Gisselquist
Technology, LLC

ZIPCPU
SPECIFICATION

Dan Gisselquist, Ph.D.
dgisselq (at) opencores.org

November 4, 2016

Copyright (C) 2016, Gisselquist Technology, LLC

This project is free software (firmware): you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTIBILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <http://www.gnu.org/licenses/> for a copy.

Revision History

Rev.	Date	Author	Description
1.0	11/4/2016	Gisselquist	Major rewrite, includes compiler info
0.91	7/16/2016	Gisselquist	Described three more CC bits
0.9	4/20/2016	Gisselquist	Modified ISA: LDIHI replaced with MPY, MPYU and MPYS replaced with MPYUHI, and MPYSHI respectively. LOCK instruction now permits an intermediate ALU operation.
0.8	1/28/2016	Gisselquist	Reduced complexity early branching
0.7	12/22/2015	Gisselquist	New Instruction Set Architecture
0.6	11/17/2015	Gisselquist	Added graphics to illustrate pipeline discussion.
0.5	9/29/2015	Gisselquist	Added pipelined memory access discussion.
0.4	9/19/2015	Gisselquist	Added DMA controller, improved stall information, and self–assessment info.
0.3	8/22/2015	Gisselquist	First completed draft
0.2	8/19/2015	Gisselquist	Still Draft, more complete
0.1	8/17/2015	Gisselquist	Incomplete First Draft

Contents

Figures

Figure Page

Tables

Table Page

Preface

Many people have asked me why I am building the ZipCPU. ARM processors are good and effective. Xilinx makes and markets Microblaze, Altera Nios, and both have better toolsets than the ZipCPU will ever have. OpenRISC is also available, RISC–V may be replacing it. Why build a new processor?

The easiest, most obvious answer is the simple one: Because I can.

There's more to it though. There's a lot of things that I would like to do with a processor, and I want to be able to do them in a vendor independent fashion. First, I would like to be able to place this processor inside an FPGA. Without paying royalties, ARM is out of the question. I would then like to be able to generate Verilog code, both for the processor and the system it sits within, that can run equivalently on both Xilinx, Altera, and Lattice chips, and that can be easily ported from one manufacturer's chipsets to another. Even more, before purchasing a chip or a board, I would like to know that my soft core works. I would like to build a test bench to test components with, and Verilator is my chosen test bench. This forces me to use all Verilog, and it prevents me from using any proprietary cores. For this reason, Microblaze and Nios are out of the question.

Why not OpenRISC? Because the ZipCPU has different goals. OpenRISC is designed to be a full featured CPU. The ZipCPU was designed to be a simple, resource friendly, CPU. The result is that it is easy to get a ZipCPU program running on bare hardware for a special purpose application–such as what FPGAs were designed for, but getting a full featured Linux distribution running on the ZipCPU may just be beyond my grasp. Further, the OpenRISC ISA is very complex, defining over 200 instructions–even though it has never been fully implemented. The ZipCPU on the other hand has only a small handful of instructions, and all but the Floating Point instructions have already been fully implemented.

My final reason is that I'm building the ZipCPU as a learning experience. The ZipCPU has allowed me to learn a lot about how CPUs work on a very micro level. For the first time, I am beginning to understand many of the Computer Architecture lessons from years ago.

To summarize: Because I can, because it is open source, because it is light weight, and as an exercise in learning.

Dan Gisselquist, Ph.D.

1.

Introduction

The goal of the ZipCPU was to be a very simple CPU. You might think of it as a poor man's alternative to the OpenRISC architecture. You might also think of it as an Open Source microcontroller. For this reason, all instructions have been designed to be as simple as possible, and the base instructions are all designed to be executed in one instruction cycle per instruction, barring pipeline stalls.[1] Indeed, even the bus has been simplified to a constant 32-bit width, with no option for more or less. This has resulted in the choice to drop push and pop instructions, pre-increment and post-decrement addressing modes, the integrated memory management unit (MMU), and more.[2]

For those who like buzz words, the ZipCPU is:

- A 32-bit CPU: All registers are 32-bits, addresses are 32-bits, instructions are 32-bits wide, etc. Indeed, the "byte size" for this processor, as per the C–language definition of a "byte" being the smallest addressable unit, is 32–bits.
- A RISC CPU. There is no microcode for executing instructions. All instructions are designed to be completed in one clock cycle.
- A Load/Store architecture. (Only load and store instructions can access memory.)
- Wishbone compliant. All peripherals are accessed just like memory across this bus.
- A Von-Neumann architecture. The instructions and data share a common bus.
- A pipelined architecture, having stages for **Prefetch**, **Decode**, **Read-Operand**, a combined stage containing the **ALU**, **Memory**, **Divide**, and **Floating Point** units, and then the final **Write-back** stage. See Fig. 1.1 for a diagram of this structure.
- Completely open source, licensed under the GPL.[3]

The ZipCPU also has one very unique feature: the ability to do pipelined loads and stores. This allows the CPU to access on-chip memory at one access per clock, minus any stalls for the initial access.

