically take a single clock cycle, while floating point multiplication by a power of 2 takes multiple clock cycles. This is useful for fast multiplication or division by a power of 2.

This instruction has the same op1 code as shift_left, but applies to floating point types only.

rem

format	op-code	operands
multi	18	all types. Optional for vectors of more than one element

int32 r0 = r1 % r2
float v0 = rem(v1, v2)

Modulo.

Signed with integer operands or floating point operands.

A floating point number modulo zero gives NAN. An integer modulo zero gives zero.

rem_u

format	op-code	operands
multi	19	integers. Optional for vectors of more than one element

uint32 r0 = r1 % r2

Unsigned modulo or remainder.

An integer modulo zero gives zero.

round

format	op-code	operands
1.3 B	14	floating point vectors

float v0 = round(v1, 0)

Round floating point number to integer in floating point representation.

The rounding mode is specified in bit 0-1 of IM1. See table 3.3 page 28.

roundp2

format	op-code	operands
1.8 B	3	g.p. registers

int64 r0 = roundp2(r1, 1)

Round unsigned integer up or down to the nearest power of 2.

Options:

IM1 bits	meaning
bit 0	0: Round down to power or 2: dest = 1 « bitscan_reverse(src1). 1: Round up to power or 2: dest = ((src1 & (src1-1)) == 0) ? src1 : 1 « (bitscan_reverse(src1) + 1)
bit 4	0: returns 0 if the input is 0. 1: returns -1 if the input is 0.
bit 5	0: returns 0 if the result overflows. 1: returns -1 if the result overflows.

round2n

format	op-code	operands
1.3 B	15	vector registers. Optional

float v0 = round2n(v1, -4)

Round to nearest multiple of 2^n.
dest = $2^n \cdot$ round($2^{-n} \cdot$ src1)
n is a signed integer constant in IM1.

sqrt

format	op-code	operands
1.2 A	28	floating point vectors. Optional

Square root.

sub

format	op-code	operands
multi	9	all standard types
multi	45	float16. Optional
2.9	3	g.p. register and 32-bit zero-extended constant

int32 r0 = r1 - r2
int32 r0 = r1 - 2
int32+ r0 -= 4
int32+ r0–

float v0 = v1 - [r2 + 8, length = r5]

Subtraction.

sub_rev

format	op-code	operands
multi	10	all types

int32 r0 = 1 - r2
int32 v0 = - v2 + v1
float v0 = -v1 + [r2 + 8, length = r5]

Reverse subtraction.

dest = src2 - src1.

Arithmetic instructions with carry, overflow check, or saturation

These instructions do not generate traps on overflow because they provide alternative ways of handling overflow.

abs

see page 63.

add_c

format	op-code	operands
1.2 A	42	integer vectors with two elements. Optional

Addition with carry.

The vector has two elements. The upper element of src1 is used as carry in. The upper element of dest is used as carry out. Only the lower element of src2 is used.

Longer vectors are not supported. See page 176 for an alternative for longer vectors.

add_oc

format	op-code	operands
1.2 A	38	vector registers. Optional

Integer addition with overflow check.

Instructions with overflow check use the even-numbered vector elements for arithmetic instructions. Each following odd-numbered vector element is used for overflow detection.

Overflow conditions are indicated with the following bits:

bit 0. Unsigned integer overflow (carry or borrow).
bit 1. Signed integer overflow.

The values are propagated so that the overflow result of the operation is OR'ed with the corresponding values of both input operands.

add_ss

format	op-code	operands
1.2 A	32	integer vectors. Optional

Add signed integers with saturation.
Overflow and underflow produces INT_MAX and INT_MIN.

add_us

format	op-code	operands
1.2 A	33	integer vectors. Optional

Add unsigned integers with saturation.
Overflow produces UINT_MAX.

compress_ss

format	op-code	operands
1.2 A	5	integer vectors. Optional

Compress, signed with saturation.

Same as compress (see page 49). Integers are treated as signed and compressed with saturation. Floating point operands cannot be used. Masks cannot be used and overflow traps cannot be enabled for this instruction.

compress_us

format	op-code	operands
1.2 A	6	integer vectors. Optional

Compress, unsigned with saturation.

Same as compress (see page 49). Integers are treated as unsigned and compressed with saturation. Floating point operands cannot be used. Masks cannot be used and overflow traps cannot be enabled for this instruction.

div_oc

format	op-code	operands
1.2 A	41	vector registers. Optional

Divide signed integers with overflow check.
See add_oc for options.

mul_oc

format	op-code	operands
1.2 A	40	vector registers. Optional

Multiply integers with overflow check.
See add_oc for options.

mul_ss

format	op-code	operands
1.2 A	36	integer vectors. Optional

Multiply signed integers with saturation.
Overflow and underflow produces INT_MAX and INT_MIN.

mul_us

format	op-code	operands
1.2 A	37	integer vectors. Optional

Multiply unsigned integers with saturation.
Overflow produces UINT_MAX.

sub_b

format	op-code	operands
1.2 A	43	integer vectors with two elements. Optional

Subtraction with borrow.

The vector has two elements. The upper element of src1 is used as borrow in. The upper element of dest is used as borrow out. Only the lower element of src2 is used.

Longer vectors are not supported. See page 176 for an alternative for longer vectors.

sub_oc

format	op-code	operands
1.2 A	39	vector registers. Optional

Subtract integers with overflow check.
See add_oc for options.

sub_ss

format	op-code	operands
1.2 A	34	integer vectors. Optional

Subtract signed integers with saturation.
Overflow and underflow produces INT_MAX and INT_MIN.

sub_us

format	op-code	operands
1.2 A	35	integer vectors. Optional

Subtract unsigned integers with saturation.
Overflow and underflow produces UINT_MAX and 0.

Logic and bit manipulation instructions

and

format	op-code	operands
multi	26	all types
1.1 C	12	32-bit register and 8-bit signed constant shifted left by another constant
1.1 C	13	64-bit register and 8-bit signed constant shifted left by another constant
2.9	5	g.p. register and 32-bit constant shifted left by 32
1.4 C	2	vector of 16-bit integers, and broadcast 16-bit constant. Optional
1.4 C	12	vector of 32-bit integers, and broadcast sign-extended 8-bit constant shifted left by another constant. Optional
1.4 C	13	vector of 64-bit integers, and broadcast sign-extended 8-bit constant shifted left by another constant. Optional

int32 r0 = r1 & r2
int32 v0 = v1 & 2

Bitwise boolean and.

Floating point operands are treated as integers.
Do not use a floating point type with a constant operand unless you want the operand to be interpreted as floating point.

or

format	op-code	operands
multi	27	all types
1.1 C	14	32-bit register and 8-bit signed constant shifted left by another constant
1.1 C	15	64-bit register and 8-bit signed constant shifted left by another constant
2.9	6	g.p. register and 32-bit constant shifted left by 32
1.4 C	3	vector of 16-bit integers, and broadcast 16-bit constant. Optional
1.4 C	14	vector of 32-bit integers, and broadcast sign-extended 8-bit constant shifted left by another constant. Optional
1.4 C	15	vector of 64-bit integers, and broadcast sign-extended 8-bit constant shifted left by another constant. Optional

int32 r0 = r1 | r2
int32 v0 = v1 | 2

Bitwise boolean or.

Floating point operands are treated as integers.

Do not use a floating point type with a constant operand unless you want the operand to be interpreted as floating point.

xor

format	op-code	operands
multi	28	all types
1.1 C	16	32-bit register and 8-bit signed constant shifted left by another constant
1.1 C	17	64-bit register and 8-bit signed constant shifted left by another constant
2.9	7	g.p. register and 32-bit constant shifted left by 32
1.4 C	4	vector of 16-bit integers, and broadcast 16-bit constant. Optional
1.4 C	16	vector of 32-bit integers, and broadcast sign-extended 8-bit constant shifted left by another constant. Optional
1.4 C	17	vector of 64-bit integers, and broadcast sign-extended 8-bit constant shifted left by another constant. Optional

int32 r0 = r1 ^ r2
int32 v0 = v1 ^ 2

Bitwise boolean exclusive or.

Floating point operands are treated as integers.
Do not use a floating point type with a constant operand unless you want the operand to be interpreted as floating point.

bit_reverse byte_reverse

format	op-code	operands
1.3 B	20	vectors

int32 v0 = byte_reverse(v1, 0)
int32 v0 = bit_reverse(v1, 1)

IM1 = 0: Reverse the order of bytes within each vector element. This is useful for converting big-endian file data.
IM1 = 1: Reverse the order of bits in each element of a vector.

bits2bool

format	op-code	operands
1.2 A	12	integer vectors

int32 v0 = bits2bool(r1, v2)

Expand contiguous bits in a vector register to a boolean vector with each bit of the source going into bit 0 of each element of the destination. The remaining bits of each element are copied from the first element of the mask or the numeric control register. The number of mask or NUMCONTR bits available is implementation dependent.

The length in bytes of the result vector is specified by a general purpose register in RS.

This instruction cannot have a fallback register.

bitscan

format	op-code	operands
1.8 B	2	general purpose registers
1.3 B	21	integer vectors. Optional

int32 r0 = bitscan(r1, 0)
int64 v0 = bitscan(v1, 1)

Bit scan forward or reverse. Option bits are given in the second operand:

IM1 bits	meaning
bit 0	0: forward scan. Find index to the lowest set bit. 1: reverse scan. Find index to the highest set bit.
bit 4	0: returns 0 if the input is 0. 1: returns -1 if the input is 0.

bool_reduce

format	op-code	operands
1.3 B	26	integer vectors

int32 v0 = bool_reduce(v1)

A boolean vector is reduced by combining bit 0 of all elements.
The output is a scalar integer where bit 0 is the AND combination of all the bits, and bit 1 is the OR combination of all the bits. The remaining bits are reserved for future use.

This instruction cannot have a mask.

bool2bits

format	op-code	operands
1.3 B	25	integer vectors

int64 v0 = bool2bits(v1)

A boolean vector with n elements is packed into the lower n bits of RD, taking bit 0 of each element. The length of RD will be at least sufficient to contain n bits.

This instruction cannot have a mask.

category_reduce

format	op-code	operands
1.3 B	26	floating point vectors

float v0 = category_reduce(v1)

A floating point vector is analyzed and each element is classified as belonging to one of the eight categories listed below. Each bit in the output indicates that at least one element in RT belongs to the corresponding category.

Bit number	Category
0	at least one element is NAN
1	at least one element is zero
2	at least one element is negative subnormal
3	at least one element is positive subnormal
4	at least one element is negative normal
5	at least one element is positive normal
6	at least one element is negative infinity
7	at least one element is positive infinity

This instruction cannot have a mask.

clear_bit

format	op-code	operands
multi	36	all types

Clear bit number src2 in src1.

dest = src1 & ~(1 << src2).

Floating point operands are treated as integers.

set_bit

format	op-code	operands
multi	37	all integer types

Set bit number src2 in src1 to one.

dest = src1 | (1 << src2)

toggle_bit

format	op-code	operands
multi	38	all types

Change the value of bit number src2 in src1 to its opposite.

dest = src1 ^ (1 << src2)

compare

See page 65

fp_category

format	op-code	operands
1.3 B	17	floating point vectors

float v0 = fp_category(v1, 1)

The input is a floating point vector. The output is a boolean vector where bit 0 of each element indicates if the input RS belongs to any of the categories indicated by the bits in the immediate

operand IM1. The remaining bits of the output are taken from the numeric control register. The number of NUMCONTR bits available is implementation dependent. Any floating point value will belong to one, and only one, of these categories.

Table 5.8: Meaning of bits in fp_category

Bit number	Meaning
0	± NAN
1	± Zero
2	− Subnormal
3	+ Subnormal
4	− Normal
5	+ Normal
6	− Infinite
7	+ Infinite

make_mask

format	opcode	operands
2.6	2	integer vectors

int32 v0 = make_mask(v1, 2), mask=v3

Make a mask from the bits of the 32-bit integer constant src2. Each bit of the constant goes into bit 0 of one element of the output. The remaining bits of each element are taken from a mask register, or from NUMCONTR if there is no mask. The number of mask or NUMCONTR bits available is implementation dependent. The length of the output is the same as the length of src1. If there are more than 32 elements in the vector then the bit pattern of src2 is repeated.

make_sequence

format	opcode	operands
1.3 B	4	all vectors

int32 v0 = make_sequence(r1, 2)

Makes a vector of sequential numbers. The number of elements is indicated by a general purpose register. The first element is equal to the immediate operand IM1, the next element is IM1+1, etc. IM1 must be an integer in the range -128 →127.

mask_length

format	opcode	operands
2.2.7	1.1	integer vectors

int64 v0 = mask_length(v1, r2, 0), options=2

Make a boolean vector to mask the first n bytes of a vector, where n is the value of a general purposer register r2.
The result vector will have the same length as the input vector v1. r2 indicates the length of the part that is enabled by the mask.

The following option bits can be specified:
bit 0 = 0: bit 0 will be 1 in the first n bytes in the output and 0 in the rest.
bit 0 = 1: bit 0 will be 0 in the first n bytes in the output and 1 in the rest.
bit 1 = 1: copy remaining bits from input vector v1 into each vector element.
bit 2 = 1: copy remaining bits from the numeric control register.
bit 4 = 1: broadcast remaining bits from a constant (IM2) into all 32-bit words of the result.
Bit 1-7 of IM2 go to bit 1-7 of the result.
Bit 8-11 of IM2 go to bit 20-23 of the result.
Bit 12-15 of IM2 go to bit 26-29 of the result.
Output bits that are not set by any of these options will be zero. If multiple options are specified, the results will be OR'ed.

This instruction can have a mask but not a fallback register. The fallback value is zero.

move_bits

format	op-code	operands
2.0.7	0.1	general purpose registers. Optional
2.2.7	0.1	integer vectors. Optional

int16 r0 = move_bits(r1, r2, 3, 4, 5)
int32 v0 = move_bits(v1, v2, 3, 4, 5)

Extract, insert, or move bit fields.

Takes one or more contiguous bits from position src4 in the second source operand (src2) and insert them into position src3 in the first source operand (src1). The remaining bits of src1 are unchanged.
The third source operand (src3) is the bit position in src2 to take bits from.
The fourth source operand (src4) is the bit position to insert the bits in.
The fifth source operand (src5) is the number of bits to move.
The first two source operands must be registers, the remaining operands must be constants.

Definition:
m = (1 << src5) - 1
b = src2 >> src3
dest = (src1 & ~(m<<src4)) | (b & m) << src4

Examples:
int16 r1 = 0x1234
int16 r2 = 0xABCD
// extract 4 bits from r2, starting from position 8, and insert into position 0 of r1:
int16 r0 = move_bits(r1, r2, 8, 0, 4) // = 0x123B
// insert 8 bits from position 0 of r2 into position 4 of r1:
int16 r0 = move_bits(r1, r2, 0, 4, 8) // = 0x1CD4
// move 4 bits from position 8 in r2 into the same position of r1:
int16 r0 = move_bits(r1, r2, 8, 8, 4) // = 0x1B34

popcount

format	op-code	operands
1.8 B	4	general purpose registers. Optional
1.3 B	22	integer vectors. Optional

int32 r0 = popcount(r1)

int32 v0 = popcount(v1)

The popcount instruction counts the number of 1-bits in an integer. It can also be used for parity generation.

rotate

format	op-code	operands
multi	33	all integer types

dest = rotate(src1, src2)

Rotate the bits of src1 left if src2 is positive, or right if src2 is negative.

shift_left

format	op-code	operands
multi	32	all integer types

Shift integer left.
dest = src1 $<<$ src2

The result is zero if src2 is outside the range 0 $\leq$ src2 $<$ number_of_bits.

This instruction has the same op1 code as mul_2pow, but applies to integer operand types only.

shift_right_s

format	op-code	operands
multi	34	all integer types

Shift integer right with sign extension (arithmetic shift).

int32 dest = src1 $>>$ src2

The result is 0 or -1 if src2 is outside the range 0 $\leq$ src2 $<$ number_of_bits.

shift_right_u

format	op-code	operands
multi	35	all integer types

Shift integer right with zero extension (logical shift).

uint32 dest = src1 $>>$ src2

The result is zero if src2 is outside the range 0 $\leq$ src2 $<$ number_of_bits.

funnel_shift

format	op-code	operands
multi	53	all integer types

int64 r1 = funnel_shift(r2, r3, r4)
int64 v1 = funnel_shift(v2, v3, r4)

This instruction concatenates two bit fields and shifts this to the right. This is useful for dealing with unaligned bit fields or unaligned vectors.

dest = src1 $>>$ src3 | src2 $<<$ (operand_size - src3)

For general purpose registers: Operand 1 (low) and operand 2 (high), with n bits each, are concatenated into a bit field with 2n bits. This bit field is shifted right by the number of bits indicated by the third operand. The lower n bits of the result are returned. The result is zero if src3 is outside the range $0 \leq$ src3 $<$ n.

For vector registers: This instruction is shifting whole vectors rather than vector fields when the operands are vector registers. The shift count is counting vector elements rather than bits. Vector operand 1 (low) with n elements and vector operand 2 (high), with n elements or less, are concatenated into a larger vector with at most 2n elements. This concatenated vector is shifted down by the number of elements indicated by the third operand. The lower n elements of the result are returned. The result is zero if src3 is outside the range $0 \leq$ src3 $<$ n.

Some implementations may work slowly for high shift counts.

This instruction will rotate a vector if both input vectors are the same.

A funnel shift in the opposite direction can be made by swapping the first two operands and subtracting the shift count from the operand size.

select_bits

format	op-code	operands
multi	52	all integer types

int32 r0 = select_bits(r1, r2, r3)

dest = src1 & src3 | src2 & ~src3

This instruction combines bits from the first two source operands, using the third source operand as selector.

test_bit

format	op-code	operands
multi	39	all integer types

Test the value of bit number src2 in src1, and make it the least significant bit of the output, to use as a boolean. The result is zero if src2 is out of range.

result = (src1 $>>$ src2) & 1.

The result is indicated in bit 0 of the destination register. The remaining bits of the output may be taken from a mask register or numeric control register. The number of mask or NUMCONTR bits available is implementation dependent.

A fallback register can be used as an operand for an extra boolean operation, with or without a mask. Only bit 0 of the fallback register is used. The boolean operation is controlled by option bits 0-1. Option bit 2 inverts the result, bit 3 inverts the fallback, and bit 4 inverts the mask. These options are summarized in the following table, giving the value of bit 0 of the destination register.

Table 5.9: Alternative use of mask and fallback register controlled by option bits

bit 4	bit 3	bit 2	bit 1	bit 0	Output
0	0	0	0	0	mask ? result : fallback
0	0	1	0	0	mask ? !result : fallback
0	1	0	0	0	mask ? result : !fallback
0	1	1	0	0	mask ? !result : !fallback
1	0	0	0	0	!mask ? result : fallback
1	0	1	0	0	!mask ? !result : fallback
1	1	0	0	0	!mask ? result : !fallback
1	1	1	0	0	!mask ? !result : !fallback
0	0	0	0	1	mask & result & fallback
0	0	1	0	1	mask & !result & fallback
0	1	0	0	1	mask & result & !fallback
0	1	1	0	1	mask & !result & !fallback
1	0	0	0	1	!mask & result & fallback
1	0	1	0	1	!mask & !result & fallback
1	1	0	0	1	!mask & result & !fallback
1	1	1	0	1	!mask & !result & !fallback
0	0	0	1	0	mask & (result | fallback)
0	0	1	1	0	mask & (!result | fallback)
0	1	0	1	0	mask & (result | !fallback)
0	1	1	1	0	mask & (!result | !fallback)
1	0	0	1	0	!mask & (result | fallback)
1	0	1	1	0	!mask & (!result | fallback)
1	1	0	1	0	!mask & (result | !fallback)
1	1	1	1	0	!mask & (!result | !fallback)
0	0	0	1	1	mask & (result ^ fallback)
0	0	1	1	1	mask & (!result ^ fallback)
0	1	0	1	1	mask & (result ^ !fallback)
0	1	1	1	1	mask & (!result ^ !fallback)
1	0	0	1	1	!mask & (result ^ fallback)
1	0	1	1	1	!mask & (!result ^ fallback)
1	1	0	1	1	!mask & (result ^ !fallback)
1	1	1	1	1	!mask & (!result ^ !fallback)

The value of mask is 1 if there is no mask register. The remaining bits are copied from the mask register if option bit 5 is set, or from the numeric control register if there is no mask and bit 5 is set. The remaining bits are zero if option bit 5 is not set. The number of mask or NUMCONTR bits available is implementation dependent.

test_bits_and

format	op-code	operands
multi	40	all integer types

Test if the indicated bits are all 1.
result = ((src1 & src2) == src2)

The result is indicated in bit 0 of the destination register. The remaining bits of the output may be taken from a mask register or numeric control register.

A fallback register can be used as an operand for an extra boolean operation, with or without a

mask. Only bit 0 of the fallback register is used. These options are controlled by option bits 0-4 in the same way as for test_bit, as indicated in table 5.9.

The remaining bits are copied from the mask register if option bit 5 is set, or from the numeric control register if there is no mask and bit 5 is set. The remaining bits are zero if option bit 5 is not set. The number of mask or NUMCONTR bits available is implementation dependent.

test_bits_or

format	op-code	operands
multi	41	all integer types

Test if at least one of the indicated bits is 1.
result = ((src1 & src2) != 0)

The result is indicated in bit 0 of the destination register. The remaining bits of the output may be taken from a mask register or numeric control register.

A fallback register can be used as an operand for an extra boolean operation, with or without a mask. Only bit 0 of the fallback register is used. These options are controlled by option bits 0-4 in the same way as for test_bit, as indicated in table 5.9.

The remaining bits are copied from the mask register if option bit 5 is set, or from the numeric control register if there is no mask and bit 5 is set. The remaining bits are zero if option bit 5 is not set. The number of mask or NUMCONTR bits available is implementation dependent.

truth_tab3

format	op-code	operands
2.0.6	8.1	general purpose registers. optional
2.2.6	8.1	integer vectors. optional

int32 r0 = truth_tab3(r1, r2, r3, 0xF2), options=0
int32 v0 = truth_tab3(v1, v2, v3, 0xF2), options=0

This instruction can make an arbitrary bitwise boolean function of three integer variables, expressed by an 8-bit truth table in an immediate constant. Each bit of the result is the arbitrary boolean function of the corresponding bits of the three input registers. The boolean function is calculated for each bit position separately. Three bits from the three input registers are combined into a 3-bit index, where the bit from the first input register goes into the least significant bit and the bit from the last input register goes into the most significant bit. This index is then selecting one bit from the truth table to go into the result.

For example, the boolean function F = A & $\sim$ B | C has the truth table 0b11110010 or 0xF2.

This can be used as a universal instruction for bitwise logic functions of up to three inputs. Functions of two inputs can be obtained by using the same register for two of the three input registers.

This instruction can also be used for manipulating masks where only bit 0 contains the boolean result. The remaining bits are controlled by options according to the table below. This is useful when the result is used as a mask for floating point instructions:

Table 5.10: Options for truth_tab3

Options	Meaning
0	all bits contain boolean results

1	bit 0 contains a boolean result. The remaining bits are zero
2	bit 0 contains a boolean result. The remaining bits are taken from a mask or numeric control register. The number of mask or NUMCONTR bits available is implementation dependent.

Combined arithmetic/logic and branch instructions with integer operands

These instructions are doing an arithmetic or logic operation and a conditional jump depending on the result. Each instruction can be coded in a number of different formats described on page 29.

The instructions are listed below in pairs, where the second instruction has the branch condition inverted.

These instructions cannot have a mask. The destination operand, if any, should preferably be the same as the first source operand for optimal performance. The second source operand may be a register, a memory operand, or an immediate constant with no more than 32 bits.

add/jump_zero

format	op-code	instruction	operands
all	16	add/jump_zero	integer
all	17	add/jump_nzero	integer

Add two integer operands and jump if the result is zero.

add/jump_neg

format	op-code	instruction	operands
all	18	add/jump_neg	integer
all	19	add/jump_nneg	integer

Add two integer operands and jump if the signed result is negative.
The result will wrap around in the case of overflow and jump if the result has the sign bit set.

add/jump_pos

format	op-code	instruction	operands
all	20	add/jump_pos	integer
all	21	add/jump_npos	integer

Add two integer operands and jump if the signed result is positive.
The result will wrap around in the case of overflow and jump if the result is not zero and does not have the sign bit set.

add/jump_overflow

format	op-code	instruction	operands
all	22	add/jump_overflow	integer
all	23	add/jump_noverflow	integer

Add two signed integer operands and jump if the result overflows.

add/jump_carry

format	op-code	instruction	operands
all	24	add/jump_carry	integer
all	25	add/jump_ncarry	integer

Add two unsigned integer operands and jump if the operation produces a carry.

increment_compare/jump_above/below

format	op-code	instruction	operands
all	48	increment_compare/jump_below	integer
all	49	increment_compare/jump_aboveeq	integer
all	50	increment_compare/jump_above	integer
all	51	increment_compare/jump_beloweq	integer

Add 1 to the first source operand and jump if the signed result is less than a certain limit. The result is saved in the destination operand. This is useful for implementing a simple “for” loop.

The result will wrap around from INT_MAX to INT_MIN in case of overflow.

sub/jump_zero

format	op-code	instruction	operands
Not 1.7	0	sub/jump_zero	integer
Not 1.7	1	sub/jump_nzero	integer

Subtract two integer operands and jump if the result is zero.

Immedate constants are not supported. The assembler will automatically convert a sub/jump_zero instruction to an add/jump_zero instruction with the negative constant.

sub/jump_neg

format	op-code	instruction	operands
Not 1.7	2	sub/jump_neg	integer
Not 1.7	3	sub/jump_nneg	integer

Subtract two integer operands and jump if the signed result is negative.
The result will wrap around in the case of overflow and jump if the result has the sign bit set.

Immedate constants are not supported. The assembler will automatically convert a sub/jump_neg instruction to an add/jump_neg instruction with the negative constant.

sub/jump_pos

format	op-code	instruction	operands
Not 1.7	4	sub/jump_pos	integer
Not 1.7	5	sub/jump_npos	integer

Subtract two integer operands and jump if the signed result is positive.
The result will wrap around in the case of overflow and jump if the result is not zero and does not have the sign bit set.

Immedate constants are not supported. The assembler will automatically convert a sub/jump_pos instruction to an add/jump_pos instruction with the negative constant.

sub/jump_overflow

format	op-code	instruction	operands
Not 1.7	6	sub/jump_overflow	integer
Not 1.7	7	sub/jump_noverflow	integer

Subtract two signed integer operands and jump if the result overflows.

Immedate constants are not supported. The assembler will automatically convert a sub/jump_overflow instruction to an add/jump_overflow instruction with the negative constant.

sub/jump_borrow

format	op-code	instruction	operands
Not 1.7	8	sub/jump_borrow	integer
Not 1.7	9	sub/jump_nborrow	integer

Subtract two unsigned integer operands and jump if the operation produces a borrow.

Immedate constants are not supported. The assembler will automatically convert a sub/jump_borrow instruction to an add/jump_borrow instruction with the negative constant.

sub_maxlen/jump_pos

format	op-code	instruction	operands
1.7C, 2.5.1B, 2.5.4C	52	sub_maxlen/jump_pos	integer
1.7C, 2.5.1B, 2.5.4C	53	sub_maxlen/jump_npos	integer

Subtract the maximum vector length (in bytes) from a general purpose register and jump if the result is positive. The immediate operand indicates the operand type for which the maximum vector length is obtained. The operand size for the source and destination register is 64 bits in C formats.

This instruction makes it easy to implement the type of vector loop described on on page 13.

and/jump_zero

format	op-code	instruction	operands
Not 1.7	10	and/jump_zero	all
Not 1.7	11	and/jump_nzero	all

Bitwise and. Jump if zero.

dest = src1 & src2
jump if dest == 0

All operands are treated as integers. Floating point operands are treated as unsigned integer scalars in vector registers.

or/jump_zero

format	op-code	instruction	operands
Not 1.7	12	or/jump_zero	all
Not 1.7	13	or/jump_nzero	all

Bitwise or. Jump if zero.

dest = src1 | src2
jump if dest == 0

All operands are treated as integers. Floating point operands are treated as unsigned integer scalars in vector registers.

xor/jump_zero

format	op-code	instruction	operands
Not 1.7	14	xor/jump_zero	all
Not 1.7	15	xor/jump_nzero	all

Bitwise exclusive or. Jump if zero.

dest = src1 ^ src2
jump if dest == 0

All operands are treated as integers. Floating point operands are treated as unsigned integer scalars in vector registers.

test_bit/jump_true

format	op-code	instruction	operands
all	26	test_bit/jump_true	all
all	27	test_bit/jump_false	all

int test_bit(r1, 3), jump_true target
if (int r1 & 8) {jump target}

Test a single bit in the first source operand as indicated by the an index in the second source operand and jump if the indicated bit is 1. There is no destination operand.

jump if ((src1 >> src2) & 1) == 1

All operands are treated as unsigned integers. Floating point operands are treated as integer scalars in vector registers.

test_bits_and/jump_true

format	op-code	instruction	operands
all	28	test_bits_and/jump_true	all
all	29	test_bits_and/jump_false	all

int test_bits_and(r1, 7), jump_true target
if (int (r1 & 7) == 7) {jump target}

Test the AND combination of the bits indicated by the second source operand. Jump if the indicated bits are all 1. There is no destination operand.

jump if (src1 & src2) == src2

All operands are treated as unsigned integers. Floating point operands are treated as integer scalars in vector registers.

test_bits_or/jump_true

format	op-code	instruction	operands
all	30	test_bits_or/jump_true	all
all	31	test_bits_or/jump_false	all

int test_bits_or(r1, 7), jump_true target
if (int r1 & 7) {jump target}

Test the OR combination of the bits indicated by the second source operand. Jump if at least one of the indicated bits is 1. There is no destination operand.

jump if (src1 & src2) != 0

All operands are treated as unsigned integers. Floating point operands are treated as integer scalars in vector registers.

integer compare and branch instructions

int64 compare(r1, r2), jump_equal target

Compare instructions have no destination operand. Overflow cannot occur.

op-code	instruction	jump condition
32	compare/jump_equal	r1 = r2
33	compare/jump_nequal	r1 $\neq$ r2
34	compare/jump_sbelow	r1 $<$ r2, signed
35	compare/jump_saboveeq	r1 $\geq$ r2, signed
36	compare/jump_sabove	r1 $>$ r2, signed
37	compare/jump_sbeloweq	r1 $\leq$ r2, signed
38	compare/jump_ubelow	r1 $<$ r2, unsigned
39	compare/jump_uaboveeq	r1 $\geq$ r2, unsigned
40	compare/jump_uabove	r1 $>$ r2, unsigned
41	compare/jump_ubeloweq	r1 $\leq$ r2, unsigned

floating point branch instructions

The conditional jump instructions use general purpose registers for integer operands with at most 64 bits, and vector registers when a floating point type is specified. Only the first element of a floating point vector is used.

Addition and subtraction instructions with conditional branching do not support floating point operands.

floating point compare and branch instructions

double compare(v1, v2), jump_above target

Compare instructions have no destination operand. Overflow cannot occur.
0.0 and -0.0 are treated as equal.

The unordered versions of floating point compare instructions are true when any input operand is NAN. The versions without _uo suffix are false when any operand is NAN. The unordered versions are needed because conditions are often inversed in the compilation process. For example the inverse of compare/jump_below is not compare/jump_aboveeq but compare/jump_aboveeq_uo. This is a consequence of the rule that all comparisons except '!=' return false when the inputs are unordered, i.e. when at least one operand is NAN, according to the IEEE-754 standard for floating point arithmetic.

op-code	instruction	jump condition	high level language
32	compare/jump_equal	v1 = v2	a == b
0	compare/jump_equal_uo	v1 = v2	
33	compare/jump_nequal	v1 $\neq$ v2	
1	compare/jump_nequal_uo	v1 $\neq$ v2	a != b
34	compare/jump_below	v1 $<$ v2	a < b
2	compare/jump_below_uo	v1 $<$ v2	!(a >= b)
35	compare/jump_aboveeq	v1 $\geq$ v2	a >= b
3	compare/jump_aboveeq_uo	v1 $\geq$ v2	!(a < b)
36	compare/jump_above	v1 $>$ v2	a > b
4	compare/jump_above_uo	v1 $>$ v2	!(a <= b)
37	compare/jump_beloweq	v1 $\leq$ v2	a <= b
5	compare/jump_beloweq_uo	v1 $\leq$ v2	!(a > b)
38	compare/jump_abs_below	abs(v1) $<$ abs(v2)	
6	compare/jump_abs_below_uo	abs(v1) $<$ abs(v2)	
39	compare/jump_abs_aboveeq	abs(v1) $\geq$ abs(v2)	
7	compare/jump_abs_aboveeq_uo	abs(v1) $\geq$ abs(v2)	
40	compare/jump_abs_above	abs(v1) $>$ abs(v2)	
8	compare/jump_abs_above_uo	abs(v1) $>$ abs(v2)	
41	compare/jump_abs_beloweq	abs(v1) $\leq$ abs(v2)	
9	compare/jump_abs_beloweq_uo	abs(v1) $\leq$ abs(v2)	
24	fp_category/jump_true	value belongs to one of the indicated categories	
25	fp_category/jump_false	value does not belong to any of the indicated categories	

The _abs conditions ignore the sign bits and compare the absolute values of the two operands.

The fp_category/jump_true instruction tests if the value of the first operand belongs to any of the categories indicated by the second source operand, which is an integer. The categories are indicated according to table 5.8 on page 80

Unconditional and indirect jump, call, and return instructions

Control transfer instructions are available in a number of different formats, described on page 29.

Direct jump

format	op-code	operands
1.7 D	0	jump with 24 bit relative address
2.5.4 C	58	jump with 32 bit relative address
3.1.1 B	58	jump with 64 bit absolute address (optional)

Unconditional jump.

Direct function call

format	op-code	operands
1.7 D	8	call with 24 bit relative address
2.5.4 C	59	call with 32 bit relative address
3.1.1 B	59	call with 64 bit absolute address (optional)

Function call.

The return address is stored on the call stack. The calling conventions are described in chapter 12.4.

Indirect jump

format	op-code	operands
1.6 B	58	64 bit absolute address in memory operand with 8 bit offset
1.7 C	60	64 bit absolute address in register
1.6 A	60	Multi-way jump with table of relative addresses (see below)
2.5.2 B	58	Absolute address in memory operand with 32 bit offset

Indirect call

format	op-code	operands
1.6 B	59	64 bit absolute address in memory operand with 8 bit offset
1.7 C	61	64 bit absolute address in register
1.6 A	61	Multi-way call with table of relative addresses (see below)
2.5.2 B	59	Absolute address in memory operand with 32 bit offset

Relative and multi-way jump and call

format	op-code	operands
1.6 A	60	Jump with table of relative addresses. Has reference point, base and scaled index
2.5.2 B	60	Jump with relative address. Has reference point, base and offset
1.6 A	61	Call with table of relative addresses. Has reference point, base and scaled index
2.5.2 B	61	Call with relative address. Has reference point, base and offset

The multi-way and relative jump and call instructions, jump_relative and call_relative, are using pointers stored in memory relative to an arbitrary reference point. These instructions are intended to facilitate multi-way branches (switch/case statements), function tables in code interpreters, virtual function tables in object oriented languages with polymorphism, and general use of relative pointers. The relative pointers stored in memory use 8, 16, or 32 bits, depending on the distance to the reference point, while absolute pointers need 64 bits. This saves memory space and cache space.

Relative pointers to jump or call addresses are stored in memory as signed offsets relative to an arbitrary reference point. The reference point may be the table address, the ip_base, or any reference point defined by the programmer. The operand type specifies the size of the table entries.

This instruction works as follows. Calculate the address of a table entry as the base pointer plus the offset (unscaled) or the index (RT) scaled by the operand size. Read a relative pointer from this address, sign-extend to 64 bits, and scale by 4. Then add the reference point (RD). Jump or call to the calculated address. The array index (RT) is scaled by the operand size, while the table entries are scaled by the instruction word size (4). The reference point must be aligned by 4.

This instruction in format 1.6A has base pointer in RS, scaled index in RT, and reference point in RD. Format 2.5.2B has base pointer in RS, unscaled index in IM2 and reference point in RD.

A table of pointers used by the table-based jump_relative and call_relative instructions is preferably placed in the constant data section (CONST). This makes it possible to use the table base as reference point. This also improves security by giving read-only access to the table.

These instructions cannot have a mask and will not generate overflow traps in case of overflow in the address calculation, but you will get access violation traps when attempting to access an illegal memory address.

return

format	op-code	operands
1.6 B	62	

Return from function call. The return address is taken from the call stack.

Return instructions do not need a stack offset when the calling conventions specified in chapter 12.4 are used.

breakpoint

format	op-code	operands
1.7 C	63	

This instruction is used as a debug breakpoint.

It is the same as trap(1). The complete instruction code word is 0x7FE00001.

filler

format	op-code	operands
1.7 C	63	

This instruction is used for filling unused code memory. It will generate a trap (interrupt) if executed.

All fields are filled with ones. The complete instruction code word is 0x7FFFFFFF.

System call, system return, and traps

See page 100.

Miscellaneous instructions

address

format	op-code	operands
2.9 B	32	g.p. registers

int64 r1 = address [memory_label]

Calculate an address relative to a pointer by adding a 32-bit sign-extended constant to a special pointer register. The pointer register can be THREADP (28), DATAP (29), IP (30) or SP(31).

compare_swap

format	op-code	operands
2.5 A	18	g. p. registers and memory operand with 32 bit offset. Optional

int32 r1 = compare_swap(r1, r2, [r3+0x100])

Atomic compare and swap instruction, used for thread synchronization and for lock-free data sharing between threads. src1 and src2 are register operands, src3 is a memory operand, which must be aligned to a natural address. All operands are treated as integers, regardless of the specified operand type. The operation is:

```
temp = src3;
if (temp == src1) src3 = src2;
return temp;
```

This instruction cannot have a mask.

Further atomic instructions can be implemented if needed, preferably with the same format and consecutive values of OP1.

nop

format	op-code	operands
multi	0	
3.0	0	

No operation. Used as a filler to replace removed code or to align code entries.

Unused bits may be used for debugging information, etc.

The processor is allowed to skip NOPs as fast as it can at an early stage in the pipeline. These NOPs cannot be used as timing delays, only as fillers.

undef

format	op-code	operands
multi	63	

Undefined code. Guaranteed to generate trap (interrupt) in all future implementations

userdef

format	op-code	operands
multi	56-62	any types

Reserved for user-defined instructions.

System instructions

These instructions cannot have a mask.

input

format	op-code	operands
1.8 B	62	general purpose registers
1.2 A	62	vector registers

int32 r0 = input(r1, 4)
int64 v0 = input(r1, r2)

Read from input port into register RD. Privileged instruction.

General purpose register input with immediate port address:
The immediate operand contains a port address in the interval 0 - 254. Register RS is ignored.

General purpose register input with port address in register:
The immediate operand is 255. Register RS contains a 64 bit port address.

Vector register input with port address in register:
RS = port address. RT = vector length in bytes,
Vector input is not necessarily supported for all input ports.
Masks are not necessarily supported.

output

format	op-code	operands
1.8 B	63	general purpose registers
1.2 A	63	vector registers

int32 output(r1, r2, 4)
int64 output(v0, r1, r2)

Write register value RD to output port. Privileged instruction.

General purpose register output with immediate port address:
The immediate operand contains a port address in the interval 0 - 254. Register RS is ignored.

General purpose register output with port address in register:
The immediate operand is 255. Register RS contains a 64 bit port address.

Vector register output with port address in register:
RS = port address. RT = vector length in bytes,
Vector output is not necessarily supported for all output ports.
Masks are not necessarily supported.

read_capabilities, write_capabilities

format	op-code	operands
1.8 B	34	read_capabilities(capabilities register, constant)
1.8 B	35	write_capabilities(g.p. register, constant)

Preliminary specification.

Read or write processor capabilities register. These registers are used for indicating capabilities of the processor, such as support for optional instructions and limitations to vector lengths. These registers are initialized with their default values at program start.

The immediate constant in IM1 may determine details of the operation.

Table 5.11: List of capabilities registers

Capabilities register number	Meaning
capab0 capab1	Microprocessor model or brand ID Microprocessor version number
capab2	Disable error traps. Bit 0: unknown instructions, bit 1: wrong instruction operands, bit 2: array overflow, bit 3: memory read violation, bit 4: memory write violation, bit 5: misaligned memory access.
capab4 capab5	Code cache size, level 1 Data cache size, level 1
capab8 capab9	Support for operand sizes in general purpose registers. Bit 0: int8, bit 1: int16, bit 2: int32, bit 3: int64 Support for operand sizes in vector registers. Bit 0: int8, bit 1: int16, bit 2: int32, bit 3: int64, bit 4: int128, bit 5: float32, bit 6: float64, bit 7: float128, bit 8: float16.

capab12	Maximum vector length for general instructions.
capab13	Maximum vector length for permute instructions.
capab14	Maximum block size for permute instructions.
capab15	Maximum vector length for compress_sparse and expand_sparse.

Some capabilities registers can be modified for test purposes or to tell the software not to use a specific instruction.

Setting bits in capab2 will suppress error traps. Instead, the errors will be counted in performance counter registers described on page 98. To test if a particular instruction is supported, set bit 0 in capab2, reset the performance counter, try to execute the instruction, and read the performance counter again.

Changing the values of the maximum vector length has the following effects. If the maximum length is reduced below the physical capability then any attempt to make a longer vector will result in the reduced length. The behavior of vector registers that already had a longer length before the maximum length was reduced, is implementation dependent. If the maximum vector length is set to a higher value than the physical capability then any attempt to make a vector longer than the physical capability will cause a trap to facilitate emulation, if the platform supports emulation.

Capabilities registers 12-15 can be increased for the purpose of emulation. The value of capabilities registers 12-15 must be powers of 2.

read_memory_map, write_memory_map

format	op-code	operands
1.2 A	60	vector = read_memory_map(base, index)
1.2 A	61	write_memory_map(vector, base, index)

Preliminary specification.

int64 v0 = read_memory_map(r2, r3)

Read memory map and save it to a vector register. Privileged instruction.
RD = destination vector register, RT-RS = internal address.

int64 write_memory_map(v1, r2, r3)

Write a vector register to memory map. RD = vector register source. RT-RS = internal address. Privileged instruction.

read_call_stack, write_call_stack

format	op-code	operands
1.2 A	58	read_call_stack(r1, r2)
1.2 A	59	write_call_stack(v1, r2, r3)

Preliminary specification.

int64 v0 = read_call_stack(r1, r2)

Read the internal call stack into a vector register. This instruction is used for saving the internal call stack to system memory in case of overflow. Privileged instruction.

RD = destination vector register, RT-RS = internal address.

int64 write_call_stack(v1, r2, r3)

Write a vector register to the internal call stack. This instruction is used for restoring the internal call stack. Privileged instruction.

read_perf

format	op-code	operands
1.8 B	36	performance counter register, constant

int64 r0 = read_perf(perf1, 1)

A number of internal registers are used for counting performance related events. This instruction reads performance counter registers and performance related information. Some performance counters may be implementation-specific.

Table 5.12: List of performance counter registers

Performance counter	Second operand	Meaning
perf0	-1	Reset all performance counters
perf1	1	CPU clock cycles
perf1	0	Reset CPU clock cycles counter
perf2	1	Number of instructions executed
perf2	2	Number of double size instructions
perf2	3	Number of triple size instructions
perf2	4	General purpose register instructions
perf2	5	G. p. register instructions with mask zero
perf2	0	Reset counters
perf3	1	Vector instructions executed
perf3	0	Reset counter
perf4	1	Vector registers in use. Returns one bit for each vector register
perf5	1	Jumps, calls, and return instructions
perf5	2	Direct, unconditional jumps, calls, and returns
perf5	3	Indirect jumps and calls
perf5	4	Conditional jumps
perf5	0	Reset counters
perf16	1	Unknown instructions attempted
perf16	2	Wrong operands for instruction
perf16	3	Array overflow
perf16	4	Memory read violation
perf16	5	Memory write violation
perf16	6	Memory access misaligned
perf16	62	Code address where first error occurred
perf16	63	Type of first error
perf16	0	Reset error counters

The perf16 register is useful for detecting errors when error traps are disabled using the capabilities registers described on page 96.

read_perfs

format	op-code	operands
1.8 B	37	performance counter register, constant

This is the same as the read_perf instruction, but serializing. The pipeline is flushed before reading the counter so that no instruction can execute out of order with read_perfs.

read_spec, write_spec

format	op-code	operands
1.8 B	32	read_spec(special register, constant)
1.8 B	33	write_spec(g.p. register, constant)

int64 r0 = read_spec(spec1, 0)
int64 r1 = read_spec(datap)

Read a special system register. The following special registers are currently defined. The size is 64 bits. These registers are initialized with their default values at program start.

The immediate operand (IM1) is currently unused. This instruction cannot have a mask.

Table 5.13: List of special registers

Special register name	number	Meaning
numcontr	spec0	Numeric control register
threadp	spec1	Thread environment block pointer
datap	spec2	Data section pointer

read_spev

format	op-code	operands
1.2 A	56	special vector register, general purpose register

int64 v0 = read_spev(spec0, r2)

Read special vector register spev1 into vector register result with length r2 bytes.

The following special registers are currently defined:

Table 5.14: Special registers that can be read into vectors

Special register number	Meaning
spec0	Numeric control register (NUMCONTR). The value is broadcast into all elements of the destination register with the indicated operand size and length.
spec48	Name of processor. The output is a zero-terminated UTF-8 string containing the brandname and model name of the microprocessor.

read_sys, write_sys

format	op-code	operands
1.8 B	38	read_sys(system register, constant)
1.8 B	39	write_sys(g.p. register, constant)

Read or write system register. Details are not defined yet. These instructions are privileged.

sys_call

System calls use ID numbers rather than addresses to identify system functions. The ID is the combination of a module ID identifying a particular system module or device driver and a function ID identifying a particular function within this module. The module ID and the function ID are both 16 or 32 bits, so that the combined system call ID is up to 64 bits. The sys_call instruction has the following variants:

Table 5.15: Variants of system call instruction

Format	Operand type	Register operands	Module ID	Function ID
1.6 A	32 bit	3	RT bit 16-31	RT bit 0-15
1.6 A	64 bit	3	RT bit 32-63	RT bit 0-31
2.5.7 C	64 bit	0	IM3 bit 0-31	IM1,IM2 bit 0-15
3.1.2 B	64 bit	2	IM3 bit 0-31	IM2 bit 0-31

The sys_call instruction can indicate a block of memory to be shared with the system function. The address of the memory block is pointed to by the register specified in RD and the length is in register RS. This memory block, which the caller must have access rights to, is shared with the system function. The system function will get the same access rights to this block as the calling thread has, i. e. read access and/or write access. This is useful for fast transfer of data between the caller and the system function. No other memory is accessible to both the caller and the called function. If the RD and RS fields are both r0 then no memory block is shared. If RD and RS are both SP then all the application's data memory is shared. The sys_call instruction in format 2.5.7 has no register operands and no shared memory block. System calls cannot have a mask.

Parameters for system functions are transferred in registers, following the same calling conventions as normal functions. The registers used for function parameters are usually different from the registers in the RD, RS and RT fields. Function parameters that do not fit into registers must reside in the shared memory block.

sys_return

format	op-code	operands
1.7 C	62	

Return from system call.

trap

format	op-code	instruction	immediate operand
1.7 C	63	trap	0-254
1.7 C	63	filler	255

Traps work like interrupts. The unconditional trap has an 8-bit interrupt number in IM1. This is an index into the interrupt vector table, which initially starts at absolute address zero. The unconditional trap instruction may use IM2 for additional information.

A trap instruction with all 1's in all fields (opcode 0x7FFFFFFF) can be used as filler in unused parts of code memory.

conditional trap

format	op-code	instruction	immediate operand
2.5.5C	63	compare, trap_uabove	limit

Conditional traps are currently not supported.

The conditional trap generates a trap if the specified condition is true.
IM2 contains the interrupt number.
IM3 contains an immediate operand

Compare/trap_uabove will generate a trap if RD $>$ IM3. This is useful for checking if an array index exceeds the upper bound. The lower bound does not have to be checked because we use unsigned compare.

5.1 Common operations that have no dedicated instruction

This section discusses some common operations that are not implemented as single instructions, and how to code these operations in software.

Change sign

For integer operands, do a reverse subtract from zero. For floating point operands, use the toggle_bit instruction on the sign bit.

Not

To invert all bits in an integer, do an XOR with -1. To invert a Boolean, do an XOR with 1.

Rotate through carry

Rotates through carry are rarely used, and common implementations can be very inefficient.
A left rotate through carry can be replaced by an add_c with the same register in both source operands.

Horizontal vector add

See example 14.10.

5.2 Unused instructions

Unused instructions and opcodes can be divided into three types:

1. The opcode is reserved for future use. Attempts to execute it will trigger a trap (synchronous interrupt) which can be used for generating an error message or for emulating instructions that are not supported.
2. The opcode is guaranteed to generate a trap, not only in the present version, but also in all future versions. This can be used as a filler in unused parts of the memory or for indicating unrecoverable errors. It can also be used for emulating user-specific instructions.
3. The error is ignored and does not trigger a trap. It can be used for future extensions that improve performance or functionality, but which can be safely ignored when not supported.

All three types are implemented, where type 1 is the most common.

Nop instructions with nonzero values in unused fields are type 3. These instructions are ignored.

Prefetch and fence instructions with no memory operand, with nonzero values in unused fields, or with undefined values in IM3 are type 3. These instructions are ignored.

Unused bits in masks and numeric control register are type 3. These bits are ignored.

Trap instructions and conditional trap instructions with nonzero values in unused fields or undefined values in any field are type 2. These instructions are guaranteed to generate a trap. A special version of the trap instruction is intended as filler in unused or inaccessible parts of code memory.

The undef instruction is type 2. It is guaranteed to generate a trap in all systems. It can be used for testing purposes and emulation.

The userdef__ instructions are type 1. These instructions are reserved for user-defined and application-specific purposes.

Instructions with erroneous coding should preferably behave as type 1. This includes instruction codes with nonzero values in unused fields, operand types not supported, or any other bit pattern with no defined meaning in any field. Type 3 behavior may alternatively be allowed in these cases. If so, the instruction should behave as if it were coded correctly.

All other opcodes not explicitly defined are type 1. These may be used for future instructions.

Small systems with no operating system and no trap support should define alternative behavior.

Chapter 6

Other implementation details

6.1 Endianness

The memory organization is little endian. Instructions for byte swapping are provided for reading and writing big endian binary data files.

Rationale

The storage of vectors in memory would depend on the element size if the organization was big endian. Assume, for example, that we have a 128 bit vector register containing four 32-bit integers, named A, B, C, D. With little endian organization, they are stored in memory in the order:

A0, A1, A2, A3, B0, B1, B2, B3, C0, C1, C2, C3, D0, D1, D2, D3,

where A0 is the least significant byte of A and D3 is the most significant byte of D. With big endian organization we would have:

A3, A2, A1, A0, B3, B2, B1, B0, C3, C2, C1, C0, D3, D2, D1, D0.

This order would change if the same vector register is organized, for example, as eight integers of 16 bits each or two integers of 64 bits each. In other words, we would need different read and write instructions for different vector organizations.

Little endian organization is more common for a number of reasons that have been discussed many times elsewhere.

6.2 Pointers and addresses

ForwardCom is using a flat memory model with 64-bit addresses. Pointers stored in registers contain absolute addresses with 64 bits. A lower number of bits is allowed for systems with a smaller address space.

Function pointers and code pointers must be divisible by 4 because the code consists of 4-byte words.

Data pointers and function pointers stored in data memory may contain absolute or relative addresses. There is a problem if absolute pointers in static data memory need to be initialized prior to program start. There are three different ways to accomplish this:

1. Initialize the data to an absolute address. This is possible, but it is not recommended because the data section containing an absolute address will be position-dependent. A particular platform may or may not support position-dependent code.

2. Make a start-up code that initializes the pointer, using a relative address for calculation.

3. Store a relative pointer and calculate the full address in the running program. This is explained below

Relative pointers can be more compact than absolute pointers. A relative pointer is storing the address of a function, code label, or data label relative to an arbitrary reference point. The reference point must be placed in a section with the same address regime as the target that we want to address - either IP-based or DATAP-based or THREADP-based.

A relative pointer may be scaled by a power of 2. For example, it is useful to scale a relative code pointer by 4 because all code addresses are divisible by 4. This allows us to use fewer bits for the relative pointer. The relative pointer may be a signed integer of 8, 16, or 32 bits. The number of bits must be sufficient to contain the distance between the target and the reference point, divided by the scale factor, in a signed integer. The calculation of the scaled relative address is done by the linker.

A relative pointer must be converted to an absolute pointer before it can be used for addressing a target. The sign_extend_add instruction is useful for converting a relative pointer to an absolute pointer. This instruction will read the relative pointer, sign-extend it to 64 bits, optionally scale it by a factor of 2, 4, or 8, and then add the address of the reference point. See page 164 for details.

Relative code pointers can also be used directly in the jump_relative or call_relative instruction. This instruction will read one entry from an indexed list of relative addresses, sign extend the relative address, scale it by 4, add an arbitrary reference point, and then jump or call to the calculated address. This is a very compact and efficient way of implementing a multiway jump or call, such as a switch/case statement. See page 165 for an example. The jump_relative instruction can also be used without an index if you have just one relative code pointer. (page 165)

Direct jump and call instructions, and conditional jump instructions, use relative addresses of 8, 16, 24, or 32 bits stored as part of the instruction. The least significant two bits of relative code addresses are not stored because they are always zero. The target address of a relative jump or call is calculated in the following way. The relative address is multiplied by 4 and sign-extended to 64 bits. The address of the end of the instruction (beginning of next instruction) is added to this value to get the address of the target.