[1] The exceptions to this rule are the multiply, divide, and load/store instructions. Once the floating point unit is built, I anticipate these will also be exceptions to this rule.

[2] A not–so integrated MMU is currently under development.

[3] Should you need a copy of the ZipCPU licensed under other terms, please contact me.

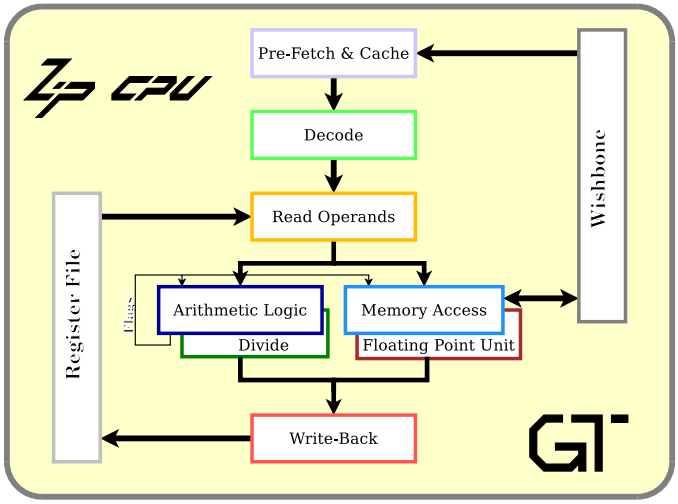

Figure 1.1: ZipCPU internal pipeline architecture

1.1 Characteristics of a SwiC

This section might also be called *the ZipCPU philosophy*. It discusses the basis for the ZipCPU design decisions, and why a low logic count CPU is or can be a good thing.

Many other FPGA processors have been defined to be good Systems on a Chip, or SoC's. The entire goal of such designs, then, is to provide an interface to the processor and its external environment. This is not the case with the ZipCPU. Instead, we shall define a new concept, that of a soft core internal to an FPGA, as a "System within a Chip," or a SwiC. SwiCs have some very unique properties internal to them that have influenced the design of the ZipCPU. Among these are the bus, memory, and available peripherals.

Many other approaches to soft core CPU's employ a Harvard architecture. This allows these other CPU's to have two separate bus structures: one for the program fetch, and the other for the memory. Indeed, Xilinx's proprietary Microblaze processor goes so far as to support four busses: two for cacheable memory, and two for peripherals, with each of those split between instructions and data. The ZipCPU on the other hand is fairly unique in its approach because it uses a Von Neumann architecture, requiring only one bus within any FPGA. This structure was chosen for its simplicity. Having only the one bus helps to minimize real-estate, logic, and the number of wires that need to be passed back and forth, while maintaining a high clock speed. The disadvantage is that both prefetch and memory access units need to contend for time on the same bus.

Soft core's within an FPGA have an additional characteristic regarding memory access: it is slow. While memory on chip may be accessed at a single cycle per access, small FPGA's often have only a limited amount of memory on chip. Going off chip, however, is expensive. Two examples will prove this point. On the XuLA2 board, Flash can be accessed at 128 cycles per 32–bit word, or 64 cycles per subsequent word in a pipelined architecture. Likewise, the SDRAM chip on the XuLA2 board allows a 6 cycle access for a write, 10 cycles per read, and 2 cycles for any subsequent

pipelined access read or write. Either way you look at it, this memory access will be slow and this doesn't account for any logic delays should the bus implementation logic get complicated.

As may be noticed from the above discussion about memory speed, a second characteristic of memory is sequential memory accesses may be optimized for minimal delays (pipelined), and that pipelined memory access is faster than non–pipelined access. Therefore, a SwiC soft core should support pipelined operations, but it should also allow a higher priority subsystem to get access to the bus (no starvation).

As a further characteristic of SwiC memory options, on-chip cache's are expensive. If you want to have a minimum of logic, cache logic may not be the highest on the priority list. Any SwiC capable processor must be able to either be built without caches, or to scale up or down the logic required by any cache.

In sum, memory is slow. While one processor on one FPGA may be able to fill its pipeline, the same processor on another FPGA may struggle to get more than one instruction at a time into the pipeline. Any SwiC must be able to deal with both cases: fast and slow memories.

A final characteristic of SwiC's within FPGA's is the peripherals. Specifically, FPGA's are highly reconfigurable. Soft peripherals can easily be created on chip to support the SwiC if necessary. As an example, a simple 30-bit peripheral could easily support reversing 30-bit numbers: a read from the peripheral returns its bit–reversed address. This is cheap within an FPGA, but expensive in instructions. Reading from another 16–bit peripheral might calculate a sine function, where the 16–bit address internal to the peripheral was the angle of the sine wave.