Indirect jump and call instructions can use a pointer containing an absolute address. The pointer can be a register or a memory operand initialized as described above.

Multiway jump and call instructions use a table of relative addresses stored in data memory as explained above. A read-only data section is preferred for security reasons.

6.3 Implementation of call stack

There are various methods for saving the return addresses for function calls: a link register, a separate call stack, or a unified stack for return addresses and local data. Here, we will discuss the pro's and con's of each of these methods.

Link register

Some systems use a link register to hold the return address. The advantage of a link register is that a leaf function can be called without storing anything on the stack. This saves cache bandwidth in programs with many leaf function calls. The disadvantage is that every non-leaf function needs to save the link register on a stack before calling another function, and restore the leaf register before returning.

If we decide to have a link register then it should be a special register, not one of the general purpose registers. A link register does not need to support all the things that a general purpose register can do. If the link register is included as one of the general purpose registers then it will be tempting for a programmer to save it to another register rather than on the stack, and then end the function by jumping to that other register. This will work, of course, but it will interfere with the way returns are predicted. A branch predictor is typically using a special mechanism for predicting returns, which is different from the mechanism used for predicting other jumps and branches. This mechanism, which is called a return stack buffer, is a small rolling cache that remembers the addresses of the last calls. If a function returns by a jump to another register than the link register then it will use the wrong prediction mechanism, and this will cause severe delays due to misprediction of the subsequent series of returns. The return stack buffer will also be messed up if the link register is used for indirect jumps or other purposes.

The only instructions that are needed for the link register other than call and return, are push and pop. We can reduce the number of instructions in non-leaf functions by making a combined instruction for “push link register and then call a function” which can be used for the first function call in a non-leaf function, and another instruction for “pop link register and then return” to end a non-leaf function. However, this will violate the principle that we want to avoid complex instructions in order to simplify the pipeline design.

The only performance gain we get from using a link register is that it saves cache bandwidth by not saving the return address on leaf function calls. It will not affect performance in applications where cache bandwidth is not a bottleneck. The performance of the return instruction is not influenced by cache bandwidth because it can rely on the prediction in the return stack buffer.

The disadvantage of using a link register is that the compiler has to treat leaf functions and non-leaf functions differently, and that non-leaf functions need extra instructions for saving and restoring the leaf register on the stack.

Therefore, we will not use a link register in the ForwardCom architecture.

Separate call stack

We may have two stacks: a call stack for return addresses and a data stack for the local data of each function. Most programs have a quite limited call depth so that the entire call stack, or at least the “hot” part of it, can be stored on the chip. This will improve the performance because no memory or cache operations are needed for call and return operations – at least not in the critical innermost loops of the program. It will also simplify prediction of return addresses because the on-chip rolling stack and the return stack buffer will be one and the same structure.

The call stack can be implemented as a rolling register stack on the chip. The call stack is spilled to memory if it overflows. A return instruction after such a spilling event will use the on-chip value rather than the value in memory as long as the on-chip value has not been overwritten by new calls. Therefore, the spilling event is unlikely to occur more than once in the innermost part of a program. Spilling of the call stack will be extremely rare, even for programs with recursive functions, if the on-chip call stack has a reasonable size.

The pointer for the call stack should not be a general purpose register because the programmer will rarely need to access it directly. Direct manipulation of the call stack is only needed in a stack unroll event (after an exception or long jump) or a task switch.

A function does not have easy access to the return address that it was called from. Information about the caller may be supplied explicitly as a function parameter in the rare case that it is needed. There is a security advantage in hiding the return address inside the chip. This prevents overwriting return addresses in case of program errors or malicious buffer overflow attacks.

A further advantage of a separate call stack is that call and return instructions can be executed at a very early stage in the pipeline because there is no dependence on other instructions that may

modify the stack pointer. This avoids delays or bubbles in the pipeline that may otherwise occur when a return address is only available at a late stage in the pipeline.

The disadvantage of having a separate call stack is that it makes memory management more complicated if there are two stacks that can potentially overflow. The size of the call stack can be predicted accurately for programs without recursive functions by using the method described on page 141.

Unified stack for return addresses and local data

Many current systems use the same stack for return addresses and local data. Corruption of the stack is an error that often occurs with a unified stack, resulting in a wrong return address.

Conclusion for ForwardCom

It is decided to use a separate call stack for ForwardCom because of the security advantage and the more efficient hardware implementation.

The calling conventions defined in chapter 12.4 will work with a separate stack as well as a with unified stack.

6.4 Floating point errors and exceptions

The traditional ways of detecting floating point errors is to check a global status register or to use traps (software interrupts). Both methods are problematic and inefficient for SIMD processing and for out-of-order processing. A global status register does not tell where the error occurred or how many errors occurred. It is a problem for out-of-order processing that all floating point operations can write to the floating point status register. ForwardCom has no status register for this reason. In fact, the ForwardCom hardware design may not support instructions with multiple output registers.

Traps are avoided too because traps have to occur in order. This means that all instructions would have to execute speculatively until all preceding instructions that might possibly generate traps have retired.

See www.agner.org/optimize/nan_propagation.pdf for a detailed discussion of these problems.

The ForwardCom design can overcome these problems by signaling numerical exceptions in the same register that outputs the normal result of an instruction. This behavior is controlled by bits 2-5 in the mask register or the numerical control register. Floating point exceptions will generate a NAN with a payload that indicates the kind of error and its position if these bits are enabled. The NAN code will appear in the specific vector element that caused the exception, and this value will propagate to the end result if the sequence of instructions allows it. This method is sure to give deterministic results regardless of out-of-order processing and parallelism. The propagation of NAN values is further explained on page 107.

The exception indications are as follows:

Table 6.1: Floating point results with and without exceptions enabled

Error type	Option bit	Result
inexact		rounded result
inexact	bit 5	NAN(1)
underflow		$\pm$ 0
underflow	bit 4	NAN(2)

division by zero		$\pm$ INF
division by zero	bit 2	NAN(3)
overflow		$\pm$ INF
division overflow	bit 3	NAN(4)
multiplication overflow	bit 3	NAN(5)
addition overflow	bit 3	NAN(6)
conversion overflow	bit 3	NAN(7)
other overflow	bit 3	NAN(8)
$\infty - \infty$		NAN(0x20)
$0/0$		NAN(0x21)
∞/∞		NAN(0x22)
$0 * \infty$		NAN(0x23)
invalid remainder		NAN(0x24)
sqrt of negative		NAN(0x25)
invalid pow		NAN(0x28)
log of negative		NAN(0x29)
other math functions		NAN(0x2A - 0xFF)

Bits 0-7 of the NAN payload contains the exception code according to the table above. The quiet bit of the NAN is set. The remaining bits of the NAN payload contain the code address where the error occurred, with all bits inverted (lower 14 bits of the address for single precision, lower 43 bits for double precision, 64 bits for quadruple precision). This makes sure that the first error in a linear code sequence will have preference when multiple NAN codes are propagated to the end result.

Floating point exceptions can be detected by enabling the exception types you want to detect and checking for NAN values at the end of the floating point instruction sequence, at the end of any try/catch block, and before any instruction that cannot propagate NANs.

While ForwardCom has no status flags, the NAN payload is replacing the status flags for floating point exceptions specified by the IEEE-754 standard. Any arithmetic operation can signal at most one exception per vector element, according to the standard.

If you want to find the first floating point error during debugging then you may insert additional NAN checks before any backward jump or call and before any overflow of the address bits.

Traditional fault trapping is not possible in the preferred implementation of ForwardCom. Signaling NANs and signaling operations are not supported in the preferred implementation. However, bits 26-30 of the mask register or numerical control register may be used for enabling floating point fault trapping and trapping of signaling NANs if hardware support for traditional fault trapping is needed.

6.5 Propagation of NANs

If the result of a floating point calculation cannot be represented as a real number, then it will be coded as $\pm$ infinity or NAN (Not a number). Overflow and division by zero gives infinity. Operations that generate NAN include the following:
0/0, ∞/∞, 0*∞, ∞-∞, rem(1,0), rem(∞,1), sqrt(-1), log(-1), pow(-1, 0.1), asin(2).

These values are propagated through a series of calculations because any floating point calculation with a NAN input will produce a NAN output. This makes it possible to detect the error in the final result after a series of calculations. This method of detecting floating point errors is particularly useful for vector instructions because the result is independent of the vector length.

A NAN can contain a bit pattern of diagnostic information called the payload, and this bit pattern is propagated to the end result. A problem arises when two different NANs are combined, for ex-

ample NAN1 + NAN2. The IEEE-754 floating point standard has no definite rule for the result of the combination of two NANs, but it recommends that one of the two NAN operands is propagated to the result. Most microprocessors just propagate the first operand in this case. This is unfortunate because the result is not predictable when the compiler may code a+b as b+a. ForwardCom solves this problem by propagating the NAN value with the highest payload. Therefore, a+b and b+a will always give the same result.

Another problem is that NAN values are not propagated through the max and min instructions according to the 2008 version of the IEEE-754 standard. This problem is solved in the 2019 revision of the IEEE standard. ForwardCom is using the 2019 standard so that max and min instructions always produce a NAN output when at least one of the inputs is NAN.

Certain instructions have floating point inputs but no floating point output. This includes float2int, compare, and compare-and-jump instructions. Some systems are able to trap NANs in these cases, but ForwardCom is not fond of traps, as explained above on page 106. The float2int instruction can detect NANs by generating error codes in the output vector. Compare instructions can explicitly detect NAN inputs.

Infinity will also propagate to the end result in many cases. Infinity will be converted to NAN in cases like ∞/∞, 0*∞, and ∞-∞. Infinity will be converted to zero in situations like $1/\infty$. Infinity will not be propagated in situations like min(1,∞).

It is possible to generate NANs instead of infinity by enabling exception handling, as explained above. This will improve propagation of error information.

Signaling NANs are not trapped in ForwardCom, and there is no difference in the behavior of signaling NANs and quiet NANs. Signaling NANs are propagated without conversion to quiet NANs. Signaling NANs may be used for NAN boxing of non-numeric data or for application-specific error tracking.

Other methods for generating error messages in function libraries are discussed on page 127.

6.6 Detecting integer overflow

There is no common standard method for detecting overflow in integer calculations. The detection of overflow in signed integer operations is a real nightmare in some programming languages like C++ (see stackoverflow.com/questions/3944505/detecting-signed-overflow-in-c-c).

Possible solutions with a status register or fault trapping are not recommended in ForwardCom for the same reasons as explained above for floating point errors. Instead, the following methods may be used for detecting signed and unsigned integer overflow:

- Use conditional jump instructions.
  Signed: add/jump_overfl, sub/jump_overfl.
  Unsigned: add/jump_carry, sub/jump_borrow.
  Division: compare with zero and jump if equal before dividing.
  This method cannot detect multiplication overflow.
- Use instructions with overflow check in a vector register. The even-numbered vector elements are used for calculations, while the odd-numbered vector elements are used for propagating error flags. See page 73 for details.
- Use floating point instructions with integer data. Enable the mask bits for detection of overflow and inexact.

6.7 Performance monitoring and error tracking

ForwardCom supports a set of performance counter registers. These registers count the number of instructions of different kinds that are executed, for example double-size instructions, vector instructions, or branch instructions. The read_perf instruction can read and reset these registers. This instruction is described on page 99. Specific hardware implementations may have different performance counter registers. These will be described in the manual for the specific hardware implementation or soft core.

Tracking of floating point errors is described on page 106. Tracking of integer overflow is described on page 108.

A special set of performance counters may be used for tracking non-recoverable errors such as unknown instructions, wrong parameters for instructions, and memory access violations. These errors will normally generate a trap or stop execution, but the error traps can be disabled using the capabilities registers described on page 96. This makes it possible to execute a piece of code tentatively and afterwards check if any errors have occurred. The number of errors of each type is counted, and the position and type of the first error is recorded.

Specific hardware implementations may have additional features for debugging and error tracking. These will be described in the manual for the specific hardware implementation or soft core.

6.8 Multithreading

The ForwardCom design makes it possible to implement very large vector registers to process large data sets. However, there are practical limits to how much you can speed up the performance by using larger vectors. First, the actual data structures and algorithms often limit the vector length that can be used. And second, large vectors mean longer physical distances on the semiconductor chip and longer transport delays.

Additional parallelism can be obtained by running multiple threads in each their CPU core. The design should allow multiple CPU chips or multiple CPU cores on the same physical chip.

Communication and synchronization between threads can be a performance problem. The system should have efficient means for these purposes, perhaps including speculative synchronization.

It is probably not worthwhile to allow multiple threads to share the same CPU core and level-1 cache simultaneously (this is what Intel calls hyper-threading) because this could allow a low priority thread to steal resources from a high priority thread, and it is difficult for the operating system to determine which threads might be competing for the same execution resources if they are run in the same CPU core. Simultaneous multithreading may also involve security problems by making the design vulnerable to side-channel attacks.

6.9 Security features

Security is included in the fundamental design of both hardware and software. This includes the following features.

- A flexible and efficient memory protection mechanism.
- Separation of call stack and data stack so that return addresses cannot be compromised by buffer overflow.

- Each thread has its own protected memory space, except where compatibility with legacy software requires a shared memory space for all threads in an application.
- Device drivers and system functions have limited memory access. Buffer overrun errors can be effectively prevented by controlling which memory area a device driver has access to, since all program input goes through device drivers. A calling program can limit the memory access of a device driver to only a specific input buffer.
- Device drivers and system functions have carefully controlled access rights to input/output ports and other system resources. These access rights are defined in the executable file header of the device driver and controlled by the system core.
- A fault in a device driver should not generate a “blue screen of death”, but generate an error message and close the application that called it and free its resources.
- Application programs have only access to specific resources as specified in the executable file header and controlled by the system.
- Array bounds checking is simple and efficient, using an addressing mode with built-in bounds checking or a simple branch.
- There are various optional methods for checking integer overflow.
- There is no “undefined” behavior. There is always a limited set of permissible responses to an error condition.

How to improve the security of applications and systems

Several methods for improving security are listed below. These methods may be useful in ForwardCom applications and operating systems where security is important.

Protect against buffer overflow

Input buffers must be protected against overflow. This can be done efficiently by limiting the memory access of the device driver that handles the input. Furthermore, it is possible to allocate an isolated block of memory for a data buffer. See page 121.

Protect arrays

Array bounds should be checked.

Protect against integer overflow

Use one of the methods for detecting integer overflow mentioned on page 108.

Protect thread memory

Each thread in an application should have its own protected memory space. See page 120.

Protect code pointers

Function pointers and other pointers to code are vulnerable to control flow hijack attacks. These include:

Return addresses. Return addresses on the stack are particularly vulnerable to buffer overflow attacks. ForwardCom has separate call stack and data stack so that return addresses are isolated from other data.

Jump tables. Switch/case multiway branches are often implemented as tables of jump addresses. These should use the jump table instruction with the table placed in a read-only section. See page 92.

Virtual function tables. Programming languages with object polymorphism, such as C++, use tables of pointers to virtual functions. These should use the call table instruction with the table placed in the a read-only section. See page 92.

Procedure linkage tables. Procedure linkage tables, import tables and symbol interposition are not used in ForwardCom. See page 139.

Callback function pointers. If a function receives a pointer to a callback function as parameter, then keep this pointer in a register rather than saving it to memory.

State machines. If a state machine or similar algorithm is implemented with function pointers then place these function pointers in a read-only array, use a state variable as index into this array and check the index for overflow. The compiler should have support for defining an array of relative function pointers in a read-only section and access them with the call table instruction.

Other function pointers. Most uses of function pointers can be covered by the methods described above. Other uses of function pointers should be avoided in high security applications, or the pointers should be placed in protected memory areas or with unpredictable addresses.

Control access rights of application programs

The executable file header of an application program should include information about which kinds of operations the application needs permission to. This may include permission to various network activities, access to particular sensitive files, permission to write executable files and scripts, permission to install drivers, permission to spawn other processes, permission to interprocess communication, etc. The user should have a simple way of checking if these access rights are acceptable. We may implement a system for controlling the access rights of scripts as well. Web page scripts should run in a sandbox.

Control access rights of device drivers

Many operating systems are giving very extensive rights to device drivers. Rather than having a bureaucratic centralized system for approval of device drivers, we should have a more careful control of the access rights of each device driver. The system call instruction in ForwardCom gives a device driver access to only a limited area of application memory (see page 100). The executable file header of a device driver should have information about which ports and system registers the device driver has access to. The user should have a simple way of checking if these access rights are acceptable.

Standardized installation procedure

The operating system should provide a standardized way of installing and uninstalling applications. The system should refuse to run any program, script or driver that has not been installed through this procedure. This will make it possible for the user to review the access requirements of all installed programs and to remove any malware or other unwanted software through the normal uninstallation procedure.

Malware protection should be an integral part of the operating system, not a third-party add on with possible compatibility problems and performance problems.

Chapter 7

Programmable application-specific instructions

Rather than implementing a lot of special instructions for specific applications, we may provide a means for generating user-defined instructions which can be coded in a hardware description language, e. g. VHDL or Verilog.

The microprocessor can have an optional FPGA or similar programmable hardware. This structure can be used for making application-specific instructions or functions, e. g. for coding, encryption, data compression, signal processing, text processing, etc.

If the processor has multiple CPU cores then each core may have its own FPGA. The hardware definition code is stored in its own cache for each core. The operating system should prevent, as far as possible, that the same core is used for different tasks that require different hardware codes. There may be features for allowing an application to monopolize an FPGA or part of it.

If it cannot be avoided that multiple applications use the same FPGA in the same CPU core, then the code, as well as the contents of any memory cells in the FPGA, must be saved on each task switch. This saving may be implemented as lazy, i. e. the contents is only swapped when the second task needs the FPGA structure that contains code for the first task.

There must be instructions for accessing the user-defined functions, including means for input and output, and for adapting to the latency of the user-defined functions.

Chapter 8

Microarchitecture and pipeline design

The ForwardCom instruction set is intended to facilitate a consistent and efficient design of the pipeline of a superscalar microprocessor. Normal instructions can have no more than one destination operand, up to three source operands, a mask register, and a fallback register. A source operand can be a register, a memory operand, or an immediate constant. An instruction cannot have more than one memory operand. The total number of input registers to an instruction, including source operands, mask, fallback, memory base pointer, index, and vector length specifier cannot exceed five.

Some instruction formats have multiple immediate operand fields. Any extra immediate operand field can be used for option bits, memory offset, or memory limit.

A high performance pipeline may be designed as superscalar with the following stages. Simple designs may have fewer stages. The number of pipeline stages should be as few as possible in order to reduce the latency of non-predicted jumps.

- Fetch. Fetching blocks of code from the instruction cache, one cache line at a time, or as determined by the branch prediction machinery.
- Instruction length decode. Determine the length of each instruction. Distribute the first P instructions into each their pipeline, where P is the number of parallel pipelines implemented. Excess instructions may be queued for the next clock cycle. The length of an instruction is determined by two bits of the first code word in order to simplify this process.
- Instruction decode. Identify and classify all operands, opcode, and option bits. Determine input and output dependencies. A consistent template system simplifies this step.
- Register read.
- Calculate address and length of memory operand. Check access rights.
- Instruction queue.
- Put instructions into reservation station.
- Read memory operand. Schedule for execution units. Do calculations that depend on immediate operand only.
- Execution units.
- Retire or branch.

A disadvantage of register renaming and out-of-order execution is that it makes the pipeline long. This increases the branch misprediction delay. A simpler design may have one or more in-order pipelines for integer instructions in general purpose registers and one pipeline for vector registers. Such a design will have fewer pipeline stages, for example:

- Fetch. Fetching blocks of code from the instruction cache.
- Instruction decode. Identify and classify all operands, opcode and option bits. Determine input and output dependencies.
- Register read. Read any registers needed for address calculation. Other register operands are read as well if the values are available at this stage.
- Calculate address and length of memory operand. Check access rights.
- Read memory operand. Do calculations that depend on immediate operand only.
- Execution units.

It is not necessary to split read-operate instructions into micro-operations if the reading of memory operands is done in a separate pipeline stage and instructions are allowed to wait until the memory operand has been read.

Each stage in the pipeline should ideally require only one clock cycle unless they are waiting for an operand. Most instructions will use only one clock cycle in the execution unit. Multiplication and floating point addition need a pipelined execution unit with several stages. Division and square root may use a separate state machine.

Jump, branch, call, and return instructions also fit into this pipeline design.

A reservation station has to consider all the input and output dependencies of each instruction. Each instruction can have up to four or five input dependencies and one output dependency.

There may be multiple execution units so that it is possible to run multiple instructions in the same clock cycle if their operands are independent.

An efficient out-of-order processing requires renaming of the general purpose registers and vector registers, but not necessarily the special registers.

Some current CPUs have a “stack engine” in order to predict the value of the stack pointer for a push, pop, or call instruction when preceding stack operations are delayed due to operands that are not available yet. Such a system is not needed if we have a separate call stack (see page 105). The depth of function calls in most programs is so small that even a moderately small on-chip call stack would rarely need to spill to main memory.

Branch prediction is important for the performance of a CPUs with long pipelines. We may implement four different branch prediction algorithms: one for ordinary branches, one for loops, one for indirect jumps, and one for function returns. The long form of branch instructions have an option bit for indicating loop behavior. The short form of branch instructions does not have space for such a bit. The initial guess may be to assume loop behavior if the branch goes backwards and ordinary branch behavior if the branch goes forwards. This assumption may be corrected later, if necessary, by the branch prediction machinery. The code following a branch may be executed speculatively until it is determined whether the prediction was right. We may implement features for running both sides of a branch speculatively at the same time. Other ways of reducing branch misprediction delays are discussed on page 116 below.

A full out-of-order design with register renaming requires a lot of chip space for a reservation station typically holding hundreds or pending instructions and hundreds or rename-able temporary registers, including vector registers. A simpler design may prioritize the chip space differently. Instead of a large reservation station, we may have more execution pipelines. Each pipeline is processing its instructions in order, but instructions may execute out of order as long as there are vacant pipelines.

8.1 Vector design

The chip layout of a vector processor is typically divided into “data lanes”. An efficient design may divide the vector processing into multiple lanes of 64 (or 128) bits each. Each lane would have its own register file containing a 64-bit slice of each vector register. There would be very little necessary communication between the lanes as long as the output data stay in the same lane as the input data.

The ForwardCom design allows large vector processors with very long vector registers. The priority of a particular design may be either to have one vector pipeline with very large vectors, or multiple pipelines with a smaller maximum vector lenght.

A consequence of this lane layout is that transfer of data between the lanes is likely to be slower. We may have one or more data buses connecting the lanes, but long distance connections is often a limited resource. This means that instructions that transfer data horizontally across a vector, such as permute instructions, may have longer latencies than other vector instructions.

The scheduler may need to know the instruction latency, and this can be a problem if the latency depends on the distance of data transfer on very long vectors. This problem is addressed by indicating the vector length or the distance of data transfer for such instructions in a separate operand, which always uses the RT register field. This information may be redundant because the vector length is stored in the vector register operands, but the scheduler may need this information as early as possible. The other register operands are typically not ready until the clock cycle where they go to the execution unit, while the vector length is typically known earlier. The microprocessor can read the RT register at the address calculation stage in the pipeline, where it also reads any pointer, index register and vector length for memory operands. This allows the scheduler to predict the latency a few clock cycles earlier. The instruction set provides the extra information about vector length or data transfer length in the RT register for instructions that involve horizontal data transfer, including memory broadcast, permute, insert, extract, and shift instructions, but not broadcasting of immediate constants.

Each vector lane might have its own data cache. This may be easier to implement than one big data cache with a data bus as wide as the maximum vector length. This solution will be efficient as long as large data arrays are aligned to addresses divisible by the maximum vector length, but it will be less efficient if data arrays are misaligned. The data traffic to cache and memory is often the limiting bottleneck in modern computer systems. A system with a separate data cache for each vector lane will make it feasible to have a larger total cache size. This may be a useful alternative to having multiple CPU cores. Data transfer between the lanes will be slower than if we have one cache servicing all lanes, but faster than data transfer between different CPU cores. Whether this is an efficient solution depends on the amount of data permutation needed in a specific application.

8.2 Complex instructions

Some current systems are using a microcode ROM to implement very complex instructions, such as mathematical functions. This has turned out to be inefficient. The use of microcode should be avoided in ForwardCom processors.

A limited amount of complexity can still be implemented with the flexible design of format templates in ForwardCom. Instructions that do multiple things can be useful because they allow the code to do more work per instruction. Complex instructions should only be implemented if they fit into the streamlined template system, pipeline, and timing constraints.

A number of complex instructions have been implemented because they offer a definite advantage at relatively low hardware costs. The most complex instructions may be implemented with state machines rather than microcode. The following instructions have complex functionality:

- Instructions with memory operand can calculate a memory address, read the value of the memory operand, and do some calculation on this operand. Multiformat instructions can have a memory operand with a choice of different addressing modes. This feature has significant hardware costs because it requires a few extra pipeline stages, but it has a high advantage because it allows a single instruction to do what traditionally requires several instructions in a pure RISC design.
- Combined arithmetic and branch instructions. All branches depend on the result of some arithmetic or logic operation. Rather than saving the result of the arithmetic instruction in a flag and then make a branch depending on the flag, it is more efficient to combine these two operations into one instruction. The CPU needs the circuitry for both arithmetic and branching anyway, and there are no big costs to combining these two circuits.
- Loop instructions. It is possible to implement a loop with a single instruction. This instruction can increment a loop counter, compare it with a limit, and branch back if the limit has not been reached. This fits into the general framework of combined arithmetic and branch instructions. Timing problems can be overcome by doing most of the compare operation in parallel with the increment. Another efficient way of making a loop is to decrement a counter down to zero. The special vector loop (see page 13) is also implemented with a single instruction.
- Switch-case statements can be coded efficiently with the jump_relative instruction. This instruction can read a relative pointer from a table of jump addresses, convert it to an absolute address, and jump to this address. Function dispatching can be coded in the same way with the call_relative instruction. These instructions are straightforward to implement in hardware because they fit into the general pipeline design.
- Compare instructions with extra boolean operands. Boolean variables are generated by compare instructions and bit test instructions. Boolean variables are often used as input to boolean operations such as AND, OR, XOR. The compare and bit test instructions can use mask registers and fallback registers as extra boolean operands to be combined with the result of the compare. This makes it possible to combine the results of multiple compare operations without the extra instructions or branches for AND, OR, etc. This feature is quite cheap to implement in hardware.
- mul_add and add_add. These instructions have significant hardware costs for floating point operands. The integer versions are less costly.
- Division and square root. These have high costs in the hardware budget.
- Push and pop instructions. These instructions are complicated to implement in hardware because they require extra functionality in the decoder to generate multiple micro-operations. They do not require extra functionality in the execution unit. These instructions are so useful that it is recommended to implement them despite the extra complexity.
- System calls, traps, and interrupts. These are complicated to implement, but necessary in many cases. They may not be needed in small embedded systems.
- cmp_swap and other instructions for atomic memory operations. This is complicated to implement, but necessary for efficient mutexes in multithreaded systems.

8.3 Proposals for reducing branch misprediction delay

Modern superscalar processors often have quite long pipelines. This gives a long branch misprediction delay. The branch misprediction delay is normally equal to the number of pipeline stages from a branch instruction is fetched until it is executed.

It may be possible to reduce this delay by executing branch instructions as early as possible in the pipeline. Any preceding instructions that a branch instruction depends on may also be executed early. It is proposed to execute all control transfer instructions (branch, jump, call, and return) in the front end of the pipeline, before any register renaming and scheduler. Any preceding instruction that a branch depends on, could likewise be executed in the front end. This idea is somewhat similar to the principle described in the article:
Sheikh, Rami, James Tuck, and Eric Rotenberg. "Control-Flow Decoupling." In Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture, 329–340. IEEE Computer Society, 2012. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.952.6936&rep=rep1&type=pdf

The vision is this: There are two sets of ALU's, one in the front end used for control flow and simple integer instructions, and one in the back end used for all other calculations. The front end does not use register renaming, but relies on the permanent register file.

In most cases, the control flow depends only on simple integer operations such as incrementing a loop counter. The front end ALU's should be able to handle simple integer operations on general purpose registers. The front end may handle all simple integer instructions on general purpose registers if the operands are available. If the capacity of the front end ALU's is insufficient then it may prioritize instructions on those registers that have recently been used as loop counters or branch conditions, but it may be cheaper to implement a few extra ALU's to increase the front end capacity than to implement such a mechanism for prioritization.

Instructions that are executed in the front end use the permanent register file for both input and output. The remaining instructions are sent to the out-of-order back end that may use register renaming. All general purpose register operands are replaced by their value if the value is available before the instruction is sent to the back end. Instructions that execute in the front end may write their result not only to the permanent register file but also to any pending instruction that needs it.

The result of this mechanism is that a loop that is controlled only by simple instructions on general purpose registers will be unrolled in the front end. The instructions in the loop body may be passed on to the back end with the loop counter replaced by its value. The back end may have register renaming so that it can execute the body of the second iteration before it has finished the body of the first iteration, etc.

All other control flow, such as branches, jumps, calls, returns, etc., will be unrolled in the front end as well so that control flow is effectively resolved before execution as far as it does not depend on delayed data.

The front end may support a minimum of out-of-order execution in the sense that an instruction that is waiting for an operand should not delay any independent subsequent instructions. But the front end does not need a complicated structure with a long queue, register renaming, and scheduler.

The front end should have access to the permanent register file, while the back end may use temporary registers with register renaming. The renamed registers will be written to the permanent register file when they retire. We have to keep track of whether the permanent registers are up to date. When the decoder sees an instruction that writes to a register, it will mark this register as invalid in the permanent register file and add a tag to tell which instruction it belongs to. When this instruction is executed (whether in the front end or the back end), the register entry is marked as valid if the tag is matching the instruction that delivered the result.

The front end may fetch and decode speculatively, but not execute speculatively. It may even fetch both sides of a branch that it expects to have poor prediction. Fetching both sides of a branch becomes cheaper here than with a traditional superscalar design because it only needs to duplicate two pipeline stages (fetch and decode) in order to minimize branch misprediction delays.

The fetch and decode stages should have higher throughput than the rest of the pipeline so that it can catch up and fill the pipeline after a branch delay. The fetch stage should be able to fetch a large block of code in one clock cycle. The decoder should be able to decode the first one or two instructions in a fetched code block in the first clock cycle, and determine the starting points of the remaining instructions. This will allow it to decode maybe four or more instructions in the second clock cycle after fetching a block of code. The aim of this is to keep the branch misprediction delay as low as possible.

Multi-way branches with a jump table are particularly critical here because they have a memory operand that may be delayed due to a cache miss. Implementing memory access in the front end may be too complicated. Therefore, such instructions probably have to go to the back end. The out-of-order mechanism may give high priority to these instructions and execute them as soon as the register operands are available. Jump tables are normally placed in the read-only memory block (see page 119). The speed of access to jump tables may be increased by having a separate cache for read-only memory.

An extra advantage of this design is that the registers used in memory operands for pointer, index, and vector length are likely to be available early so that memory operands can be fetched before they are needed.

The demand for advanced branch prediction is somewhat relaxed if we can drastically reduce the branch misprediction delay with the measures described here. The branch prediction machinery in modern processors is very complicated and requires a lot of chip space for branch target buffers and pattern/history tables. We can cut down on this and use a simpler branch prediction scheme if branches are resolved in the front end. A few extra ALU's in the front end will use much less chip space than an advanced branch prediction mechanism.

The early processing of jumps and branches is also possible with a simpler design without register renaming and out-of-order processing. We may have a fast-track in-order pipeline for branches and simple integer instructions without memory operands. Branches often depend only on loop counters and other simple calculations in general purpose registers. These instructions can be executed in the fast-track pipeline in most cases, while the more complicated loop body instructions go to a longer full-featured pipeline. Direct call and return instructions can be executed very early in the pipeline if we have an on-chip call stack because the call stack pointer is independent of the other instructions and registers.

Chapter 9

Memory model

ForwardCom has a flat address space using unsigned 64-bit addresses and 64-bit pointers. Future extension to 128-bit addresses is possible, but this will probably not be relevant in a foreseeable future.

Absolute addresses are rarely used. Most data objects, functions and jump targets are addressed with signed offsets of 32 bits or less relative to some reference point contained in a 64-bit pointer. This pointer can be the instruction pointer (IP), the data section pointer (DATAP), the thread data pointer (THREADP), the stack pointer (SP), or a general purpose register.

An application can have access to the following sections of data:

- Program code (CODE). This memory block is executable, but usually without read and write access. The CODE section can be shared between multiple processes running the same program.
- Read-only program data (CONST). This contains constants and tables used by the program without write access. It may be shared between multiple processes.
- Static read/write program data sections, which can be initialized (DATA) and uninitialized (BSS). This is used for global data and for static data inside functions. Multiple instances are needed if multiple processes are running the same code.
- Stack data (STACK). This is used for non-static data inside functions. Each process or thread has its own stack, with addresses relative to the stack pointer. The stack grows downward from high to low addresses when data are added to the stack.
- Thread data (THREADD). Allocated when a thread is created and used for thread-local static data and thread environment block. This has one instance for each thread.
- Program heap (HEAP). Used for dynamic memory allocation by an application program.

References within the CODE section use 8-bit, 16-bit, 24-bit, and 32-bit signed references relative to the instruction pointer, scaled by the code word size which is 4 bytes.

A CONST section may be placed either near the CODE section and addressed relative to the instruction pointer (IP), or together with the other data sections and addressed relative to the the data section pointer (DATAP).

The DATA and BSS sections are addressed relative to the data section pointer (DATAP) which is a special register that points to some reference point in these sections. The default reference point is where DATA ends and BSS begins. Multiple running instances of the same program will have different values of the data section pointer. The CODE and CONST sections contain no absolute references to DATA or BSS, only references relative to the data section pointer. This makes it possible for multiple processes to share the same CODE and CONST sections, but have each their private DATA and BSS sections without the need for virtual address translation.

The DATA and BSS sections can be placed anywhere in the address space independently of where CONST and CODE are placed.

STACK data are addressed relative to the stack pointer (SP). Heap data are addressed through pointers provided by a heap allocation function.

Thread data are addressed relative to the thread data pointer (THREADP), which is separate for each thread in the process. The thread data section may be allocated on the stack when a new thread is created.

The STACK, DATA, BSS, and HEAP data sections are preferably kept together in one contiguous block in order to optimize caching and memory management.

This model allows the program to access up to 8 GB of CODE, 2 GB of CONST, 2 GB of DATA, 2 GB of BSS, 2 GB of THREADD, an almost unlimited size of STACK with 2 GB frames, and an almost unlimited amount of HEAP data.

A pointer to the CONST section may be provided in the thread environment block in order to access CONST data in case the CONST section is placed with a distance of more than 2 GB from IP or DATAP or if the CONST section is relocated independently of CODE and DATA.

Padding space

Any read/writeable data section that is not followed by another readable section may have an unused space at the end, at least the size of the maximum vector length. The purpose of this extra space is to make it possible for the pop instruction to read more than necessary when restoring a vector of unknown length, and to make it possible to search for the end of a zero-terminated string without having to read only a single byte at a time. This does not apply to IP-addressed read-only data sections that contain no strings, because a variable-size unused space between the constant section and the code section would make it necessary to recalculate relative addresses when a program is loaded on a system with a different maximum vector length, which would reduce the advantage of position-independent code.

An alternative solution is to allow vector reads that begin in a readable section to extend past the end of the readable section without raising an exception. The vector part that is outside the permitted section must get the value zero. It remains to be investigated which of these two solutions is most efficient.

Stack direction

Most microprocessor systems have the data stack growing backwards. This also applies to the ForwardCom system, but mainly for a different reason. When a vector register is saved on the stack by the push instruction, it is stored as the length followed by the amount of data indicated by the length. When the vector register is restored using the pop instruction, it is necessary to read the length followed by the data. The stack pointer must point to the low end where the length is stored, otherwise it would be impossible to find where the length is stored.

It is possible to make additional stacks growing forwards or backwards, but any stack containing variable-length vectors must grow backwards.

9.1 Thread memory protection

Each thread has its own stack. The thread data (THREADD) may be placed on this stack. The ForwardCom system allows inter-thread memory protection. The stack data of the main thread of a program is accessible to all its child threads, but all other threads in the program can have

private data which is not accessible to any other threads, not even to the main thread. Any communication and synchronization between threads must use static memory or memory belonging to the main thread.

It is recommended to use this inter-thread memory protection in all cases except where legacy software requires one memory space shared by all threads.

Isolated memory blocks

It is possible to make a system function that allocates an isolated memory block surrounded by inaccessible memory on both sides. Such a memory block, which will be accessible only to a specific thread, can be used for example for a data buffer in cases where security requirements are high. Each thread can have only a limited number of such protected memory blocks because of the limited size of the memory map.

9.2 Memory management

It is a design goal to minimize memory fragmentation and to minimize the need for virtual address translation. Traditional designs often have very complicated memory management systems with multilevel address translation, large translation-lookaside-buffers (TLB), and huge page tables.

The reason why many current systems have large page tables with fixed-size memory pages is that this is the only efficient way to look up an address in a heavily fragmented memory space. It is a design goal of ForwardCom to minimize or eliminate memory fragmentation. This is achieved in several ways explained below. If there are only a few memory blocks visible to a particular thread, then it is possible to have variable-size memory blocks. It requires only a limited number of hardware comparators to look up an address in a table of only a few memory blocks with variable size. The ambition is to replace the TLB, which has a large number of fixed-size memory blocks, by a memory map with a few memory blocks of variable size. A memory map with such a limited number of entries can easily be implemented on the chip in a very efficient way and it can easily be changed on task switches. Each process and each thread must have its own memory map.

In most cases, the main thread of an application will only need three blocks of memory: CONST (read only), CODE (execute only), and the combined STACK+DATA+BSS+HEAP (read-write). A child thread needs one more entry for its private stack. Similar blocks are defined for system code.

The memory map supports virtual address translation in the form of a constant offset that defines the distance between the virtual address and the physical address for each map entry. The hardware should not waste time and power on virtual address translation when it is not used.

A limited number of extra entries are needed in the memory map to deal with cases where the memory becomes fragmented, but memory fragmentation can be avoided in most cases. The following techniques are provided to simplify memory management and avoid memory fragmentation:

- There is only one type of function libraries which can be used for both static and dynamic linking. These are linked with a mechanism that keeps the CONST, CODE and DATA sections contiguous with the similar sections of the main program. It is described on page 139 below how this avoids memory fragmentation and improves caching.
- The required stack size is calculated by the compiler and the linker so that stack overflow can be avoided in most cases. This technique is described on page 141.

- The operating system can keep statistical records of the heap use of each program in order to predict the required heap size. The same technique can be used for predicting stack use in cases where the required stack size cannot be predicted exactly (e. g. recursive function calls).

The memory space may become fragmented despite the use of these techniques. Problems that can result in memory fragmentation are listed below.

- Recursive functions can use unlimited stack space. We may require that the programmer specifies a maximum recursion level in a pragma.
- Allocation of variable-size arrays on the stack using the alloca function in C. We may require that the programmer specifies a maximum size.
- Runtime linking. The program can reserve space for loading and linking function libraries at run time (see page 139). The memory may become fragmented if the memory space reserved for this purpose turns out to be insufficient.
- Script languages and byte code languages. It is difficult to predict the required size of stack and heap when running interpreted or emulated code. It is recommended to use a just-in-time compiler instead. Self-modifying scripts cannot be compiled. The same problem can occur with large user-defined macros.
- Unpredictable number of threads without protection. The required stack size for a thread may be computed in advance, but in some cases it may be difficult to predict the number of threads that a program will generate. Multiple threads will mostly share the same code sections, but they need separate stacks. The stack of a thread can be placed anywhere in memory without problems if inter-thread memory protection is used. But if memory is shared between threads and the number of threads is unpredictable then the shared memory space may become fragmented.
- Unpredictable heap size. Programs that process large amounts of data, e. g. multimedia processing, may need a large heap. A program heap can use discontiguous memory, but this will require extra entries in the memory map.
- Lazy loading and code overlay. A large program may have certain code units that are rarely used and loaded only when needed. Lazy loading can be useful to save memory, but it may require virtual memory translation and it may cause memory fragmentation. A straightforward solution is to implement such code units as separate executable programs.
- Hot patching, i. e. updating of code while it is running.
- Shared memory for inter-process communication. This requires extra entries in the memory map as explained below.
- Many programs running. The memory can become fragmented when many programs of different sizes are loaded and unloaded randomly or swapped to memory.
- Memory mapping of files and devices.

A possible remedy against overflow of stack and heap is to place the STACK, DATA, BSS and HEAP data together (in this order) in an address range with large unused virtual address spaces below and above, so that the stack can grow downwards and the heap can grow upwards into the vacant spaces. This method can avoid fragmentation of the virtual address space, but not the physical address space. Fragmentation of the physical address space can be remedied by moving data from a memory block of insufficient size to another block that is larger. This method has the cost of a time delay when the data block is moved.

If runtime linking runs into memory problems and lack of memory map entries then it is allowed to mix CONST and CODE sections together in a common section with both read and execute access. If a library function contains constant data that originate from an untrusted source, while the code is trusted, then it is preferred to put the untrusted data into the DATA section rather than the CONST section in order to prevent execution of malicious code placed in the CONST section.

An application that needs many independent memory blocks can make a separate thread or subprocess to service each memory block. Each thread has its own memory map with an entry for its private memory block.

Shared memory can be used when there is a need to transfer large amounts of data between two processes. One process shares a part of its memory with another process. The receiving process needs an extra entry in its memory map to indicate read and/or write access rights to the shared memory block. The process that owns the shared memory block does not need any extra entry in its memory map. There is a limit to how many shared memory blocks an application can receive access to, if we want to keep the memory map small. If one program needs to communicate with a large number of other programs then we can use one of these solutions: (1) let the program that needs many connections own the shared memory and give each of its clients access to one part of it, (2) run multiple threads in (or multiple instances of) the program that needs many connections so that each thread has access to only one shared memory block, (3) let multiple communication channels use the same shared memory block or parts of it, (4) communicate through function calls, (5) communicate through network sockets, or (6) communicate through files.

Executable memory cannot be shared between different applications. A function that is used by multiple different applications will usually be placed in a function library that these applications are linking to. The mechanism of interprocess calls must be used if one application needs to call a function in another application. This is described on page 127.

Software that relies heavily on memory mapping should be avoided. As the trend in microprocessor technology goes towards an increasing number of CPU cores, it should be possible to give critical applications and critical threads each their core with its own memory space so that context switching and memory fragmentation can be avoided or minimized.

Conclusion

We can probably keep memory fragmentation low by using the principles discussed here so that a relatively small memory map for each thread will be sufficient to cover normal cases. This will be much more efficient than the large TLB and multilevel address translation of current designs. It will save a lot of chip space and power. We can avoid the cost of TLB misses and page faults, and it will make task switches faster. The actual size of the memory map will depend on the hardware implementation.

This system puts the priority on performance-critical applications. The programmer will be able to get top performance by observing some discipline to avoid memory fragmentation, for example by recycling allocated memory. However, the system should also be able to run less well-behaved applications. Applications that cause heavy memory fragmentation are probably built with less regard for performance. The system should have methods to allow such applications to work, perhaps even software based methods to deal with memory fragmentation, but we can regard it as acceptable to make a system that prioritizes the performance of well-behaved applications at the cost of inferior performance for applications that cause heavy memory fragmentation.

Chapter 10

System programming

The system instructions have not been fully defined yet. There is more work to do making an efficient system design. However, the first experimental implementations of ForwardCom do not include an operating system so the details of system design do not have to be fixed yet. It is preferred to spend more time on optimizing the system design rather than to define a complete standard at this stage of development.

There should be at least three different levels of privilege:

- The system core has the highest privilege level. Memory management and thread scheduling takes place here. This is the only part that can modify memory maps and control access rights at the lower levels.
- Device drivers and system plugin modules have carefully controlled access rights. A structure similar to the memory map (see page 121) gives a device driver access to the particular range of input/output ports and system registers that it needs. A user application can give a device driver read and write access to a specific range of the data memory it owns. This is done through the system call instruction. A device driver has no access to the code memory of the application that calls it. This means that callback function pointers cannot be used with system calls.
- An application program has access to only the memory that is allocated to it or shared with it. Memory belonging to a thread is usually not shared with other threads in the same process. Application programs have access to a few system registers and no input/output ports.

Transitions between these levels are managed by the system call and system return instructions and by traps and interrupts.

There are various system registers for control purposes. In addition, there may be two sets of registers used for temporary storage, one set for the device driver level and one for the system core level. The temporary registers for the device driver level are cleared for security reasons every time a device driver is called. These registers are used mainly for temporary saving of the general purpose registers.

10.1 Memory map

There are three kinds of memory access: read, write, and execute access. These kinds of access are separate, but can be combined. For example, execute access does not imply read access. Write access and execute access should not normally be combined, because self-modifying code is discouraged.

The memory map is stored in the CPU chip. Each entry has three fields: A virtual address (up to 64 bits), access rights (3 bits), and an addend for address translation (up to 64 bits). There is no memory paging. Instead, the memory blocks have variable sizes.

The entries in the memory map must be kept sorted at all times so that each memory block ends where the next block begins. The addresses must be divisible by 8. Each thread has its own memory map. A typical memory map for an application thread may look like this.

Table 10.1: Example of memory map

Start address	Access	Addend	Comment
0x10000	Read	0	CONST section
0x10100	Execute	0	CODE section
0x10800	None	0	Belongs to other processes
0x20000	Read, Write	0	Main STACK, DATA, BSS, and HEAP sections
0x24000	None	0	Belongs to other processes
0x30000	Read, Write	0	Thread STACK, thread environment block, and tread static data
0x32000	None	0	The rest belongs to other processes

There may be a few further entries for memory blocks shared between processes and for secure isolated memory blocks. A virtual memory block may have multiple entries in case the memory becomes fragmented. The addends are used for keeping the virtual addresses of the block contiguous while the physical addresses are noncontiguous. The start addresses are virtual memory addresses.

The size of the memory map is variable. The maximum size is implementation dependent. A scalar memory access cannot cross a memory map boundary. A vector memory access cannot cross more than one memory map boundary.

There may be multiple memory maps on the chip, one for each privilege level. This makes transitions between the levels fast. The chip space used for memory maps may be reconfigurable so that the memory maps of multiple processes can remain on the chip in case the memory maps are small. This makes task switching faster.

The memory maps are controlled at the system core level. The instructions read_memory_map and write_memory_map may use the vector loop mechanism for fast manipulation of memory maps.

The methods described on page 121 for avoiding memory fragmentation are important for keeping the memory maps small.

Task switches can be very fast because we have replaced the large page tables and translation-lookaside-buffer (TLB) of traditional systems with a small on-chip memory map. This makes the system suitable for real-time operating systems.

10.2 Call stack

It is preferred to have a separate call stack as discussed on page 104.

The two-stack system has the call stack stored inside the CPU rather than in RAM memory. A method may be required for saving this stack to memory when it is full. This method may use vector-size memory access. It should be possible to manipulate the call stack for task switches and for stack unrolling in the exception handler.

10.3 System calls and system functions

Calls to system functions are made with a system call instruction (sys_call). The system functions are identified by ID numbers rather than addresses. Each ID number consists of a function ID in the lower half and a module ID in the upper half. The module ID identifies a system module or device driver. The system core has ID = 0 and basic system functions have ID = 1. Each part of the ID can be either 16 bits or 32 bits so that the combined ID is 32, 48, or 64 bits.

System add-on modules and device drivers do not necessarily have fixed ID numbers because this would require some central authority to assign these ID numbers. Instead, the program will have to translate the module name to an ID number before the first call to a module. This translation is done by a system function with a fixed ID number. The functions within a module can have fixed or variable ID numbers.

The ID number of a system function can be put into the program in three ways:

1. The most important system functions have fixed ID numbers which can be inserted at compile time.
2. The ID number can be found at load time in the same way as load-time linking works. The loader will find the ID number and insert it in the code before running the program.
3. The ID number is found at run time before the first call to the desired function.

The calling convention for system functions is the same as for other functions, using registers for parameters and for return value. The registers used for parameters are determined by the general calling convention. The calling conventions are described in chapter 12.4. The parameter registers should not be confused with the operands for the system call instruction.

The system call instruction can have four operands. The first operand (RD) is a pointer to a memory block that may be used for transferring data between the calling program and the system function. The second operand (RS) is a register containing the size of this memory block. The third operand is the module ID specified as a constant. The fourth operand is the function ID specified as a constant. The last two two parameters may be combined into a 64-bit register.

The calling thread must have access rights to the memory block that it shares with the system function. This can be read access or write access or both. These access rights are transferred to the system function. The system function has no access rights to any other part of the application's memory.

A system call that contains zero-terminated strings must provide access to at least 256 bytes beyond the end of the string so that the system function does not have to read only a single character at a time when searching for the end of the string.

It is not possible to use callback function pointers with a system call because executable memory cannot be shared with a system function. If necessary, the system function can call an exported function provided by the application, using the method for inter-process calls described below.

Device driver functions may use separate stacks. The system call goes first to the system core which assigns a stack to the device driver function and makes a memory map for it before dispatching the call to the desired function. Preferably, no stack is used during this dispatching. The two registers identifying a shared memory block are copied to special registers which are accessible to the called system function. The system function runs in the same thread as the application that called it, but not with the same stack.

The old values of instruction pointer, stack pointer, and DATAP are saved in system registers, to be restored when returning to the application code.

System functions, device drivers and interrupt handlers are allowed to use all general purpose registers and vector registers if they are saved and restored according to the normal calling conventions. Interrupt handlers must save and restore all registers they use.

A method is provided to get information about the register use of system functions so that it is possible to call them using the register usage conventions of either method 1 or method 2, described on page 137. The stack use of system functions is irrelevant for the caller because they do not use the stack of the calling application program.

Some important system functions must be standardized and must be available in all operating systems. This will make it possible, for example, to make a third-party function library that works in all operating systems, even if this library needs to call system functions. It will also make it easier to adapt a program for different operating systems. The list of system functions that might be standardized includes functions for thread creation, thread synchronization, setting thread priority, memory allocation, time measurement, system information, etc.

There should be a selection of system libraries providing the most common user interface forms, such as graphical user interface, console mode, and server mode. These user interface system libraries should be provided for each operating system that the architecture can run on, so that the same executable program can run in different operating systems simply by linking with the appropriate user interface library when the program is installed. Such user interface libraries may be based on existing platform-independent GUI libraries such as, e. g., wxWidgets or QT. All user interface libraries must support the error_message function mentioned below.

10.4 Inter-process calls

Inter-process calls are mediated by a system function. This works in the following way. An application program can export a function with an entry in its executable file header. Another application can get access to this exported function by calling a system function that checks for permission and switches the memory map, the DATAP and THREADP registers and the stack pointer before calling the exported function, and switches back before returning to the caller. The call will appear as a separate thread to the called program. The general purpose registers and vector registers can be used for parameters and return value in the same was as for normal functions. This mechanism does not generate any shared memory between caller and callee. Therefore, the exported function must use only simple types that fit into registers for its parameters and return type. A block of memory can be shared between the two processes as described on page 123.

10.5 Error message handling

There is a need for a standardized way of reporting errors that occur in a program. Many current systems fail to satisfy this need, or they use methods that are not portable or thread-safe. In particular, the following situations would benefit from such a standard.

1. A function library detects an error, for example an invalid parameter, and needs to report the error to the calling program. The calling program will decide whether to recover from the error or terminate.

2. A trap is generated because of a numerical error. The program fails to catch it as an exception, or the programming language has no support for structured exception handling. The operating system must make an informative error message.

3. A program can run in different environments that require different forms of error reporting.

4. A function library in source code form, a class library, or any other piece of code needs to report an error without knowing which user interface paradigm is used (e. g. console mode or graphical user interface or server mode). It needs a standardized way of reporting the error to the operating system or to the user interface framework, which must present an

error message to the user in the way that is appropriate for the user interface (e. g. pop up a message box, print to stderr, print to a log file, or send a message to an administrator).

It is proposed to define a standard library function named error_message for this purpose. All user interface frameworks must define this function. It is possible to choose between different versions of this function when the program is installed by linking with the appropriate library. The main program may override this function by defining its own function with the same name.

The error_message function must have the following parameters: a numerical error code, a string pointer giving an error message, and another string pointer giving the name of the function where the error occurred. These strings are coded as zero-terminated UTF-8 strings. The error message is in the English language by default. It is not reasonable to require support for many different languages (see this link for a discussion of problems with internationalization). Instead, a manual in the desired language can contain a list of error codes.

The error message string may include numerical values and diagnostic information, such as the value of a parameter that is out of range.

The error_message function may or may not return. If it returns then the function that called it must return in a graceful way. The error_message function may alternatively terminate the application or it may raise an exception or trap which is handled by the operating system in case the exception is not caught by the program.

Chapter 11

Support for multiple instruction sets

A microprocessor may support multiple instruction sets. It will be useful for compatibility with legacy software that a microprocessor with support for ForwardCom also supports one or more older instruction sets. Such a microprocessor will have different modes, one for each instruction set. Instructions of different instruction sets cannot be mixed freely. Virtualization support might be useful so that we can have one virtual machine for each instruction set.

The transitions between ForwardCom and other instruction sets can be implemented with instructions dedicated to this purpose or with software interrupts or system calls.

If mode transitions are allowed only in system code then it is not necessary to save any registers across the mode switch. But if mode transitions are possible also in application code then the following principles should be applied:

- The instruction that makes the mode switch must be aligned to an address divisible by 4.
- The hardware or operating system must make sure that all general purpose registers are preserved across a mode switch between ForwardCom and another instruction set if similar registers exist in the other instruction set.
- The software must set up stack pointers and other special registers immediately before or after the mode switch.
- The software must take care of differences in function calling conventions between the two modes.
- The hardware or operating system must make sure that the contents of at least the first eight vector registers is preserved across the mode switch. The remaining vector registers must be cleared if they are not preserved.
- If the target instruction set has a lower maximum vector length than the actual length of a vector register before conversion, then the length is truncated to the maximum length for the target instruction set.
- The vector length information is lost in the transition from ForwardCom to another instruction set if the other instruction set has no similar way of representing vector length.
- The length of each vector register is set to a value that is sufficient to contain the nonzero part of the register, or longer, when converting from another instruction set to ForwardCom,

Details that are specific to a particular other instruction set are discussed in the following sections.

Transitions between ForwardCom and x86-64

Transitions between ForwardCom and the x86-64 instruction set (with the AVX512 extension) involve the following registers:

Table 11.1: ForwardCom and x86-64 registers

x86-64	Forward-Com	Comments
rax rcx rdx rbx rsp rbp rsi rdi r8 - r15	r0 r1 r2 r3 r4 r5 r6 r7 r8 - r15	 Stack pointer. Must be set before or after conversion to x86-64.
	r16-r30 r31	Not converted Stack pointer. Must be set after conversion to ForwardCom.
flags		Flags register. Not converted.
k0 - k7		Mask registers. Not converted.
zmm0 - zmm7 zmm8 - zmm31	v0 - v7 v8 - v31	Vector registers. Converted. Vector registers. Converted or cleared.

It is possible to make multi-mode functions that can be called from either ForwardCom or x86-64 mode in the following way. The first four bytes of the multi-mode function consist of a short x86-64 jump instruction, which is two bytes long, followed by two bytes of zero. The jump leads to an x86-64 implementation of the code. The four bytes will be interpreted as a NOP (no operation) if the processor is in ForwardCom mode. After this follows a ForwardCom implementation of the function.

Transitions between ForwardCom and ARM

Transitions between ForwardCom and the ARM (AArch64) instruction set involve the following registers:

Table 11.2: ForwardCom and ARM registers

ARM	Forward-Com	Comments
r0 - r30	r0 - r30	
r31	r31	Stack pointer in both instruction sets
v0 - v7	v0 - v7	Vector registers. Converted
v8 - v31	v8 - v31	Vector registers. Converted or cleared
p0 - p15		Predicate registers. Not converted

The Scalable Vector Extensions (SVE) is a future extension to the ARM architecture that allows the length of vector registers to vary from 128 to 2048 bits in increments of 128 bits. The vector length in SVE is apparently controlled through predicate masks. The vector length information cannot be converted to ForwardCom because there is no unambiguous connection between each vector register and the predicate register that contains the length information.

Transitions between ForwardCom and RISC-V

Transitions between ForwardCom and the RISC-V instruction set involve the following registers:

Table 11.3: ForwardCom and RISC-V registers

RISC-V	Forward-Com	Comments
x0		Always zero.
x1 - x31	r1 - r31	General purpose registeers. x1 = link register, x14 = stack pointer, x15 = thread pointer.
f0 - f7	v0 - v7	Floating point and vector registers. Converted.
f8 - f31	v8 - v31	Floating point and vector registers. Converted or cleared.

The SIMD / Vector Extensions to RISC-V are not fully developed yet (January 2017). It is therefore too early to tell whether the vector length information can be converted in a useful way during transitions between ForwardCom and RISC-V.

Chapter 12

Standardization of ABI and software ecosystem

The goal of the ForwardCom project is a vertical redesign that defines new standards not only for the instruction set, but also for the software that uses it. This will have the following advantages.

- Different compilers will be compatible. The same function libraries can be used with different compilers.
- Different programming languages will be compatible. It will be possible to compile different parts of a program in different programming languages. It will be possible to compile a function library in a programming language different from the program that uses it.
- Debuggers, profilers, and other development tools will be compatible.
- Different operating systems will be compatible. It will be possible to use the same function libraries in different operating systems, except if they use system-specific functions.

The previous chapter described standardization of system calls, system functions, and error messaging. The present chapter discusses standardization of the following aspects of the software ecosystem.

- Compiler support.
- Binary data representation.
- Function calling conventions.
- Register usage conventions.
- Name mangling for function overloading.
- Binary format for object files and executable files.
- Format and link methods for function libraries.
- Exception handling and stack unrolling.
- Debug information.
- Assembly language syntax.

12.1 Compiler support

Compilers can have three different levels of support for variable-length vector registers.

Level 1

The compiler will not use variable-length vectors. The compiler can call a vector function in a function library with a scalar parameter if the function is not available in a scalar version.

Level 2

The compiler can call vector functions, but not generate such functions. The compiler can vectorize a loop automatically and call a vector library function from such a loop.

Level 3

Full support. The compiler supports data types for variable-length vectors. These data types can be used for variables, function parameters, and function returns. Variable-length vectors can not be included in structures, classes, or unions because such composite types must have known sizes. Variable-length vectors cannot be stored in static and global variables. General operations on variable-length vectors can be specified explicitly, including options for applying boolean vector masks and fallback values.

Other compiler features

The compiler may support relative pointers and pointer arithmetic on function pointers in order to write compact call tables with relative addresses (see page 163). The difference between two function pointers should be scaled by the code word size, which is 4. Without this feature, the function pointers have to be type cast to integer pointers and back again.

The compiler may have support for detecting integer and floating point overflow and other numerical errors using one of the methods discussed on page 108.

The compiler may support array bounds checking using the indexed addressing mode with bounds or simple conditional jumps to an error function.

12.2 Binary data representation

Data are stored in little-endian form in RAM memory. See page 103 for the rationale.

Integer variables are represented with 8, 16, 32, 64, and optionally 128 bits, signed and unsigned. Signed integers use 2's complement representation. Integer overflow wraps around, except in saturated arithmetic instructions.

Floating point numbers are coded with half precision (16 bits), single precision (32 bits), and double precision (64 bits). Support for quadruple precision (128 bits) is optional. Floating point numbers are coded in the binary format according to the IEEE 754-2019 standard, or later. Subnormal numbers are supported for half precision. Support for subnormal numbers in other precisions is optional.

Floating point errors are preferably indicated with NAN payloads as explained on page 106, rather than with traps. Trapping of signaling NANs is not required. Error information in NAN payloads is propagated as discussed on page 107. The highest payload is propagated when two different NANs are combined.

Boolean variables are stored as integers of at least 8 bits with the values 0 and 1 for FALSE and TRUE. Only bit 0 of the boolean variable is used, while the other bits are ignored. This rule makes it possible to use boolean variables as masks and to implement boolean functions such as AND, OR, XOR, and NOT in an efficient way with simple bitwise instructions, rather than the method used in many current systems that have a branch for each variable to check if the whole

integer is nonzero. A branch instruction is needed in the compilation of expressions like (A && B) and (A || B) only if the evaluation of B has side effects.

All variables not bigger than 8 bytes stored in memory must be kept at their natural alignment.

Arrays not smaller than 8 bytes must be aligned to addresses divisible by 8. It may be recommended to align large arrays by the maximum vector length or by the cache line size.

Multidimensional arrays are stored in row-major order, except where the programming language makes this impossible.

Text strings may be stored in language-dependent forms, but a standardized form is needed for system functions and for functions that are intended to be compatible with all programming languages. The proposed standard uses UTF-8 encoding. The length of the string may be determined by a terminating zero or a length specifier, or both. The rationale is this. The CPU processing time is insignificant for text strings of a length suitable for human reading. The priority is therefore on compactness. Compactness matters if the string is stored in a file or transmitted over a network. UTF-8 is more compact than UTF-16 in most cases, though less compact for some Asian languages. UTF-8 is the most common encoding used on the Internet.

12.3 Further conventions for object-oriented languages

Object oriented languages require further standards for the binary representation of special features such as virtual function tables, runtime type identification, member pointers, etc.

These details must be standardized within each programming language for the sake of compatibility between different compilers, and if possible also between different programming languages that have compatible features.

Member pointers should be implemented in a way that prioritizes good performance in the general case where only a simple offset (to data) or a pointer (to a function) is required, while additional information for contrived cases of multiple inheritance is added only when needed. Data member pointers contain the offset to the data member relative to the “this” pointer, while the value -1 represents a NULL member pointer.

12.4 Function calling convention

Function calls will use registers for parameters as much as possible. Integers of up to 64 bits, pointers, references, and boolean scalars are transferred in general purpose registers. Vector parameters can have variable length. Floating point scalars, vectors of any type with a fixed length of up to 16 bytes, and vectors of variable length are transferred in vector registers.

The first 16 parameters to a function that fit into a general purpose register are transferred in register r0 – r15. The first 16 parameters that fit into a vector register are transferred in v0 – v15. The length of a variable-length vector parameter is contained in the same vector register that contains the data.

Composite types are transferred in vector registers if they can be considered ”simple tuples” no bigger than 16 bytes. A simple tuple is a structure or class or encapsulated array for which all non-static elements have the same type, which is not a pointer or a reference. A union is treated as a structure according to its first element.

Parameters that do not fit into a single register are transferred by a pointer to a memory object allocated by the caller. This applies to: structures and classes with elements of different types, or bigger than 16 bytes. It also applies to objects that require special handling such as a non-standard copy constructor or destructor, and objects that require extra implicit storage such as

tables of virtual member functions. It is the responsibility of the caller to call any copy constructor and destructor.

If there are not enough registers for all parameters then the additional parameters are provided in a list, which can be stored anywhere in memory. A pointer to this parameter list is transferred in a general purpose register. Such a list is also used if there is a variable argument list. There can be no more than one parameter list, as the same list is used for all purposes.

The rules for a parameter list are as follows. A parameter list is used if there are more than 16 parameters that fit into a general purpose register, if there are more than 16 parameters that fit into a vector register, or if there is a variable argument list. If there are less than 16 general purpose parameters then these parameters are put in general purpose registers, and the next vacant general purpose register is used as pointer to the list. If there are 16 or more general purpose parameters, and a parameter list is needed for any reason, then the first 15 general purpose parameters are put in r0-r14, the list pointer is in r15, and the remaining general purpose parameters are put in the list. If there are more than 16 vector parameters then the first 16 vector parameters are put in v0-v15 and the remaining vector parameters are put in the list. All parameters in the list are placed in the order they appear in the function definition, regardless of type. Variable arguments are placed last in the list because they always appear last in a function definition.

The list consists of entries of 8 bytes each. A general purpose parameter uses one entry. A vector parameter with a constant size of 8 bytes or less uses one entry. A vector parameter with a constant size of more than 8 bytes or a variable size uses two entries in the list. The first entry is the length (in bytes) and the second entry is a pointer to an array containing the vector. A parameter that would not fit into a register, if one was vacant, is transferred by a pointer in the list according to the same rules as if the pointer was in a register.

The parameter list belongs to the called function in the sense that it is allowed to modify parameters in the list if they are not declared as constant parameters. The same applies to arrays and objects with a pointer in the list. The caller can rely on parameters in the list being unchanged only if they are declared constant. The caller must put the list in a place where it cannot be modified by other threads.

Function return values follow the following rules: A single return value is returned in r0 or v0, using the same rules as for function parameters.

Multiple return values of the same type are treated as a tuple if possible and returned in v0 if the total size is no more than 16 bytes.

A function with two return values will use two registers for return, using two of the registers r0, r1, v0, v1 as appropriate, if each of the two values will fit into a single register according to the above rules. For example, a function can return a result in v0 and an error code in r0. Or a function can return two vectors of variable length.

In all cases where the return value or set of return values does not fit into at most two registers according to the above rules, the return value is placed in a space allocated by the caller through a pointer transferred by the caller in r0 and returned in r0. Any constructor is called by the callee.

A “this” pointer for a class member function is transferred in r0, except if r0 is used for a return object. In this case the “this” pointer is transferred in r1.

Rationale

It is much more efficient to transfer parameters in registers than on the stack. The present proposal allows up to 32 parameters, including variable length vectors, to be transferred in registers, leaving 15 general purpose registers and 16 vector registers for the function to use for other pur-

poses while handling the parameters. This will cover almost all practical cases, so that parameters only rarely need to be stored in memory.

Nevertheless, we must have precise rules for covering an unlimited number of parameters if the programming language has no limit to the number of function parameters. We are putting any extra parameters in a list rather than on the stack as most other systems do. The main reason for this is to make the software independent of whether there is a separate call stack or the same stack is used for return addresses and local variables. The addresses of parameters on the stack would depend on whether there is a return address on the same stack. The list method has further advantages. There will be no disagreement over the order of parameters on the stack and whether the stack should be cleaned up by the caller or the callee. The list can be reused by the caller for multiple calls if the parameters are constant, and the called function can reuse a variable argument list by forwarding it to another function. The function is guaranteed to return properly without messing up the stack even if caller and callee disagree on the number of parameters. Tail calls are possible in all cases regardless of the number and types of parameters.

12.5 Register usage convention

Most systems have rules that certain registers have callee-save status. This means that a function must save these registers and restore them before it returns, if they are used. The caller can then rely on these registers being unchanged after the function call.

Current systems have a problem with assigning callee-save status to vector registers. Future CPU versions may make the vector registers longer, and the instructions for saving the longer registers have not been defined yet. Some systems now have callee-save status on part of a vector register because of poor foresight. It is impossible in current systems to save a vector register in a way that will be compatible with future extensions.

This problem is solved by the ForwardCom design with variable vector length. It is possible to save and restore a vector register of any length, even if this length was not supported at the time the code was compiled. It is also possible to know how much of a long vector register is actually used, because the length of a vector is saved in the register itself, so that we only need to save the part of the register that is actually used. The push and pop instructions are designed for this purpose (see page 62). Unused vector registers will use only little space for saving.

It still takes a lot of cache space to save the vector registers if they are long. Therefore, we want to minimize the need for saving registers. It is proposed to have two different methods to choose between. These methods are explained here.

Method 1

This is the default method which can be used in all cases, but not the most efficient method.

The rule is simply that registers r16 – r31 and v16 – v31 have callee-save status.

A function can use registers r0 – r15 and v0 – v15 freely. Sixteen registers of each type will be sufficient for most functions. If the function needs additional registers, it must save them.

All system registers and special registers have callee-save status, except in functions that are intended for manipulating these registers.

Method 2

It will be more efficient if we actually know which registers are used by each function. If function A calls function B, and A knows which registers are used by B, then A can simply choose some registers that are not used by B for any data that it needs to save across the call to B. Even a

long chain of nested function calls can avoid the need to save any registers as long as there are enough registers.

If function A and B are compiled together in the same process then the compiler can easily manage this information. But if A and B are compiled separately, then we need to store the necessary information about which registers are used. This is possible with the object file format described on page 139. The information about register use must be saved in the compiled object file or library file, not in some other file that could possibly come out of sync.

Function B is preferably compiled first into an object file. This object file must contain information about which registers are modified by function B. The necessary information is simply a 64-bit number with one bit for each register that is modified (bit 0-31 for r0-r31, and bit 32-63 for v0-v31). Any registers used for parameters and return value are also marked if they are modified by the function.

When function A is compiled next, the compiler can look in the object file for B to see which registers it modifies. The compiler can take advantage of this information and choose some registers not modified by B for data that need to be saved across the call to B. Registers that are modified by B can be used in A for temporary variables that do not need to be saved across the call to B. Likewise, it will be advantageous to use the same register for multiple temporary variables if their live ranges do not overlap, in order to modify as few registers as possible. The object file for A will contain a list of registers modified by A, including all registers modified by B and by any other functions that A may call. The object file for A contains a reference to function B. This reference must contain information about which registers A expects B to modify. If B is later recompiled, and the new version of B modifies more registers, then the linker will detect the discrepancy and prompt for a recompilation of A.

If, for some reason, A is compiled before B or no information is available about B when A is compiled, then the compiler will have to make assumptions about the register use of B. The default assumption is as specified in method 1. Function A may later be recompiled if B violates these assumptions, or simply to improve efficiency.

If two functions A and B are mutually calling each other then the easiest solution is to rely on method 1. The functions should still include the information about register use in their object files.

The compiler should preferentially allocate the lower registers first in order to minimize the problem that different library functions use different registers. It may skip r6 and v6 for the caller to use for masks.

The main program function is allowed to use method 2 and to modify all registers if it includes the necessary information in its object file.

Object files that are contained in a function library must include the information about register use.

System functions and device drivers cannot be accessed in the same way as normal library functions (see page 126). System functions must obey the rules for method 1, but the system should provide a method for getting information about the register use of each system function. This can be useful for just-in-time compilers.

12.6 Name mangling for function overloading

Programming languages that support function overloading use internal names with prefixes and suffixes on the function names in order to distinguish between functions with the same name but different parameters or different classes or namespaces. Many different name mangling schemes are in use, and some are undocumented. It is necessary to standardize the name man-

gling scheme in order to make it possible to mix different compilers or different programming languages.

The most common name mangling schemes are Microsoft and Gnu. The Microsoft scheme uses characters that cannot occur in function names (?@$). This prevents name clashes, but makes it impossible to call the mangled name directly or to translate e. g. C++ to C. The Gnu scheme generates mangled names that look unwieldy, but contain no special characters that prevent calling the mangled name directly. Therefore, the proposal is to use the Gnu mangling scheme (version 4 or later) with necessary additions for variable-length vectors, etc.

Functions with mangled names may optionally supplement the mangled name with the simple (non-mangled) name as a weak public alias in the object file. This makes it easier to call the function from other programming languages without name mangling. The weak linking of the alias prevents the linker from making error messages for duplicate names.

The compiler must prefix an underscore to all symbol names for languages that do not use name mangling, such as C, in order to avoid name clashes with reserved names in the assembler.

12.7 Binary format for object files and executable files

The ELF file format is used for ForwardCom because it is the most flexible and well-structured format in common use.

The details of the ELF format for ForwardCom are specified in a file named elf_forwardcom.h. The official specification of the ForwardCom ELF file format resides in this file and nowhere else. This specification includes details for section types, symbol types, relocation types, etc. Additional information about event handlers (see page 169), register use (see page 137) and stack use (see page 141) is included in the file format.

File names must have extensions that indicate their type. The following extensions are preferred: Assembly code: .as, object file: .ob, library file: .li, executable file: .ex.

12.8 Function libraries and link methods

Dynamic link libraries (DLLs) and shared objects (SOs) are not used in the ForwardCom system. Instead, we will use only one type of function libraries that can be used in several different ways:

1. Static linking. The linker finds the required functions in the library and copies them into the executable file. Only the parts of the library that are actually needed by the specific main program are included. This is the normal way that static libraries are used in current systems (.lib files in Windows, .a files in Unix-like systems such as Linux, BSD, and Mac OS).

2. Re-linking. The library can be linked or re-linked into the executable during the installation process. There may be a selection of different libraries for different platforms. Third party libraries may also be added in this way. The library may be updated at any time, if needed, by the re-linking method without updating the main program.

3. Run-time linking. The required function is extracted from the library and loaded into memory, preferably at a memory space reserved for this purpose by the main program. Any reference from the newly loaded function to other functions, whether already loaded or not, can be resolved in the same way as for static linking.

These methods will improve the performance and remedy many of the problems that we encounter with the traditional DLLs and SOs. It also helps solve the frequent problems with missing or incompatible library versions.

A typical program in Windows and Unix systems will require several DLLs or SOs when it is loaded. These dynamic libraries will all be loaded into each their memory block, using an integral number of memory pages each, and possibly scattered over the memory space. This leads to a waste of memory space and poor caching. A further performance disadvantage with shared objects is that they use procedure linkage tables (PLT) and global offset tables (GOT) for all accesses to public functions and variables in order to support the rarely used feature of symbol interposition. This requires a lookup in the PLT or GOT for every access to a function or variable in the library, including internal references to globally visible symbols.

The ForwardCom system replaces the traditional dynamic linking with method 2 or 3 above, which will make the code just as efficient as with static linking because the library sections are contiguous with the main program sections, and all access is immediate with no intermediate tables. The time required to load the library will be similar to the time required for dynamic linking because the bottleneck will be disk access, not calculation of function addresses.

A traditional DLL or SO can share its code section (but not its data section) between multiple running programs that use the same library. A ForwardCom library can share its code section between multiple running instances of the same program, but not between different programs. The amount of memory that is wasted by possibly loading multiple instances of the same library code is more than compensated for by the fact that we are loading only the part of the library that is actually needed and that the library does not require its own memory pages. It is not uncommon in Windows and Unix systems to load a dynamic library of one megabyte and use only one kilobyte of it.

These linking methods are efficient in the ForwardCom system because of the way relative addresses are used. The main program typically contains a CONST section immediately followed by a CODE section. The CONST section is addressed relative to the instruction pointer so that these two sections can be placed anywhere in memory as long as they have the same position relative to each other. Now, we can place the CONST section of the library function before the CONST section of the main program, and the CODE section of the library function after the CODE section of the main program. We don't have to change any cross-references in the main program. Only cross references between the main program and the library function and between the CODE and CONST sections of the library function have to be calculated by the linking or relinking process and inserted in the code.

A library function does not necessarily have any DATA and BSS sections. In fact, a thread-safe function has little use of static data. However, if the library function has any DATA and BSS sections, then these sections can be placed anywhere within the ± 2GB range of the DATAP pointer. The references in the library function to its static data have to be calculated relative to the point that DATAP points to; but no references to data in the main program have to be modified when a library is added as long as DATAP still points to some point in the DATA or BSS sections of the main program.

The combined main program and library files can now be loaded into any vacant spaces in memory. It will need only three entries in the memory map: (1) the combined CONST sections of library and main program, (2) the combined CODE sections of main program and library functions, and (3) the combined STACK, DATA, BSS, and HEAP of the main program and the library functions.

Run-time linking works slightly differently. The reference from the main program to the library function goes through a function pointer that is provided when the library is loaded. Any references the other way – from the library function to functions or global data in the main program – can be resolved in the same way as for static linking or through pointer parameters to the function. The main program should preferably reserve space for the CONST, CODE and DATA/BSS sections of any libraries that it will load at run time. The sizes of these reserved spaces are provided in the header of the executable file. The loader has considerable freedom to place these sections anywhere it can in the event that the reserved spaces are insufficient. The only require-

ments are that the CONST section of the library function is within a range of ± 2GB of the CODE section of the library, and the DATA and BSS sections of the library are within ± 2GB of DATAP. The library function may be compiled with a compiler option that tells it not to use DATAP. The function will load the absolute address of its DATA section into a general purpose register and access its data with this register as pointer.

12.9 Predicting the stack size

In most cases, it is possible to calculate exactly how much data stack space an application needs. The compiler knows how much stack space it has allocated in each function. We only have to make the compiler save this information. This can be accomplished in the following way. If a function A calls a function B then we want the compiler to save information about the difference between the value of the stack pointer when A is called and the stack pointer when A calls B. These values can then be summed up for the whole chain of nested function calls. If function A can call both function B and function C then each branch of the call tree is analyzed and the value for the branch that uses most stack space is used. If a function is compiled separately into its own object file, then the information must be stored in the object file.

A function can use any amount of memory space below the address pointed to by the data stack pointer (a so-called red zone) if this is included in the stack size reported in the object file, provided that the system has a separate system stack.

The amount of stack space that a function uses will depend on the maximum vector length if full vectors are saved on the stack. All values for required stack space are linear functions of the vector length: Stack_frame_size = Constant + Factor · Max_vector_length. Thus, there are two values to save for each function and branch: Constant and Factor. We need separate calculations for each thread and possibly also information about the number of threads. We need to save separate values for the call stack and the data stack. The size of the call stack does not depend on the maximum vector length.

The linker will add up all this information and store it in the header of the executable file. The maximum vector length is known when the program is loaded, so that the loader can finish the calculations and allocate a stack of the calculated size before the program is loaded. This will prevent stack overflow and fragmentation of the stack memory. Some programs will use as many threads as there are CPU cores for optimal performance. It is not essential, though, to know how many threads will be created because each stack can be placed anywhere in memory if thread memory protection is used (see page 120).

Avoiding virtual address translation

In theory, it is possible to avoid the need for virtual address translation if the following four conditions are met:

- The required stack size can be predicted and sufficient stack space is allocated when a program is loaded and when additional threads are created.
- Static variables are addressed relative to the data section pointer. Multiple running instances of the same program have different values in the data section pointer.
- The heap manager can handle fragmented physical memory in case of heap overflow.
- There is sufficient memory so that no application needs to be swapped to a hard disk.

A possible alternative to calculating the stack space is to measure the actual stack use the first time a program is run, and then rely on statistics to predict the stack use in subsequent runs.

The same method can be used for heap space. This method is simpler, but less reliable. The calculation of stack requirements based on the compiler is sure to cover all branches of a program, while a statistical method will only include branches that have actually been used.

We may implement a hardware register that measures the stack use. This stack-measurement register is updated every time the stack grows. We can reset the stack-measurement register when a program starts and read it when the program finishes. This method can be useful if the program contains recursive function calls. We do not need a hardware register to measure heap size. This information can be retrieved from the heap manager.

These proposals can eliminate or reduce memory fragmentation in many cases so that we only need a small memory map which can be stored on the CPU chip. Each process and each thread will have its own memory map. However, we cannot completely eliminate memory fragmentation and the need for virtual memory translation because of the complications discussed on page 121.

12.10 Exception handling, stack unrolling, and debug information

Executable files must contain information about the stack frame of each function for the sake of exception handling and stack unrolling for programming languages that support structured exception handling. It should also be used for programming languages that do not support structured exception handling in order to facilitate stack tracing by a debugger.

This system should be standardized. It is recommended to use a table-based method that does not require a stack frame register.

Debuggers need information about line numbers, variable names, etc. This information should be included in object files when requested. The debug information may be copied into the executable file or saved in a separate file which is stored together with the executable file. It is yet to be decided which system to use.

12.11 Assembly language syntax

The definition of a new instruction set should include the definition of a standardized assembly language syntax. The syntax should be suitable for human processing, not only for machine processing. We must avoid a situation similar to the x86 environment where many different syntaxes are in use, with different instruction names and different orders of the operands.

ForwardCom is using an assembly language syntax that looks similar to C and Java in order to make it more intelligible to high-level language programmers and to avoid any confusion over what is source and destination of an instruction. The details are described on page 151.

Chapter 13

Binary tools

All the basic development tools for ForwardCom except compilers are combined into a single executable file named **forw** or **forw.exe**.

The tools can be run using the following command lines:

assemble	forw -ass assemblyfile objectfile
disassemble	forw -dis objectfile assemblyfile
link	forw -link exefile objectfiles libraryfiles
relink	forw -relink inputfile outputfile objectfiles libraryfiles
make library	forw -lib libraryfile objectfiles
emulate	forw -emu exefile
debug	forw -emu exefile -list=debugout.txt
dump	forw -dump objectfile
help	forw -help

General options:
The following options can be used with most or all of the commands listed above:

@file	read additional command line options or file names from file.
-ilist=file	alternative instruction list file.
-wdNNN	disable Warning NNN.
-weNNN	treat Warning NNN as Error. -wex: treat all warnings as errors.
-edNNN	disable Error number NNN.
-ewNNN	treat Error number NNN as Warning.

A list of instructions is stored separately in a comma-separated file named **instruction_list.csv**. This file must be present when running the assembler, disassembler, or debugger. The format for the instruction list is defined in table 4.1.

The tools are running in console mode and returning a status value which is nonzero in case of error. This is useful when running the tools from a makefile, shell script, or Windows .bat file.

13.1 Assembler

Introduction

The assembler is using a syntax similar to C and Java in order to make the code intelligible to high-level language programmers. The details are described in the programming manual in chapter 14, page 151.

Command line

The command line for the assembler has the following format:

```
forw -ass [options] assemblyfile objectfile
```

The following options are supported:

-list=name	Make output list file. This is useful for checking the generated code.
-ilist=name	Specify alternative instruction list name.
-b	Make binary listing in -list file.
-O0	Optimization level 0: The assembler finds the smallest possible instruction that fits the specified operands.
-O1	Optimization level 1 (default): An instruction may be replaced by another instruction that does the same thing more efficiently. For example, `float v1 *= 4` can be replaced by `float v1 = mul_2pow(v1, 2)`. A conditional jump statement is merged with a preceding arithmetic instruction when possible. An `if` statement immediately followed by an unconditional jump is converted to a single conditional jump.
-O3	Optimization level 3. Enable optimizations that ignore subnormal numbers. For example, `float v1 *= 0.25` can be replaced by `float v1 = mul_2pow(v1, -2)`.
-debug=1	Include debug information in object file. Currently, debug information includes only label names.
-maxerrors=n	Specify maximum number of error messages from the assembler.
-datasize=n	Specify maximum combined size of writeable static data sections. Static data can be accessed with 16 bit relative addresses if datasize $\leq$ 32000. This makes the code more compact. See page 180.
-codesize=n	Specify maximum combined size of code and read-only sections. Inter-module jumps and call tables can use 16 bit relative addresses when codesize $\leq$ 131000, and 24 bit relative addresses when codesize $\leq$ 33000000. Read-only static data can be accessed with 16 bit relative addresses if codesize $\leq$ 32000. See page 180.

The preferred extension for file names in ForwardCom are .as for assembly files and .ob for object files.

13.2 Disassembler

The disassembler can convert object files and executable files back to assembly language. The command line for the disassembler has the following format:

```
forw -dis inputfile outputfile [options]
```

The following options are supported:

-ilist=name	specify alternative instruction list name.

The disassembler produces output lines that may look like this example:

```
float v1 = add(v2, 2.5) // 002C _ 227_E 08.0 5 01.02.02.02 _ 4100 00
```

The comment is interpreted as follows:

002C	hexadecimal address relative to the beginning of the current section
227_E	il, mode, and mode2 in the E2 format template. The last digit can indicate various kinds of subdivision of the code format
08.0	operation code OP1 and OP2
5	operand type. 5 means float
01.02.02.02	register fields RD, RS, RT, RU
_	the mask field is unused in this case
4100	the 16-bit data field IM2 contains the value 2.5 in half precision. If the value had been 2.6 we would need full precision and another code format
00	the IM3 field is unused in this case

The addresses shown are byte-addresses for data sections, but word-addresses (i.e. divided by 4) for executable code sections.

The disassembler is used internally to generate a list output for the assembler. The assembly listing and the disassembly are very similar. The names of local labels are lost in the disassembly unless the -debug option is used when assembling and linking. The disassembler will assign label names of the form @_001 where original label names are missing or lost.

13.3 Linker

The linker joins object files and library files into an executable file.
The command line for the linker has the following format:

```
forw -link outputfile [options] objectfiles libraryfiles
forw -relink inputfile outputfile [options] objectfiles libraryfiles
```

The preferred extensions for file names in ForwardCom are .ob for object files, li. for library files, and .ex for executable files.

Linking an executable file

The link command will create an executable file and overwrite any existing file with the same name.

The following options are supported:

-a	(Default) Add the following object files or library files to the executable.
-r	The following object files or library files will be added as relinkable modules so that they can be replaced later. The executable file will be relinkable.
-m	Add the module(s) with the following name(s) from the previously specified library, even when these modules are not needed for resolving external references.
-d	Delete the following relinkable modules or libraries from the executable file (when relinking).
-u	Allow the executable file to have unresolved external symbols. These symbols can be added later when relinking the incomplete executable file. The executable file will be relinkable.
-x	Extract relinkable module from executable file. -xall = all relinkable modules.
-map=filename	Make a link map showing addresses and sizes of sections.
-debug	Add debugging information to the executable file. Currently, the only debugging information is label names and section names.
-hex2	Make a file containing the executable code section in hexadecimal format, with 2 32-bit words per line.

This example will create an executable file from the modules file1.ob and file2.ob, and any members of the library library3.li that may be needed:

```
forw -link -debug myprogram.ex file1.ob file2.ob library3.li
```

Making a relinkable executable file

A relinkable executable file is a file where some or all of the object modules and libraries can be replaced later and new modules can be added. A relinkable executable file contains more information about symbol names and cross-references than a non-relinkable. This extra information is used only when relinking, it is not loaded into memory when the program is executed.

Any of the modules that are included in a relinkable executable file can be relinkable or non-relinkable. The relinkable modules can be removed or replaced later. The non-relinkable modules are permanent.

This example produces a relinkable executable file where all of the modules are relinkable:

```
forw -link myprogram.ex -r file1.ob file2.ob library3.li
```

This example produces a relinkable executable file where file1.ob is permanent and the remaining modules are relinkable:

```
forw -link myprogram.ex -a file1.ob -r file2.ob library3.li
```

This example produces a relinkable executable file where all the modules are permanent, but new modules can be added later:

```
forw -link myprogram.ex -a file1.ob file2.ob library3.li -r
```

This example produces a relinkable executable file where file1.ob and file2.ob are permanent, library3.li is relinkable, and module4.ob from library3.li is added explicitly. Note that module4.ob will be lost by subsequent relinking if there is no reference to it because it is stored as a relinkable library module:

```
forw -link myprogram.ex -a file1.ob file2.ob -r library3.li -m module4.
   ob
```

You can list the relinkable modules contained in an executable file with the dump command:

```
forw -dump -m myprogram.ex
```

Relinking an executable file

The relink command will modify an existing relinkable executable file and produce a new relinkable executable file with a different name. The options are the same as above.

This example will relink an executable file, replacing file2.ob by file2b.ob and replacing library3.li by a newer version of this library with the same name:

```
forw -relink myprogram.ex myprogram2.ex -d file2.ob -r file2b.ob
   library3.li
```

Adding a plugin to a relinkable executable file

You can use the relinking feature to add a plugin to a relinkable executable file.

There are two ways in which the program can access the plugin. The first method is to define a weak external function in the main program and make a public function with the same name in the plugin module. Calling this function will have no effect if the plugin module is not present.

The other way is to define an event handler in the plugin module. This event handler will be called if the event occurs, for example when a menu item is clicked.

This example will add a plugin module plugin4.li to a relinkable executable file. The member start.ob in plugin4.li is added explicitly. The remaining members of plugin4.li are added only if there is a reference to them.

```
forw -relink myprogram.ex myprogram2.ex -r plugin4.li -m start.ob
```

Extracting a module from a relinkable executable file

You can extract modules from a relinkable executable file. This feature is used mainly for testing and debugging purposes. The extracted file is not guaranteed to be identical to the original object file.

```
forw -relink myprogram.ex myprogram2.ex -x file1.ob
```

The output file will be written to the current directory and have a name with prefix ”x_”. ”-xall” will extract all relinkable modules. The output executable file (myprogram2.ex in the above example) is specified but not used.

Relinking and library functions

A library function is included in an executable file only if there is a reference to it. The library functions that are included in a relinkable executable file can be reused when relinking. If a later addition or modification to the executable file needs a library function that was not included in the original executable then the library has to be added again during relinking.

A relinkable library function in an executable file will be replaced during relinking if a new module or library contains a function with the same name. A library function that was included as non-relinkable in a relinkable executable file cannot be replaced later.

Relinking and communal sections

Communal sections are described on page 153. Communal sections have certain similarities with library functions.

A communal section will not be included in the executable file if there is no reference to it. You will get a linking error if a later addition or modification to the executable file attempts to reference a symbol in a discarded communal section. There are three ways to fix this problem:

- Make the section not communal
- Make a reference to the symbol in order to make sure it is always included in the executable file
- Include a copy of the communal section in a module added when relinking

A communal section in a relinkable module will retain its properties. A communal section in a non-relinkable module will become non-communal, and any weak symbols in this section will become public and non-weak. If multiple communal sections with the same name are contained in both relinkable and non-relinkable modules then the linker will include only one. A bigger section is chosen before a smaller one. A section in an object file is chosen before a section in a library file. Finally, the one that comes first on the command line is preferred.

Making a hexadecimal file

A hexadecimal file may be needed for coding a loader into ROM memory. The hexadecimal file is generated in the same way as an executable file, using the -hex option. The output file will contain the executable code section in hexadecimal format. The number of 32-bits per line is specified as a suffix, e.g. -hex2 gives 2 words per line. Each line contains a hexadecimal number in big endian format.

13.4 Library manager

A function library is a collection of object files each defining one or more functions that can be called from other modules. The library manager can make a function library and add or remove modules in it.

The command line for the library manager has the following format:

```
forw -lib libraryfile [options] objectfiles
```

The following options are supported:

-a	(Default) Add the following object files to the library. Any existing object file with the same name will be replaced.
-d	Delete the following object files from the library.
-l	List object file members.
-l2	List object file members and their exported symbols.
-l3	List object file members and their exported and imported symbols.
-x	Extract the following object files from the library.
-xall	Extract all object files from the library.

If a library with the specified name already exists, then it will be modified by adding, deleting, or replacing object files. The library will be created if it does not already exists.

The preferred extension for file names in ForwardCom are .ob for object files and .li for library files.

The names of the object files are stored in the library without the file path so that they can be extracted on another computer with a different directory structure. The library cannot contain multiple object files with the same name.

The ForwardCom system has only one type of library files. The same library can be used for static and runtime linking and for relinking of an executable file.

A standard C library is provided with the name libc.li. A library of mathematical functions is provided with the name math.li. A lightweight version of libc.li is provided with the name libc-light.li for the small softcore with limited capabilities and no system calls.

13.5 Emulator and debugger

The emulator can execute a ForwardCom executable file on another platform. It is useful for testing and debugging.

The command line for the emulator has the following format:

```
forw -emu executable_file [options]
```

The following options are supported:

-list=name	Make list file with debug output.
-maxlines=n	Maximum number of instructions in debug output list.
-ilist=name	Specify alternative instruction list name.

The debug list output will show all executed instructions and their results. It is recommended to use the -debug option when assembling and linking in order to preserve label names in the debug output.

Interactive single-step debugging is currently not supported in the emulator.

The current version of the emulator supports all general instructions but only few system instructions. Integers of 8, 16, 32, and 64 bits are supported. Floating point numbers with half, single, and double precision are supported. Quadruple precision is not supported. Only few instructions with 128 bit integers are supported. Most optional features are supported by the emulator, including exception handling, rounding control, and subnormal numbers.

13.6 Dump utililty

The dump utility can show metadata from object files and executable files.

The command line for the dump utility has the following format:

```
forw -dump-options object_file
```

The following options are supported:

f	file header.
h	section headers.
l	link map.
s	symbol table.
n	string table.
r	relocation records.
m	relinkable modules.

13.7 Compiling the forw tools

These tools can be compiled for Windows, Linux, MacOS, and other platforms.

To compile for Windows using Visual Studio, use the project files forw.sln and forw.vcxproj.

To compile for Linux or other platforms using a Gnu or Clang compiler, use Gnu make with the command make -f forw.make

See the file forwardcom_sourcecode_documentation for details.

13.8 Code examples

A collection of code examples are provided in the examples folder. You can try an example by assembling, linking, and emulating it as follows:

```
forw -ass hello.as
forw -link hello.ex hello.ob libc.li
forw -emu hello.ex
```

Chapter 14

Programming manual

This manual is based primarily on assembly language. Instructions for other programming languages should be described in the manuals for the respective compilers.

14.1 Assembly language syntax

Introduction

The ForwardCom assembly language is standardized to avoid the confusion that we have seen with other instruction sets. The basic syntax of an instruction looks like a function call:

```
datatype destination_operand = instruction(source_operands)
```

This syntax leaves no doubt about which operands are source and destination. You can also use common operators, such as + - * / etc. instead of instructions that correspond to these operators.

Branches and loops can use conditional jump instructions or the high-level language keywords: if, else, for, do, while, break, continue.

Before defining the syntax details we will look at a few examples.

The following example shows a function that calculates the factorial of an integer:

Example 14.1.

```
code section execute          // define executable code section

// factorial function calculates n!
// input: r0, output: r0
_factorial function public
if (uint64 r0 <= 20) {        // check for overflow, 64 bit unsigned
   uint64 r1 = 1              // start with 1
   while (uint64 r0 > 1) {    // loop through r0 values
      uint64 r1 *= r0         // multiply all values
      uint64 r0--             // count down to 1
   }
   uint64 r0 = r1             // put result in r0
   return                     // normal return from function
}
int64 r0 = -1                 // overflow. return max unsigned value
return                        // error return
_factorial end                // end of function
```

```
code end                        // end of code section
```

The next example illustrates the use of the efficient vector loop described on page 13. It calculates the polynomial $y = 0.5x^2 - 4x + 1$ for all elements of an array x and stores the results in an array y.

Example 14.2.

```
data section read write datap       // define data section
% arraysize = 100                   // define constant
float x[arraysize], y[arraysize]    // define arrays
data end                            // end of data section

code section execute                // define code section

// This function calculates a polynomial on all elements of an
// array x and stores the results in an array y
_polyn function public
int64 r1 = address([x+arraysize*4]) // end of array x
int64 r2 = address([y+arraysize*4]) // end of array y
int64 r0 = arraysize*4              // array size in bytes = 400

for (float v0 in [r1-r0]) {         // vector loop
   float v0 = [r1-r0, length=r0]    // read x vector
   float v1 = v0 * 0.5              // 0.5 * x
   float v1 -= 4                    // 0.5 * x - 4
   float v0 = v0 * v1 + 1           // (0.5 * x - 4) * x + 1
   float [r2-r0, length=r0] = v0    // save y vector
}
return                              // return from function
_polyn end                          // end of function

code end                            // end of code section
```

While this looks very much like high-level language code, you have to explicitly specify which register to use for each variable, and you cannot put more code on one line than fits into a single instruction. In example 14.2, the line `float v0 = v0 * v1 + 1` is allowed because it fits the `mul_add2` instruction, but we cannot write `float v1 = v0 * 0.5 - 4` because this instruction cannot have two immediate constants. The line `int64 r1 = address([x+arraysize*4])` also fits a single instruction because `arraysize*4` can be calculated at assembly time (it involves only constants) and the result can be added to the relative address of `x` by the linker. The only thing the `address` instruction has to do at runtime is to add a constant to the `datap` pointer.

More code examples are given in chapter 14.3

File format

An assembly file can be in ASCII or UTF-8 format. An UTF-8 byte order mark is allowed at the beginning of the file, but not required.

Whitespace can be spaces or tabs. The use of tabs is discouraged because different editors may have different tabstops.

Linefeeds can be UNIX style (\n), Mac style (\r), or Windows style (\r\n). There is no limit to the line length.

Comments are C style: /* */ or //

Nested comments are allowed. This makes it possible to comment out a piece of code that already contains comments.

Sections

A section containing code or data is defined as follows:

```
name section options
...
name end
```

The following options can be defined for a section. Multiple options are separated by space or comma.

read	Readable data section.
write	Writeable data section.
execute	Executable code section. This does not imply read access. Write access is usually not allowed for executable sections.
ip	Addressed relative to the instruction pointer. This is the default for executable and read-only sections.
datap	Addressed relative to the data pointer. This is the default for writeable data sections.
threadp	Addressed relative to the thread data pointer. The section will have one instance for each thread.
communal	Communal section. Allows identical sections in different modules, where only one of the communal sections with the same name is included by the linker. Unreferenced communal sections may be removed. Public symbols in communal sections must be weak.
uninitialized	Data section containing only zeroes. The data of this section does not take up space in object files and executable files.
exception_hand	Exception handler and stack unroll information.
event_hand	Event handler information, including static constructors and destructors.
debug_info	Debug information.
comment_info	Comments, including copyright and required library names.
align=n	Align the beginning of the section to an address divisible by n, which must be a power of 2. The default alignment for executable sections is 4. The default alignment for a data section is the size of the biggest data type in the section. A higher alignment can be specified with the align directive.

Sections with the same name are placed consecutively in the executable file. They must have the same attributes, except for alignment.

Sections with different names but the same attributes (except for alignment) are placed in alphabetical order in the executable file.

All code must be placed in executable sections. Data can be placed in read-only sections or writeable sections. Read-only sections are used for constants and tables. Function pointers and jump tables are preferably placed in read-only sections for security reasons.
Variables can be placed in writeable data sections, on the stack, or in registers.

Registers

There are 32 general purpose registers r0 - r31, and 32 vector registers of variable length v0 - v31.

r31 is the stack pointer, also called sp. r0 - r27 can be used as general pointers. r28 - r30 can only be used as pointers if there is no offset or a scaled offset that fits into 8 bits. r0 - r30 can be used as array indexes.

r0 - r6 and v0 - v6 can be used as masks. r0 - r30 and v0 - v30 can be used as fallback values.

The use of registers in function calls must obey the function calling conventions described in chapter 12.4 and the regster usage conventions in chapter 12.5.

Names of symbols

The names of data symbols, code labels, functions, etc. are case sensitive. A name can consist of letters a-z, A-Z, numbers 0-9, the special characters _ $ @, and unicode letters. A name cannot begin with a number. There is no limit to the length of a name.

All names are prefixed with an underscore or mangled in some other way when compiling high-level language code. This is to prevent high-level language names from clashing with assembly keywords, register names, etc.

The names of keywords and instructions are not case sensitive. The following are reserved keywords:

```
align
break broadcast
capab0-capab31 case comment_info communal constant continue
datap debug_info do double
else end event_hand exception_hand execute extern
fallback false float float16 float32 float64 float128 for function
if in int int8 int16 int32 int64 int128 ip
length limit
mask
option options
perf0-perf31 pop public push
r0-r31 read
scalar section sp spec0-spec31 string switch sys0-sys31
threadp true
uint8 uint16 uint32 uint64 uint128 uninitialized
v0-v31
weak while write
```

Constant expressions

Integer constants can be expressed in the following ways:

decimal numbers	contains only digits 0-9 Numbers beginning with 0 are interpreted as decimal, not octal.
binary numbers	0b followed by digits 0-1
octal numbers	0O followed by digits 0-7
hexadecimal numbers	0x followed by digits 0-9, a-f
character constants	1-8 ASCII characters enclosed in single quotes ' '. The first character will be contained in the lowest byte of a 64 bit integer. For example 'AB' = 0x4241
imported constant	extern name: constant
difference between addresses	symbol1 - symbol2. The two symbols must have the same base pointer, either ip, datap, threadp, or none.

Integer constants can be combined with all common operators:
+ - * / % & | ^ ~ && || ! << >> < <= > >= == != ?:

The operators have the same order of precedence as in C.

Additional operators not found in C are:
>>> shift right unsigned,
^^ logical exclusive or.

The results are calculated as signed 64-bit integers, except for >>> which is unsigned.

Imported constants and differences between addresses cannot be allowed in general calculations. The only operations allowed for these are addition of a local constant, and division by a power of 2. These calculations are done by the linker except for a difference between two local symbols in the same section.

Floating point constants must contain a dot or an E and at least one digit, for example 1.23E-4

Floating point expressions are calculated with double precision. The following operators can be used:
+ - * / < <= > >= == != ?:

String constants are sequences of ASCII or UTF-8 characters enclosed in " ".
The following escape sequences are recognized: \\ \n \r \t \"
String constants can be concatenated with the + operator like in Java.

Numeric constants can be included in the instruction codes. This will reduce the pressure on the data cache.

Data types

The following data types are used in data definitions and instructions:

int8	8 bit signed integer
uint8	8 bit unsigned integer
int16	16 bit signed integer
uint16	16 bit unsigned integer
int	32 bit signed integer
int32	32 bit signed integer
uint32	32 bit unsigned integer
int64	64 bit signed integer
uint64	64 bit unsigned integer
int128	128 bit signed integer (optional)
uint128	128 bit unsigned integer (optional)
float16	half precision floating point
float	single precision floating point
float32	single precision floating point
double	double precision floating point
float64	double precision floating point
float128	quadruple precision floating point (optional)

A '+' after an integer type indicates that the assembler can use a bigger type if this makes the instruction smaller or more efficient, for example int16+.

Data definitions

Static data can be defined inside a data section. Several different forms are allowed:

```
Assembly style data definition:
    label : datatype value, value, ...
C style data definition:
    datatype name = value, name = value, ...
C style array definition, uninitialized:
    datatype name[number]
C style array definition, initialized:
    datatype name[number] = {value, value, ...}
Assembly style string definition:
    label : int8 ”string”, ”string”, ...
C style string definition:
    int8 name = ”string”
```

A terminating zero after a string must be added explicitly if needed.

Examples:

```
mydata section read write datap
alpha: float 0.1, 0.2, 0.3
int16 beta = 4, gamma = 5
int8 delta[25]
align (8)
int8 epsilon[] = {6, 7, 8, 9}
zeta: int8 "Dear reader", 0
int8 eta = "Nice to meet you\0"
mydata end
```

Function definitions and labels

A function can be defined inside an executable section. It is defined like this:

```
name function attributes
...
name end
```

Attributes can be 'public', 'weak', and 'reguse=value,value'. The function will be local if no attributes are specified and the name does not appear in a public declaration.

Example:

```
mycode section execute
// this function calculates the square of a double
_square function public, reguse = 0, 1
double v0 *= v0
return
_square end
mycode end
```

The calling conventions for functions are defined in chapter 12.4.

Instructions

It is convenient to have only one instruction per line. Multiple instructions on the same line must be separated by semicolon.

Instructions can be defined only inside an executable section. The general form looks like this:

```
label : datatype destination = instruction(source_operands), options
```

For example:

```
A: int32 r1 = add(r2, 18), mask = r3, fallback = r4
```

The destination is a register in most cases. A few instructions allow a memory operand as destination.

The source operands can be registers, memory operands, or immediate constants. No instruction can have more than one memory operand.

The source operands should be written in the following order: registers, memory operand, immediate operand. The assembler will reorder the operands automatically in certain cases. For example, int r2 = 1 + r1 will be coded as int r2 = r1 + 1. Reordering of operands is not always possible. For example, there is no way to code r2 = 8 >> r1 as a single instruction. Vector operands are not reordered automatically because this will give an invalid result in case the vectors have different lengths.

The following options are possible:

```
options = integer constant
mask = register
fallback = register
fallback = 0
```

Options specify various option bits for specific instructions. Only certain instructions can have options.

Instructions can be written in a simpler form with an operator instead of the instruction name if the instruction does the same as the operator. The general form is:

```
label : datatype destination = operand1 operator operand2
```

For example:

```
int r1 = r2 + 18
```

Masks are used for conditional execution. The mask registers may also contain bits specifying various numerical options. See page 27 for details. Almost all multi-format instructions can have a mask. Single-format instructions with A or E templates can also have a mask. Jump instructions cannot have a mask.

The mask register can be r0-r6 or v0-v6. The fallback register can be r0-r30 or v0-v30. The mask and fallback registers must be the same type of register as the destination (general purpose or vector register). The fallback specifies the result when bit 0 of the mask is zero.

The default fallback register is the same as the first source register. A different fallback register is possible only if there is a vacant register field in the code template. The fallback register cannot be different from the first source register in the following cases:

- the instruction has three source operands
- the instruction needs 64 bits for an immediate constant
- the instruction has a memory operand with index
- the instruction has vector registers and a memory operand

The mask and fallback can be indicated as mask = register, and fallback = register, or in a simple way with the ?: operator:

```
datatype destination = mask ? operand1 operator operand2 : fallback
```

For example:

```
int r1 = r3 ? r2 + 18 : r4
```

A memory operand must be enclosed in square brackets. Memory references are always relative to a base pointer. A memory operand can contain:

A base pointer This is a general purpose register or a special pointer (ip, datap, threadp, sp). The base pointer contains a memory address. The base pointer is implicit for data labels in an ip, datap, or threadp section.

A scaled index This is a general purpose register multiplied by a scale factor. The scaled index is added to the base pointer. This is useful for accessing an array element where the base pointer contains the address of the beginning of the array. The scale factor is the same as the operand size in most cases. Instructions with a general purpose register destination can also have a scale factor of 1. Instructions with a vector register destination can also have a scale factor of -1.

An offset This is an integer constant that will be added to the address.

limit=constant This specifies a maximum value for the index. An error trap will be generated if the unsigned index exceeds this value.

Memory operands in vector instructions must have one of the following options:

scalar Only one element is read.

length=register The length of the vector, in bytes, is specified by a general purpose register. The number of vector elements is equal to the value of this register divided by the number of bytes used by each element.

broadcast=register Only one element is read. This element is broadcast to a vector with a total length specified by a general purpose register. The number of identical elements is equal to the value of this register divided by the number of bytes used by one element.

Examples:

```
int32 r1 = r2 + [alpha] // the base pointer is implicit
int32 r1 = r2 + [r3 + 0x10]
int32 r1 = r2 + [r3 + 4*r4, limit = 100]
int32 v1 = v2 + [r3 - r4, length = r4]
float v1 = v2 + [beta, scalar]
float v1 = v2 + [beta, broadcast=r3]
```

Remember that the base pointer, index, and offset of a memory operand must all be inside the square bracket. Otherwise they will be interpreted as separate operands.

The details for each instruction are described in chapter 5.

Unconditional jumps, calls, and returns

Direct unconditional jumps, calls, and returns are coded as follows:

```
jump target
call target
return
```

Example:

```
my_func function public
    nop
    return
my_func end

...

call my_func
```

Indirect jumps and calls

Indirect jumps and calls can use an absolute 64-bit address in a register or memory operand:

```
jump register
call register
jump ([base_register+offset])
call ([base_register+offset])
```

Example:

```
my_func function public
    nop
    return
my_func end

...

int64 r1 = address([my_func])
```

```
call r1
```

The absolute address must be divisible by 4 because code words are 32 bits (4 bytes).

It is often more efficient to use relative addresses rather than absolute addresses for indirect jumps and calls with memory operands. A relative address contains an offset relative to an arbitrary reference point. The relative address needs fewer bits because it contains the difference between the target address and the reference point. All code addresses are divisible by 4 so we can save two more bits by dividing the relative address by 4.

Indirect jumps and calls using a relative pointer can have the following forms:

```
datatype jump_relative (ref_register, [base_register+offset])
datatype call_relative (ref_register, [base_register+offset])
datatype jump_relative (ref_register, [base_register+index_register*scale])
datatype call_relative (ref_register, [base_register+index_register*scale])
```

The datatype specifies the size of the relative pointer stored in memory. This must be a signed integer type. The reference point must be contained in a 64-bit register.

Example:

```
extern function1: function, function2: function

rodata section read ip  // read-only data section
// relative address of function1, scaled by 4
funcpoint1: int16 (function1-reference_point) / 4
// relative address of function2, scaled by 4
funcpoint2: int16 (function2-reference_point) / 4
rodata end

code section execute ip
reference_point:
  // load address of reference_point:
  int64 r1 = address([reference_point])
  int16 call_relative (r1, [funcpoint1])  // call function1
  int16 call_relative (r1, [funcpoint2])  // call function2
code end
```

See page 164 for further explanation of relative pointers.

Conditional jumps and loops

Conditional jumps always involve an arithmetic or logic operation and a jump conditional upon the result:

```
datatype destination = instruction(source_operands), jump_condition target
```

Examples:

```
  int32+ r1 = add(r1, r2), jump_pos L1
  uint64    compare(r2, 5), jump_above L1
  float     test_bit(v1, 31), jump_nzero L1
  L1:
```

Vector registers can be used only when the data type does not fit into a general purpose register. Floating point addition and subtraction cannot be used with conditional jumps.

The most common conditional jumps can be coded more conveniently using the high-level language keywords: if, else, for, do, while, break, continue.

An 'if' statement can have the form:

```
if (datatype register operator operand) {...} else {...}
```

Line breaks are allowed before and after '{' and '}'. The curly brackets cannot be omitted. Comparing a register value with another register or a constant is done with one of the operators:
`<  <=  >  >=  ==  !=`

Unsigned tests can be indicated with an unsigned integer type, for example uint32. Bit tests can be coded with the & operator as described on page 89.

Examples:

```
if (int r1 >= r2) {
  nop      // r1 >= r2
} else {   // r1 < r2
  if (int !(r3 & (1 << 20))) {
    nop    // bit 20 in r3 is not set
  }
}
```

The conditions cannot be combined with && | | etc. because the statement must fit into a single instruction.

The 'while', 'do-while', and 'for' loops are written in the same way as if statements:

```
while (datatype condition) {...}
do {...} while (datatype condition)
for (datatype initialization; condition; increment) {...}
```

Examples:

```
for (int r1 = 1; r1 <= 100; r1++) {
  int64 r2 = 4  // executed 100 times
  while (uint64 r2 > 0) {
    int64 r2--  // executed 400 times
  }
}
do {
  int32 r2 = r1 - 1      // executed once
} while (int32 r2 > r1) // never true
```

The initialization and increment instructions in the 'for' loop can be anything that fits into a single instruction with the specified data type.

ForwardCom has a very efficient way of doing vector loops as explained on page 13. Vector loops are written in the following way:

```
for (datatype vector_register in [end_pointer - index_register]) {...}
```

Example:

```
int64 r1 = address([my_array])       // address of my_array
int64 r2 = my_arraysize * 4          // size of my_array in bytes
int64 r1 += r2                       // point to end of my_array
for (float v1 in [r1 - r2]) {        // loop through my_array
  float v1 = [r1 - r2, length = r2] // read vector
  float v1 *= 2                      // multiply values by 2
```

```
    float [r1 - r2, length = r2] = v1 // store results in my_array
  }
```

This loop will subtract the maximum vector length from r2 for each iteration and continue as long as r2 (interpreted as a signed 64 bit integer) is positive. It will use the maximum vector length as long as r2 is bigger than the maximum vector length. The maximum vector length may depend on the data type. The loop will use the maximum vector length for the data type specified in the for statement (float in this case). It is possible to use more vector registers and more end-of-array pointers than the ones specified in the 'for' statement.

It is recommended to use optimization level 1 or higher when assembling branches and loops.

Boolean operations

Boolean variables are using only bit 0 for indicating false or true, according to the standard defined on page 134. The remaining bits may be used for other purposes.

Boolean variables can be generated with the compare instruction or the bit test instructions. For example:

```
// C code:
// if (r2 > r3) r4 += 5
// Assembly code:
  int r1 = r2 > r3               // true if r2 > r3
  int r4 = r1 ? r4 + 5 : r4      // conditional add
```

The boolean variable r1 is 1 if the condition is true, and 0 if false.

Boolean variables may be combined with the operators & | ^. Boolean variables are negated by XOR'ing with 1:

```
  int r1 = r2 > r3               // true if r2 > r3
  int r4 = r5 != 0               // true if r5 != 0
  int r6 = r1 & r4               // r1 && r4
  int r7 = r6 ^ 1                // r7 = !r6
```

The compare and bit test instructions can have extra boolean operands. An extra boolean operand to a compare instruction can be specified conveniently with the logical operators && || ^^ This makes it possible to improve the above example:

```
  int r1 = r2 > r3               // true if r2 > r3
  int r6 = r5 != 0 && r1         // true if r5 != 0 && r2 > r3
  int r7 = r6 ^ 1                // r7 = !r6
```

It is even possible to add yet another boolean operand to a compare or bit test instruction by using a mask register. This requires manual coding of the option bits:

```
  int r1 = r2 > r3               // true if r2 > r3
  int r4 = r5 < 0                // true if r5 < 0
  int r6 = compare(r7, r8), mask=r1, fallback=r4, options=0b010001
  // r6 is true if r2 > r3 && r5 < 0 && r7 != r8

  int r9 = test_bits_or(r1,5), mask=r2, fallback=r3, options=0b10010
  // r9 is true if (((r1 & 5) != 0) || r3) && !r2
```

See page 65 for details of the compare instruction, and page 83 for the test_bit instructions.

Boolean variables may be used for conditional jump instructions by testing bit 0 of the register:

```
if (int r1 & 1) {
  // do this if boolean r1 is true
}
```

It is possible to combine two boolean variables in a conditional jump instruction with & | ^ operations. Note that this will test all the bits of the result, not just bit 0:

```
int r1 = r1 | r2, jump_nzero Label
```

The condition is defined only by bit zero when a boolean variable is used as a mask. The remaining bits of the mask may be used for specifying various options, such as floating point exception handling. These mask bits are defined on page 28. It may be necessary to insert these option bits before using a boolean variable as mask for floating point operations:

```
float v1 = 1.2
float v2 = 3.4
int32 v3 = v1 < v2            // boolean variable
int32 v4 = make_mask(v3, 0)   // copy option bits from NUMCONTR
int32 v3 |= v4                // combine boolean and option bits
float v5 = v3 ? v1 * v2 : v1 // conditional multiplication
```

Absolute and relative pointers

Absolute addresses of code and data will usually need 64 bits because the program may be loaded at any memory address. 32 bits will suffice if the program is never run on a machine with more than 2 GB of address space.

Relative addresses can be coded with fewer bits because they specify an address relative to some reference point within the code or data sections of the running program. The necessary number of bits can be further reduced by dividing the relative address by a power of 2. If the address of the target as well as the reference point are know to be divisible by, for example, 4 then we can save two bits by dividing the relative address by 4. In this case, we only need 16 bits for the relative pointer if the distance between target and reference point is less than 128 kB and the relative address is divided by 4.

A further advantage of relative addresses is that they are always position-independent, while absolute addresses must be calculated after the program has been loaded at an arbitrary address in RAM.

Absolute addresses are calculated with the address instruction. For example:

Example 14.3.

```
data section datap                  // define data section
float alpha                         // define variable
data end

code section execute                // define code section
int64 r1 = address([alpha])         // absolute address of alpha
float v1 = [r1, scalar]             // load alpha through pointer
code end
```

A variable can be initialized with an absolute address only if the loader supports relocation. The following example shows how to use an absolute address, though this is not recommended:

Example 14.4.

```
data section datap                    // define data section
float alpha                           // define variable
int64 pointer_to_alpha = alpha        // contains address of alpha
data end

code section execute                  // define code section
int64 r1 = [pointer_to_alpha]         // pointer to alpha
float v1 = [r1, scalar]               // load alpha through pointer
code end
```

Here, the address of alpha is inserted into pointer_to_alpha. Note that this makes the code position-dependent. The address of alpha must be calculated by the loader rather than by the linker. Not all platforms support position-dependent code. Therefore, it is preferred to use relative pointers.

A relative pointer contains an address relative to an arbitrary reference point. The reference point must be placed in a section with the same base pointer (IP, DATAP, or THREADP) as the target of the pointer. A relative address can be converted to an absolute address by the instruction sign_extend_add. Example:

Example 14.5.

```
data section datap                    // define data section
float alpha                           // define variable
int16 relative_pointer = (alpha - reference_point) // relative address
float reference_point                 // arbitrary reference point
data end

code section execute                  // define code section
int64 r1 = address ([reference_point]) // address of reference point
// convert relative address to absolute address:
int16 r2 = sign_extend_add(r1, [relative_pointer])
float v1 = [r2, scalar]               // load alpha through pointer
code end
```

In example 14.5, the sign_extend_add instruction will sign-extend the relative pointer to 64 bits and add the address of the reference point to get the address of the variable alpha. Note that the type int16 on the sign_extend_add instruction indicates the size of relative_pointer, while the size of the reference address r1 and the result r2 are both 64 bits.

The limit of the addresses that can be covered by a 16 bits relative address is $\pm 2^{15}$ or $\pm$ 32 kB. This range can be increased by scaling the relative address. If the target and the reference point are both aligned to addresses divisible by 4, then we can divide the relative address by 4 without losing information. The scale factor must be a power of 2. The same code with a relative address scaled by 4 looks like this:

Example 14.6.

```
data section datap                    // define data section
float alpha                           // define variable
// relative address divided by 4:
int16 relative_pointer = (alpha - reference_point) / 4
float reference_point                 // arbitrary reference point
data end
```

```
code section execute                  // define code section
int64 r1 = address ([reference_point]) // address of reference point
// Convert the relative address to absolute address.
// options = 2 is a shift count that shifts the relative address
// two bits to the left, so that is is multiplied by 4:
int16 r2 = sign_extend_add(r1, [relative_pointer]), options = 2
float v1 = [r2, scalar]               // load alpha through pointer
code end
```

This works in the following way. The linker has support for calculating relative addresses and for scaling addresses by a power of 2. The linker calculates $(\text{alpha} - \text{reference\_point})/4$ and inserts the value at relative_pointer. The sign_extend_add instruction loads the relative pointer, sign-extends to 64 bits, shifts it left by 2, which corresponds to multiplying by $2^2 = 4$, and adds the absolute address in r1. The maximum shift count supported by this instruction is usually 3, corresponding to a scale factor of 8. Support for higher shift counts is optional.

Variables in static memory are always aligned to a natural address, i.e. an address divisible by the size of a data element of the specified type. Thus, int8 is not aligned, int16 is aligned by 2, int32 is aligned by 4, int64 is aligned by 8, float is aligned by 4, and double is aligned by 8. The reference point must have at least the same alignment when relative pointers are scaled. The predefined symbols at page 168 may be used as reference points.

Relative pointers are also used for function pointers and jump tables. Relative code pointers are always scaled by 4. The function pointer can be calculated with the sign_extend_add instruction as in example 14.6, but it is easier to use the relative jump or call instruction:

Example 14.7.

```
const section read ip                 // define constant data section
// relative function pointer divided by 4:
int32 function1_pointer = (function1 - reference_point) / 4
const end

code section execute                  // define code section
reference_point:
int64 r1 = address ([reference_point]) // address of reference point
int32 call_relative (r1, [function1_pointer])
...
function1 function public
    nop
    return
function1 end

code end
```

The jump_relative and call_relative instructions work by sign extending the relative address, multiplying by 4, adding the reference point (first parameter), and then jumping or calling to the calculated address. Remember that the operand type on the call_relative instruction must match the size of the relative function pointer, which is int32 in example 14.7.

A switch-case multiway branch can be implemented in a similar same way, using a table of relative addresses. This table may be placed in read-only memory for security reasons.

Example 14.8.

```
/* C code:
int j, x;
switch (j) {
case 1:
   x = 10;  break;
case 2:
   x = 12;  break;
case 5:
   x = 20;  break;
default:
   x = 99;  break;
}
*/
// ForwardCom code:

rodata section read ip
// table of relative addresses, using DEFAULT as reference point
align (4)
jumptable: int16 0             // case 0
int16 (CASE1 - DEFAULT) / 4    // case 1
int16 (CASE2 - DEFAULT) / 4    // case 2
int16 0                        // case 3
int16 0                        // case 4
int16 (CASE5 - DEFAULT) / 4    // case 5
rodata end

code section execute ip
// r0 = j
// r1 = x
// Test if i is outside of the range,
// use unsigned test to avoid testing for r0 < 0
if (uint32 r0 > 5) {
   jump DEFAULT
}
int64 r2 = address([jumptable])
int64 r3 = address([DEFAULT])
// relative jump with r3 = DEFAULT as reference point,
// r2 as table base and r0 as index
int16 jump_relative (r3, [r2 + r0 * 2])

CASE1:
   int32 r1 = 10
   jump FINISH
CASE2:
   int32 r1 = 12
   jump FINISH
CASE5:
   int32 r1 = 20
   jump FINISH
DEFAULT:
   int32 r1 = 99
FINISH:
```

```
code end
```

The operand type for the multiway jump_relative instruction must match the size of the entries in the table of relative jump addresses, which is int16 in example 14.8. The scale factor must also match this size, which is 2 in this case. The size of the table entries must be big enough to contain the distance between the jump target and the reference point, divided by 4, as a signed integer. The address of jumptable is loaded into a register because the jump_relative instruction does not have space for both the address of the jump table and the index.

A multiway call_relative instruction is coded in the same way, using a table of relative function pointers. This can be useful for virtual functions in an object-oriented programming language with polymorphism. An example is provided on page 174

Imports and exports

A function, data symbol, or constant defined in one assembly module can be accessed from another module if it is exported from the first module and imported to the second module. The necessary cross references are calculated and inserted by the linker.

Symbols are imported as follows:

```
    extern symbolname : attributes, symbolname : attributes, ...
```

The following attributes can be specified:

function	the symbol is an executable function
ip	the symbol is a data object addressed relative to the ip pointer
datap	the symbol is a data object addressed relative to the datap pointer
threadp	the symbol is a data object addressed relative to the threadp pointer
constant	the symbol is a constant with no address
read	the symbol is in a readable data section
write	the symbol is in a writeable data section
execute	the symbol is executable code
data type	the data type for the data symbol
weak	weak linking: the symbol will be resolved only if it exists
reguse	register use, indicating which registers are modified by a function

An external symbol must have one, and only one, of the following attributes: function, ip, datap, threadp, constant. The other attributes are optional. Multiple attributes are separated by space or comma.

The reguse option indicates which registers are modified by a function. It it followed by two numbers indicating the use of general purpose registers and vector registers, respectively. Bit number n indicates that register number n is used. For example: reguse=0x1F,1 indicates that general purpose registers r0-r4 and vector register v0 are modified by the function.

A weak external symbol will only be linked if it exists. The linker will not search function libraries to find the weak symbol, but the symbol will be resolved if it exists in a module that is linked in for other reasons. A weak external constant or data symbol that is not resolved will be zero. A weak function that has not been resolved will return zero.

Symbols are exported as follows:

```
    public symbolname : attributes, symbolname : attributes, ...
```

The possible attributes are the same as for external symbols. It may not be necessary to specify all the attributes because the attributes of locally defined symbols are already known.

It is convenient to place extern and public declarations in the beginning of the assembly file. Forward references are allowed.

Weak public symbols work as follows. If more than one module contains a weak public symbol with the same name then the linker will not issue an error message but link the first symbol. It there is one occurrence of a non-weak symbol with this name then the non-weak symbol will be chosen. If there are more than one non-weak public symbol with the same name then the linker will issue an error message.

Special address symbols

The following symbol names are defined by the linker:

Table 14.1: SpecialSymbols

Name	Base	Meaning
__ip_base	ip	The linker will place this symbol where read-only data ends and code begins. It is possible to explicitly place this symbol elsewhere in an ip-addressable section.
__datap_base	datap	The linker will place this symbol where static initialized writeable data ends and uninitialized data begins. It is possible to explicitly place this symbol elsewhere in a datap-addressable section.
__threadp_base	threadp	The linker will place this symbol at the beginning of thread-local data. It is possible to explicitly place this symbol elsewhere in a threadp-addressable section.
__program_entry	ip	This symbol must be specified. It marks the first instructions to execute. This must be a startup code that makes any necessary initialization and then calls _main. The library forwc.li includes a suitable startup code that will be included automatically if __program_entry is not defined elsewhere.
__event_table	ip	Table of event handler records.
__event_table_num	constant	Number of event handler records.

Other directives

Align

The 'align' directive can align data or code:

```
align (number)
```

The number must be a power of 2. The next data or code will be aligned to an address divisible by the specified number. The beginning of the section will automatically be aligned to at least the same size.

Options

The 'options' directive can change certain parameters with effect from the place of the directive. The syntax is:

```
options name = value, name = value, ...
```

The value must be an integer constant or an expression that can be evaluated immediately to an integer constant.

```
options codesize = 0x100000
options datasize = 0x4000
```

These options change the codesize or datasize so that subsequent instructions will use address sizes that fit the specified codesize or datasize as explained on page 180. Setting codesize = 0 or datasize = 0 will reset these parameters to the value specified on the assembler command line, or a default value if no value is specified on the command line.

Combining vectors of different lengths

The length of the destination register is the same as the length of the first vector register source operand when vectors of different lengths are combined. For example:

```
float v1 = v2 - v3              // v1 has length of v2
float v1 = -v3 + v2             // Same, but v1 has length of v3
float v1 = v2 - [r3,length=r4] // Has length of v2, even if memory
                                // operand has a different length
```

Event handlers

You can specify event handlers for all the events defined in the file elf_forwardcom.h. An event handler consists of a function to be called when the event occurs, and a record defining the event. The event record must contain the following fields:

Table 14.2: Event record structure

Name	Size	Meaning
functionPtr	32 bit	Pointer to the event function relative to __ip_base, scaled by 4.
priority	32 bit	If there are more than one handler for the same event then the ones with the highest value of priority will be called first. Normal priority = 0x1000.
key	32 bit	Subdivision of event. This can define a hotkey, menu item, or icon id for user command events.
event	32 bit	Event ID. Possible values are defined in the file elf_forwardcom.h.

The linker will make a table of all the event handler records, sorted by event, key, and priority.

The event function uses register r0 as status. The function should return a value of 1 in r0 in the normal case. The function can return 0 in r0 to bypass further event handlers with lower priority.

r1 can be used for further parameters. r2 can point to a zero-terminated string.

A common use of event handlers it to call initialization functions that must be called before _main (constructors) and cleanup functions that must be called after _main (destructors). The following example shows an event handler for the event EVT_CONSTRUCT (= 1), initializing some data.

Example 14.9.

```
// section for event handler records only
events section read ip event_hand
int32 (init_func - __ip_base) / 4, 0x1000, 0, 1
events end

// data section
data section read write datap
somedata: int64 0
data end

// code section
code section execute ip
init_func function
int64 r1 = 5
int64 [somedata] = r1 // initialize somedata to 5
int64 r0 = 1          // return status = 1
return
init_func end
```

You can activate an event by calling the function _call_event in the library libc.li. This function will search the table of event records for an event with the specified event and key number and call all corresponding event handler functions.

The table of events cannot be changed while the application is running. A module or library that is added by dynamic linking while the application is running cannot add new event handler records.

14.2 Metaprogramming

Metaprogramming means variables and instructions that do not form part of the final executable code but are useful for controlling the assembly process.

Traditional assemblers use macros for this purpose. The syntax for this is often confusing, especially when macros are nested.

Instead, the ForwardCom assembler will implement metaprogramming involving the following features:

- variables that are used only during the assembly process
- include files
- assembly-time branches and loops
- assembly-time functions
- generate a text string and emit this string as assembly code

Metaprogramming commands are indicated by a percent sign at the beginning of the line. At present, only variables are implemented. The other metaprogramming features will be added later.

Metaprogramming variables

Metaprogramming variables are defined and assigned on a separate line in the following way:

```
% name = value
```

Meta-variables are weakly typed. They can have one of the following types:

integer	evaluated as signed 64-bit integer
floating point	evaluated as double precision
string	ASCII or UTF-8 text string
register	the variable can be an alias for any register
memory operand	the variable can be an alias for any memory operand
type name	the variable can be an alias for a type

It makes no difference whether this meta-code is inside a section or not.

Meta-variables can be redefined or reassigned at any time. Meta-code is sequential so that the same variable can have different values at different places in the code.

An integer meta-variable can be exported as a constant with a public declaration if it has only one value. This is the only case where a forward reference to a meta-symbol is allowed.

While general assembly code allows forward references to data and code labels, meta-code cannot in general have forward references.

Example:

```
% A = 5          // meta-variable integer A = 5
% R = r1         // alias for register r1
int32 R = A      // r1 = 5
% A++            // change value of A to 6
int32 R = A      // r1 = 6
```

14.3 Code examples

This section contains examples of assembly code to illustrate the features of the ForwardCom instruction set. The syntax for assembly language is described in chapter 13.1. The function calling conventions are described in chapter 12.4.

Horizontal vector add

ForwardCom has no instruction for adding all elements of a vector because this would be a complex instruction with variable latency depending on the vector length.

The sum of all elements in a vector can be calculated by repeatedly adding the lower half and the upper half of the vector. This method is illustrated by the following example, finding the horizontal sum of a vector of single precision floats.

Example 14.10.

```
v0 = my_vector               // we want the horizontal sum of this
int64 r0 = get_len(v0)       // length of vector in bytes
int64 r0 = roundp2(r0, 1)    // round up to nearest power of 2
float v0 = set_len(v0, r0)   // adjust vector length
while (uint64 r0 > 4) {      // loop to calculate horizontal sum
   uint64 r0 >>= 1           // the vector length is halved
```

```
   float v1 = shift_reduce(v0,r0) // get upper half of vector
   // the result vector has the length of the first operand:
   float v0 = v1 + v0       // Add upper half and lower half
}
// The sum is now a scalar in v0
```

Horizontal vector minimum

The same method can be used for other horizontal operations. It may cause problems that the set_len instruction inserts elements of zero if the vector length is not a power of 2. Special care is needed if the operation does not allow extra elements of zero, for example if the operation involves multiplication or finding the minimum element. A possible solution is to mask off the unused elements in the first iteration. The following example finds the smallest element in a vector of floating point numbers:

Example 14.11.

```
v0 = my_vector             // find the smallest element in this
r0 = get_len(v0)           // length of vector in bytes
int64 r1 = roundp2(r0, 1)  // round up to nearest power of 2
uint64 r1 >>= 1            // half length
v1 = shift_reduce(v0, r1)  // upper part of vector
int64 r2 = r0 - r1         // length of v1
float v0 = set_len(v0, r1) // reduce length of v0
// make mask for length of v1 because the two operands may
// have different length
int64 v2 = mask_length(v0, r2, 0), options=4
// Get minimum. Elements of v0 fall through where v1 is empty
float v0 = min(v0, v1, mask=v2, fallback=v0) // minimum
// loop to calculate the rest. vector length is now a power of 2
while (uint64 r1 > 4) {
   // Half vector length
   uint64 r1 >>= 1
   // Get upper half of vector
   float v1 = shift_reduce(v0, r1)
   // Get minimum of upper half and lower half
   float v0 = min(v1, v0)  // has the length of the first operand
}
// The minimum is now a scalar in v0
```

Boolean operations

Boolean combinations of conditions can be implemented with branches as shown in this example.

Example 14.12.

```
/* C code:
float condfunc (float a, float b) {
   if (a >= 0 && (a < 20 || a == b)) {
      a = sqrt(a);
```

```
   }
   return a;
}
*/

// ForwardCom code:

code section execute ip
extern _sqrtf : function

// v0 = a, v1 = b
_condfunc1 function public
if (float v0 >= 0) {
   if (float v0 < 20) {jump L1}
   if (float v0 == v1) {
      L1:
      call _sqrtf
   }
}
return  // return value is in v0
_condfunc1 end

code end
```

Branches can be quite slow, especially if they are poorly predictable. It is often faster to generate boolean variables for each condition and use bit operations to combine them. This corresponds to replacing && with & and || with |. The code below shows the same example where three conditional jumps are reduced to one conditional jump and two bit operations. Note that this transformation is not valid if the evaluation of unused conditions has side effects.

Example 14.13.

```
_condfunc2 function public
float v2 = v0 >= 0      // boolean variable for a >= 0
float v3 = v0 < 20      // boolean variable for a < 20
float v4 = v0 == v1     // boolean variable for a == b
int32+ v3 |= v4         // a < 20 || a == b
int32+ v2 &= v3         // a >= 0 && (a < 20 || a == b)
if (float v2 & 1) {     // test bit 0 of boolean v2
   call _sqrtf
}
return
_condfunc2 end
```

We can reduce the number of instructions and make the code still faster by using a special feature of the compare instruction that uses the mask and fallback registers as extra boolean operands:

Example 14.14.

```
_condfunc3 function public
float v2 = v0 >= 0      // boolean variable for a >= 0
float v3 = v0 == v1     // boolean variable for a == b
// use mask and fallback register as extra boolean operands
```

```
float v4 = compare(v0, 20), options=0x22, mask=v2, fallback=v3
if (float v4 & 1) {    // test bit 0 of boolean v4
   call _sqrtf
}
return
_condfunc3 end
```

The high level operators && || ^^ allow a more intuitive coding:

Example 14.15.

```
_condfunc4 function public
float v3 = v0 < 20          // boolean variable for a < 20
float v3 = (v0 == v1) || v3 //  a == b || a < 20
float v3 = v0 >= 0 && v3    // a >= 0 && (a == b || a < 20)
if (float v3 & 1) {         // test bit 0 of boolean v3
   call _sqrtf
}
return
_condfunc4 end
```

Virtual functions

Virtual functions are used in C++ for polymorphous classes. This example shows how to implement a virtual class in ForwardCom. We can save space by using 32-bit relative pointers rather than 64-bit absolute pointers as other systems do.

Example 14.16.

```
/* C++ code:

class VirtClass {
public:
   // constructor:
   VirtClass() {x = 0;}
   // virtual functions:
   virtual void func1() {x++;}
   virtual int  func2() {return x;}
protected:
   int x;
};

int test() {
   VirtClass obj;      // create object
   obj.func1();        // call virtual function 1
   return obj.func2(); // call virtual function 2
}
*/

// ForwardCom code:
rodata section read ip align = 4
// table of virtual function pointers for VirtClass
```

```
// with REFPOINT as reference point
VirtClasstable:
int32 (VirtClass_func1 - REFPOINT) / 4
int32 (VirtClass_func2 - REFPOINT) / 4
rodata end

code section execute ip
// choose any reference point for the relative pointers,
// for example the beginning of the code section:
REFPOINT: nop

VirtClass_constructor function public
// The pointer 'this' is in r0
// At [r0] is a relative pointer to VirtClasstable,
// next the class data members, in this case: x
int32 r1 = VirtClasstable - REFPOINT
int32 [r0] = r1
int32 [r0+4] = 0
return
VirtClass_constructor end

VirtClass_func1 function
// The pointer 'this' is in r0
// x is in [r0+4]
int32 r1 = [r0+4]
int32 r1++
int32 [r0+4] = r1
return
VirtClass_func1 end

VirtClass_func2 function
int32 r0 = [r0+4]
return
VirtClass_func2 end

_test function public
push (r16)       // save register
// get the address of the reference point
int64 r16 = address([REFPOINT])
// the object 'obj' uses 8 bytes, allocate space on the
// stack for it
int64 sp -= 8
// call the constructor. The 'this' pointer must be in r0
int64 r0 = sp
call VirtClass_constructor
// r0 still points to the object because a constructor
// always returns a reference to the object.
// Get the address of the virtual table
int32 r1 = sign_extend_add(r16, [r0])
// call VirtClass_func1 as the first table entry
int32 call_relative (r16, [r1])
// get a pointer to the object again
int64 r0 = sp
```

```
// Get the address of the virtual table
int32 r1 = sign_extend_add(r16, [r0])
// call VirtClass_func2 as the second table entry
int32 call_relative (r16, [r1+4])
// release space allocated for 'obj'
int64 sp += 8
pop (r16)        // restore register
// the return value from callVirtClass_func2 is in r0
return
_test end

code end
```

High precision arithmetic

Function libraries for high precision arithmetic typically use a long sequence of add-with-carry instructions for adding integers with a very large number of bits. A more efficient method for big number calculation is to use vector addition and a carry-look-ahead method. The following algorithm calculates A + B, where A and B are big integers represented as two vectors of n·64 bits each, where n < 64.

Example 14.17.

```
uint64 v0 = A                // first vector,  n*64 bits
uint64 v1 = B                // second vector, n*64 bits
uint64 v2 = carry_in         // single bit in vector register
uint64 v0 += v1              // sum without intermediate carries
uint64 v3 = v0 < v1          // carry generate = (SUM < B)
uint64 v4 = v0 == -1         // carry propagate = (SUM == -1)
uint64 r0 = get_len(v0)      // length of vector in bytes
uint64 v3 = bool2bits(v3)    // compressed to bitfield
uint64 v4 = bool2bits(v4)    // compressed to bitfield
// calculate propagated additional carry:
// CA = CP ^ (CP + (CG<<1) + CIN)
uint64 v3 <<= 1              // shift left carry generate
uint64 v2 = v2 + v3 + v4
uint64 v2 ^= v4
uint64 v1 = bits2bool(r0,v2) // expand additional carry to vector
uint64 v0 += v1              // add correction to sum
uint64 r0 >>= 3              // n = number of elements in vectors
uint64 v3 = gp2vec(r0)       // copy to vector register
uint64 v2 >>= v3             // carry out
// v0 = sum, v2 = carry out
```

If the numbers A and B are longer than the maximum vector length then the algorithm is repeated. If the vector length is more than 64 * 8 bytes then the calculation of the additional carry involves more than 64 bits, which again requires a big number algorithm.

Matrix multiplication

Matrix operations can be difficult because they involve a lot of permutations. The following example shows the multiplication of two 4×4 matrixes of floating point numbers, assuming that the vector registers are long enough to contain an entire matrix.

Example 14.18.

```
float v1 = first_matrix          // first matrix,  4x4 floats
float v2 = second_matrix         // second matrix, 4x4 floats
int64 r1 = 64                    // size of entire matrix in bytes
int64 r2 = 1                     // shift count, elements
int64 r3 = 4                     // shift count, elements
float v0 = replace(v1,0)         // make a matrix of zeroes
for (int64 r0 = 0; r0 < 4; r0++) { // repeat 4 times
   float v3 = repeat_within_blocks(v1,r1,16) // repeat column
   float v4 = repeat_block(v2,r1,16)         // repeat row
   float v0 = v0 + v3 * v4       // multiply rows and columns
   float v1 = shift_down(v1, r2) // next column
   float v2 = shift_down(v2, r3) // next row
}
// Result is in v0.
```

You may roll out the loop and calculate partial sums separately to reduce the loop-carried dependency chain of v0

14.4 Detecting support for particular instructions

The capabilities registers can give information about the capabilities of the processor the code is running on, including support for certain instructions and features, and maximum vector length. See page 96 for details.

While ForwardCom is in a phase of experimental development, there may not be specific bits in the capabilities registers for every instruction that may be supported. An alternative way of testing whether a particular instruction is supported is to disable error trapping and try to execute the instruction. This can be done without system access. For example:

Example 14.19.

```
// Disable error traps for unknown instructions and wrong operands
int r0 = 3
int64 capab2 = write_capabilities(r0, 0)
// Reset counter registers
int r0 = read_perfs(perf16, 0)
// Try to execute instruction
int r0 = userdef56(r0, r0)
// Read counters
int r1 = read_perfs(perf16, 1)   // counter for unknown instruction
int r2 = read_perfs(perf16, 2)   // counter for wrong operands
// Enable error traps again
int r0 = 0;
int64 capab2 = write_capabilities(r0, 0)
// Test if any of the two counters is nonzero, and
// jump to some code if the instruction is not supported
int r1 = r1 | r2, jump_nzero INSTRUCTION_NOT_SUPPORTED
```

14.5 Optimization of code

The ForwardCom system is designed with high performance as a top priority. The following guidelines may help programmers and compiler makers obtain optimal performance.

Use general purpose registers for control code

Any variables that control program execution, such as branch conditions and loop counters, should preferably be stored in general purpose registers rather than memory or vector registers. This may enable the microprocessor to resolve branches early and prefetch the code after the branch earlier.

Efficient loops

The overhead of a loop can be reduced to a single instruction in most cases. A loop that counts up is most efficient when the loop counter is a 32-bit signed integer incremented by 1 and the loop condition is expressed as counter < limit or counter <= limit. A loop that counts down is most efficient when the condition is expressed as counter > 0 or counter >= 0. Examples:

```
for (int32 r1 = 0; r1 < r2; r1++) {  }
for (int32 r1 = 1; r1 <= 100; r1++) {  }
for (int32 r1 = 100; r1 > 0; r1--) {  }
```

The assembler will not insert an initial check before the loop if the start and end values are both constants.

Array loops are particularly efficient if the vector loop feature is used. See the example page 152. Loops containing function calls can be vectorized if the functions allow vector parameters.

Avoid long dependency chains

ForwardCom may be implemented on a superscalar processor that can execute multiple instructions simultaneously. This works most efficiently if the code does not contain long dependency chains.

Minimize instruction size

The assembler will automatically pick the smallest possible version of an instruction. Instructions can have different versions with 8-bit, 16-bit, 32-bit, and 64-bit integer constants. A large integer constant with few significant bits can be represented as a smaller constant with a left shift. For example, the constant 0x5000000000 can be represented as $5 << 36$. The assembler will do this automatically when possible.

Small floating point constants with no decimals can be represented as 8-bit signed integers. Simple rational numbers where the denominator is a power of 2 can be represented as half precision without loss of precision. The assembler does this automatically, too. For example, the constant 2.25 can be coded in half precision without loss of precision, while the constant 2.24 can not.

Memory addresses can use an offset of 8, 16, or 32 bits relative to a base pointer. A 32-bit offset is needed for data in static memory when the data size is large or not specified (see page 180). A smaller offset is possible when data are addressed relative to a stack pointer, structure pointer, class pointer, or a pointer to a strategically placed reference point. An smaller offset will often allow the assembler to use a smaller instruction format.

It is recommended to check the output listing from the assembler to see how much space each instruction takes. Sometimes, you can reduce the instruction size by simple changes in the code. Many instructions will be smaller if the first source register is the same as the destination register.

Optimize cache use

Memory and cache throughput is often a bottleneck. You can improve caching in several ways:

- Optimize code caching by minimizing instruction sizes and inlining functions.
- Optimize data caching by embedding immediate constants in instructions.
- Use register variables instead of saving variables in memory.
- Avoid spilling registers to memory by using the information about register use in object files, as described on page 137.
- Use relative pointers rather than absolute pointers for pointer tables and function pointers in static memory (see page 164).

Calculate pointers early

Registers used for pointers, array index, or for specifying the vector length of a memory operand are used at an early stage in the pipeline. The CPU may have to wait for these registers if they are not available at the time they are needed. Therefore, it is recommended to calculate pointer, array index, and vector length before the other instruction operands.

Optimize jumps

The assembler will merge a jump with a preceding arithmetic instruction if possible, unless optimization is turned off. For example, an integer addition followed by a conditional jump that compares the result with zero may be merged into a single instruction. This is only possible when a number of conditions are fulfilled:

- the two instructions have the same data type
- the destination of the arithmetic instruction is the same register as the source of the branch instruction
- the arithmetic instruction uses the same register for source and destination
- there are no other instructions between the two, and no jump label at the branch instruction.

The output listing will show if the two instructions have been merged.

Optimization of chained jumps, etc., is generally the responsibility of the compiler. The assembler will do only a few simple jump optimizations.

Avoid conditional jumps

Conditional and indirect jumps are quite slow. Conditional jumps can sometimes be replaced by conditional execution of instructions with the use of mask registers. This can be advantageous even if it requires a few more instructions. It is good to calculate the mask register before the operands are calculated because some ForwardCom implementations will not wait for delayed operands if the mask is already known to be zero so that the operands are not needed.

Specify data size and code size

It is recommended to specify a maximum size for code and static data on the assembler command line (see page 144) or in a directive (see page 168). This allows the assembler to optimize relative addresses for both code and data to the minimum number of bits necessary. You may use a link map to see how much memory is needed for code and static data, and add some extra for future additions to the code. A link map can be generated during the link process with the link option `-map=filename`, or after linking with the command `forw -dump-L filename.ex`

Direct jump and call instructions use 24 or 32 bits for relative addresses. Conditional jumps use 8, 16, 24, or 32 bits. The table below shows the largest distance you can jump with these numbers of bits, using signed relative offsets scaled by 4:

Bits	Jump distance
8	508 bytes
16	128 kbytes
24	32 Mbytes
32	8 Gbytes

For example, if you specify codesize=100000, then you will be able to make conditional jumps to external labels using 16 bits for relative addresses. The distance to local labels within the same assembly file and the same section are calculated by the assembler and optimized to the appropriate number of bits regardless of the specified codesize.

Instructions with a memory operand in static memory are using an offset relative to a base pointer, typically DATAP. The offset can be 8, 16, or 32 bits. 8-bit offsets are scaled by the operand size. 16-bit and 32-bit offsets are not scaled. The maximum distance from the base pointer you can address with different sizes of offset are listed in the following table:

Bits	Data type	Addressing range
8	int8	127 bytes
8	int16	254 bytes
8	int32	508 bytes
8	int64	1 kbyte
16	any	32 kbytes
32	any	2 Gbytes

Specifying a maximum datasize on the assembler command line or in a directive will help the assembler select the smallest possible instruction for addressing static data in a writeable data section.

Note that data in a read-only section are usually addressed with IP as base pointer. The number of bits needed for addressing IP-based read-only data is determined by codesize, not datasize.

Conditional jumps can use small single-word instructions if there is no constant immediate operand bigger than 8 bytes and if the distance to the destination is no more than 127 code words (= 508 bytes). This will often suffice for conditional jumps within the same function or module. See page 29 for a list of jump instruction formats with different number of bits for the offset.

Instructions with a memory operand in static data are two or three code words long. The three-word version is needed if the datasize is bigger than 32 kbytes and the instruction has an array index or option bits or three input operands. Example:

Example 14.20.

```
data section read write datap
alpha: int32 123
data end
```

```
code section execute
int32 r0 = r1 < [alpha]
code end
```

The compare instruction in this example will need three words if datasize is bigger than 32 kbytes. A smaller 2-word version of this instruction can be used if datasize is specified to a value less than 32 kbytes.

The assembler will automatically choose the smallest version of an instruction that fits the specified datasize and codesize. The linker will give an “Address overflow” error message if a relative address does not fit the available number of bits.

It is not possible to have memory operands with an 8-bit offset for data with IP, DATAP, or THREADP as implicit base pointer, but it is possible to make the instruction smaller if a general purpose register is used as base pointer. Instructions with a memory operand can use a single-word version of the instruction if the following conditions are satisfied:

- the base pointer is a general purpose register or stack pointer
- the scaled offset fits into an 8-bit signed integer
- the instruction has no more than two source operands
- the first source operand is the same register as the destination
- there is no mask
- there are no option bits
- if vector registers are used: must be scalar

We can make the code more compact by placing a reference point near the data we want to access, and load the address of this reference point into a register. This example shows how:

Example 14.21.

```
data section read write datap
int32 A, B, C
align 8
refPoint: // reference point, aligned by 8
int32 D
double E
data end

code section execute
// load address of refPoint:
int64 r1 = address ([refPoint])
// address of A relative to refPoint:
int32 r2 = r2 + [r1 + A - refPoint]
// address of E relative to refPoint:
double v3 = v3 * [r1 + E - refPoint], scalar

code end
```

The offset relative to the reference point is scaled by 4 and 8 for A and E, respectively, in this example. The scaled offset is calculated automatically by the assembler. The reference point must be aligned by 8 in this example in order to make the offset to E divisible by 8.

This method will make the code more compact if the base pointer is used in more than two instructions. Data on the stack can be accessed with single-word instructions if they are near the address that the stack pointer points to. Data in a structure or class can be accessed in the same way relative to a structure pointer or 'this' pointer.

You can use an assembler listing to check the size and format of each instruction. A table of instruction formats is given on page 20.

Chapter 15

Test suite

A suite of test is provided for testing all instructions and formats.

The test programs are written in ForwardCom assembly for the purpose of self test of emulator and softcores.

The source code for the test programs are available at:
github.com/ForwardCom/test-suite

The following test programs are available so far:

- tests_arithmetics.as: Test all integer arithmetic instructions
- tests_bool_bit.as: Test all boolean and bit manipulation instructions
- tests_branch.as: Test all jump, call, and branch instructions
- test_formats.as: Test all integer instruction formats

Test programs for vector instructions and floating point instructions are not available yet.

The test programs must be assembled and linked with the library libc.li for testing on the emulator or full-featured softcores. Use libc-light.li instead of libc.li for the softcore with limited capabilities and no system calls.

Chapter 16

Softcore

A hardware implementation of ForwardCom as an FPGA softcore is available at github.com/ForwardCom/softcoreA

Features of softcore model A version 1.00

- Runs on Nexys A7-100T FPGA board
- Maximum clock frequency 50 - 70 MHz, depending on configuration
- 32-bit or 64-bit registers
- Can execute one instruction per clock cycle
- Data memory 32 kB. Code memory 64 kB. Call stack 1023 entries.
- Implements all integer instructions, except multiplication, division, push, pop
- Implements all instruction formats and all addressing modes defined by the ForwardCom standard version 1.11.
- No vector registers yet. No floating point instructions
- No system calls, no memory protection. Useful for embedded designs
- Memory reads and writes must be aligned
- RS232 serial interface for standard input and output
- On-chip loader (uses 2 kB code memory)
- On-chip debug interface
- On-chip event counter
- Code examples and test suite provided

Please see the manual for the softcore for details and documentation.

Chapter 17

Conclusion

The ForwardCom instruction set architecture is a consistent, modular, flexible, orthogonal, scalable, and expansible instruction set offering a good compromise between the RISC principle that gives fast decoding and efficient pipelining, and the CISC principle that gives a more compact code and more work done per instruction. Support for efficient out-of-order execution and vector processing is a basic part of the design rather than a suboptimal patch added later as we have seen in other systems.

There are relatively few instructions, but each instruction can be coded in many different variants with integer operands of different sizes and floating point operands of different precisions. The operands can be scalars or vectors of any length. Operands can be registers, immediate constants in various compact forms, or memory operands with different addressing modes. Instructions can have predicates or masks with specified fallback values. Vectors have variable lengths with unlimited room for future expansions.

All in all, the same basic instruction can have many different variants with the same operation code where other instruction sets have many different instructions to cover the same diversity. Everything fits into a consistent template format that simplifies the hardware implementation. The design also has plenty of space for single-format instructions with fewer variants.

The instruction format is designed so that the microprocessor pipeline can be simple and efficient. All instructions fit into the same format templates and the same pipeline structure to facilitate an efficient hardware design. It is possible to make complex instructions with complex functionality, but only if this fits into the pipeline system and the timing constraints. Microcode is preferably not used.

The decoder front-end can load multiple instructions per clock cycle because it is easy to detect the length of each instruction, and the decoder needs only distinguish between a few different instruction sizes. Actually, it is possible to make a working program with only single-word (32 bits) instructions, but it is highly recommended to also support double-word and triple-word instructions.

It is possible to add support for longer instructions in future extensions, but the priority has been to avoid any bottleneck in the decoding of instruction length (which is a serious bottleneck in the x86 architecture).

The code format is designed to be compact in order to save code cache space. This compactness is obtained in several ways. The same instruction can be coded in different sizes with two- and three-operand forms, different sizes of immediate constants, shifted immediate constants, and relative addresses with different sizes of offsets and scale factors, while avoiding absolute addresses that would require 64 bits for the address alone. It is always possible to choose the smallest version of an instruction that fits the particular need. The load on the data cache can be reduced by storing immediate constants in the code rather than in memory operands.

Most instructions can have a mask register which is used for predication in scalar instructions

and for masking in vector instructions. The same mask register is also used for specifying various options such as rounding mode, exception handling, etc., that would otherwise require extra bits in the instruction code.

The introduction of vector registers with variable length is an important improvement over the most common current architectures. The ForwardCom vector system has the following advantages:

- The system is scalable. Different microprocessors can have different maximum vector lengths with no upper limit. It can be used for small embedded systems as well as large supercomputers with very long vectors.
- The same code can run on different microprocessors with different maximum vector lengths and automatically utilize the full vector capabilities of each microprocessor.
- The code does not have to be recompiled when a new microprocessor version with longer vectors becomes available. Software developers do not have to maintain multiple versions of their software for different vector lengths.
- The software can save and restore a vector register in a way that is guaranteed to work with future processors with longer vectors. The inability to do so is a big problem in current architectures.
- Only the part of a vector register that is actually used needs to be saved and restored. Each vector register includes information about how many bytes of it are used. Therefore, no unnecessary resources are wasted on saving a full-length vector if it is unused or only partially used.
- A special addressing mode supports a very efficient loop structure that will automatically use the maximum vector length on all but the last iteration of an array loop. The last iteration will automatically use a shorter vector to handle the remaining array elements in case the array size is not divisible by the maximum vector length. There is no need to handle the remaining elements separately outside the main loop and no need to make separate versions of the loop for different special cases.
- Functions can have variable-length vector registers as parameters. This makes it easy for the compiler to vectorize loops that contain function calls.
- Instructions with vector register operands need no extra information about the vector length because this information is included in the vector registers. This makes these instructions more compact. Instructions with vector memory operands do need this extra information, though.
- The design takes into account the hardware requirements of microprocessors with very long vectors where transport delays across a vector may depend on the vector length.
- A new efficient system with re-linkable function libraries eliminates the need for dynamic link libraries and shared objects.
- Strong security features are built into the design.
- Software standards guarantee compatibility between different compilers, programming languages, user interface frameworks, and operating systems.

The memory model is flexible with relative addresses. Everything is position-independent. Memory management is simpler than in many current systems. There is little or no need for virtual address translation. The preferred implementation has no translation lookaside buffer (TLB) and no memory paging, but a simple on-chip memory map with variable-size memory sections. Problems with stack overflow, memory fragmentation, etc. can be avoided completely in most cases.

Task switches will be fast because of the small memory map and because of the efficient mechanism for saving vector registers.

The principle that a fundamental redesign enables us to learn from history and integrate late additions into the basic design also applies to the whole ecosystem of ABI standard, function libraries, compilers, linkers, and operating system. By defining not only an instruction set, but also ABI standard, binary file format, interface library standard, etc. we get the further advantage that different compilers and different programming languages will be compatible with each other. It will be possible to write different parts of a program in different programming languages and to use the same function libraries with all compilers. Even different operating systems will be compatible to some degree. A feature for re-linking an executable file makes it possible to run the same binary program file in different operating systems or on different platforms where the appropriate user interface framework is selected when the program is installed.

We have also learned from past mistakes that it is difficult to predict future needs. While the ForwardCom instruction set is intended to be flexible with room for future extensions, we may ask whether the future will bring needs for new features that are difficult to integrate into our design and standards. The best way to prevent such unforeseen problems is to allow input and suggestions from the entire community of hardware and software developers. It is important that the design and standards are developed through an open process that allows everybody to comment and make suggestions. We have often seen the problems of leaving this to a commercial industry. The industry often makes short-term decisions for marketing reasons. Patents, license restrictions, and trade secrets harm competition and prevent niche operators from entering the market. The microprocessor industry often keeps new features and instruction set extensions secret for competitive reasons until it is too late to change them in case the IT community comes up with better proposals.

The ForwardCom project is being developed in an open process with contributions and ideas from various people based on the philosophy that the problems mentioned above can best be avoided through openness and collaboration.

Chapter 18

Revision history

Version 1.11, 2021-08-07.

- Templates E2 and E3 are reorganized so that IM2, IM3, and OP2 are adjacent. IM3 can now be extended into OP2 to form an 8-bit immediate operand.
- New instruction formats 2.0.5, 2.2.5, 3.0.5, and 3.2.5 with both memory and immediate operands. Special cases for bit manipulation instructions with both memory and immediate operands are no longer needed.
- New branch instruction formats 2.5.2 and 3.1.2 for conditional jumps with a memory operand. Format 2.5.0 modified to allow three registers. Other jump instruction formats modified to facilitate these changes.
- Instruction formats with memory operands are modified so that the base pointer always uses the RS field while index and vector length always use the RT field. Other instructions with a length parameter are modified to use RT for the length parameter.
- Optional support for push and pop instructions with a sequence of registers in one instruction.
- Performance monitoring counters and error tracking features added.
- New instructions: select_bits, funnel_shift, breakpoint.
- Removed instructions: and_not, and_bit, rotate_up, rotate_down, truth_tab2, replace_bits.
- Renamed instructions: test_bits_and, test_bits_or, test_bits_and/jump_true
- Modified instructions: sign_extend_add, mul_add, mul_add2, truth_tab3, input, output.
- New branch instructions: increment_compare/jump_below, test_bits_or/jump_true, compare_abs/jump_below/above, fp_category/jump_true.
- Removed branch instructions: shift_right/left/jump_zero, rotate/jump_carry, compare/jump_finite
- Modified branch instructions: and, or, xor/jump_zero, test_bit/jump_true.
- Operand size for some instructions with C template changed from int64 to int32.
- Op1 code changed in several instructions.

Version 1.10, 2020-05-21.

- Tiny instructions are dropped. Earlier versions defined a compact instruction format that allowed two tiny instructions to be packed into a single 32-bit code word. However, it was found that most tiny instructions go unpaired so that they use 32 bits anyway. The limited advantage of tiny instructions did not justify the extra complexity. The code space that was occupied by tiny instruction pairs are reused for other purposes.
- Single-word jump instructions are moved from format 1.4 and 1.5 to format 1.6 and 1.7, formerly used for tiny instructions. Format 1.3B and 1.3C is split into 1.3B and 1.4C. Format 1.5 is now vacant for application-specific vector instructions.
- New push, pop, and clear instructions to replace the former tiny instruction pairs with the same functionality. These instructions may generate multiple micro-operations.

Version 1.09, 2020-04-14.

- Methods for detecting floating point and integer numerical errors modified and specified.
- Mask register bits changed to support floating point exception tracking.
- Some instructions with format 1.0, 1.2 and 1.3 modified or moved to different formats.

Version 1.08, 2018-03-30.

- Binary tools now include assembler, disassembler, linker, library manager, emulator, and debugger
- Function libraries and code examples are provided
- ID numbers for system functions, events, and interrupts defined in the file system_functions.h
- Format for library files defined in the file elf_forwardcom.h
- Format for event handlers and other features defined in the file elf_forwardcom.h
- File header modified to remove the limitation of 64000 sections
- Improved NAN propagation made mandatory. Addition of NANs gives the highest payload. Min and max instructions propagate NANs according to the forthcoming revision of the IEEE-754 standard.
- Floating point compare/jump instructions have ordered and unordered versions.
- Better support for half precision floating point vectors.
- Many small changes in instruction list.

Version 1.07, 2017-11-03.

- The first version of binary tools is published, including a high-level assembler and disassembler. A manual is included in chapter 13. The assembly language is modified.
- A feature for re-linking of executable files replaces the previous idea of load-time library dispatching.
- Triple size formats with template E added. Some formats renumbered.
- Some support for half precision floating point vectors.

- Function calling convention allows return using two registers.
- 4-bit constants in tiny format changed from signed to unsigned.
- Many small changes in instruction list.
- Object file format modified to indicate IP, DATAP, or THREADP addressing for sections and symbols

Version 1.06, 2017-02-14.

- Added chapter: Proposal for reducing branch misprediction delay
- Added instruction: increment_jump_sabove.
- Modified various conditional jump instructions. More detailed descriptions.

Version 1.05, 2017-01-22.

- Systematic description of all instructions.
- Instruction list updated.
- Added chapter: Support for multiple instruction sets.
- Added chapter: Software optimization guidelines.
- Bit manipulation instructions improved.
- Shift instructions can multiply float by power of 2.
- Integer division with different rounding modes.
- Source of option bits for mul_add, add_add and compare instructions modified.

Version 1.04, 2016-12-08.

- Instruction formats made more consistent. Template E2 modified.
- Masking principle changed. Fallback value option. r0 and v0 allowed as masks.
- Compare instruction has additional features.
- Conditional jumps modified
- Several other instructions modified.

Version 1.03, 2016-08-01.

- Minor changes and additions to manual.
- Three new instructions added.

Version 1.02, 2016-06-25.

- Name changed to ForwardCom.
- Moved to github.
- Various security features added.
- Support for dual stack.
- Some instruction formats modified, including more formats for jump and call instructions.
- System call, system return and trap instructions added.
- New addressing mode for arrays with bounds checking.
- Several instructions modified or added.
- Memory management and ABI standards described in more detail.
- Instruction list in comma separated file instruction_list.csv.
- Object file format defined in file elf_forwardcom.h

Version 1.01, 2016-05-10.

- The instruction set is given the name CRISC1.
- The length of a vector register is stored in the register itself. The basic code structure is modified as a consequence of this. Function calling conventions are also simplified as a consequence of this.
- All user-level instructions are defined.
- The entire text has been rewritten and updated.

Version 1.00, 2016-03-22.

This document is the result of a long discussion on Agner Fog's blog, starting on 2015-12-27, as well as input from the RISC-V mailing list and the Opencores forum.

Additional inspiration was found in various sources listed on page 10.

Version 1.00 of this manual was published at www.agner.org/optimize.

Chapter 19

Copyright notice