Indeed, anything that must be done fast within an FPGA is likely to already be done–elsewhere in the fabric. Further, the application designer gets to choose what tasks are so important they need fabric dedicated to them, and which ones can be done more slowly in a CPU. This leaves the CPU with the simple role of solely handling sequential tasks, and tasks that need a lot of state.

This means that the SwiC needs to live within a very unique environment, separate and different from the traditional SoC. That isn't to say that a SwiC cannot be turned into a SoC, just that this SwiC has not been designed for that purpose. Indeed, some of the best examples of the ZipCPU are System on a Chip examples.

1.2 Scope

The ZipCPU is itself nothing more than a CPU that can be placed within a larger design. It is not a System on a Chip, but it can be used to create a system on a chip. As a result, this document will not discuss more than a small handful of CPU–related peripherals, as the actual peripherals used within a design will vary from one design to the next. Further, because control access will vary from one environment to the next, this document will not discuss any host control programs, leaving those to be discussed and defined together with the environments the ZipCPU is placed within.

2.

CPU Architecture

This chapter describes the general architecture of the ZipCPU. It first discusses the configuration options to the CPU and then breaks into two threads. These last two threads are a discussion of the internals of the ZipCPU, such as its instruction set architecture and the details and consequences of it, and then the external architecture describing how the ZipCPU fits into the systems surrounding it, and what those systems must do to support it. Specifically, the external architecture section will discuss both the ZipSystem, the peripherals provided by it, as well as the debug interface.

2.1 Build Options/defines

One problem with a simple goal such as being light on logic, is that some architectures have some needs, others have other needs. What is light logic in some architectures might consume all the available logic in others. As an example, the CMod S6 board built by Digilent uses a very spare Xilinx Spartan 6 LX4 FPGA. This FPGA doesn't have enough look up tables (LUTs) to support pipelined mode, whereas another project running on a XuLA2 LX25 board made by Xess, having a Spartan 6 LX25 on board, has more than enough logic to support a pipelined mode. Very quickly it becomes clear that LUTs can be traded for performance.

To make this possible, the ZipCPU has both a configuration file as well as a set of parameters that it can be built with. Often, those parameters can override the configuration file, but not all configuration file changes can be overridden. Several options are available within the configuration file, such as making the Zip CPU pipelined or not, able to handle a faster clock with more stalls or a slower clock with no stalls, etc.

The `cpudefs.v` file encapsulates those control options. It contains a series of `` `define `` statements that can either be commented or left active. If active, the option is considered to be in effect. The number of LUTs the Zip CPU uses varies dramatically with the options defined in this file. This section will outline the various configuration options captured by this file.

The first couple of options control the Zip CPU instruction set, and how it handles various instructions within the set:

`OPT_MULTIPLY` controls whether or not the multiply is built and included in the ALU by default, and if it is which of several multiply options is selected. Unlike many of the defines that follow within `cpudefs.v` that are either defined or not, this option requires a value. A value of zero means no multiply support, whereas a value of one, two, or three, means that a multiply will be included that takes one, two, or three clock cycles to complete. The option, however, only controls the default value that the `IMPLEMENT_MPY` parameter to the CPU, having the same interpretation, is given. Because this is just the default value, it can easily be overridden upon instantiation. If the

IMPLEMENT_MPY parameter is set to zero, then any attempt to execute a multiply instruction will cause an illegal instruction exception.

OPT_DIVIDE controls whether or not the divide instruction is built and included into the ZipCPU by default. Set this option and the IMPLEMENT_DIVIDE parameter will have a default value of one, meaning that unless it is overridden with zero, the divide unit will be included. If the divide is not included, then any attempt to use a divide instruction will create an illegal instruction exception that will send the CPU into supervisor mode.

OPT_IMPLEMENT_FPU will (one day) control whether or not the floating point unit (once I have one) is built and included into the ZipCPU by default. This option sets the IMPLEMENT_FPU parameter to one, so alternatively it can be set and adjusted upon instantiation. If the floating point unit is not included then, as with the multiply and divide, any floating point instruction will result in an illegal instruction exception that will send the CPU into supervisor mode.

OPT_SINGLE_FETCH controls whether or not the prefetch has a cache, and whether or not it can issue one instruction per clock. When set, the prefetch has no cache, and only one instruction is fetched at any given time. This effectively sets the CPU so that only one instruction is ever in the pipeline at a time, and hence you may think of this as a "no pipeline" option. However, since the pipeline uses so much area on the FPGA, this is an important option to use in trimming down used logic if necessary. Hence, it needs to be maintained for that purpose. Be aware, though, setting this option will disable all pipelining, and therefore will drop your performance by a factor of 8x or even more.

I recommend only defining or enabling this option if you *need* to, such as if area is tight and speed isn't as important. Otherwise, leave the option undefined since the pipelined options have a much better speed performance.

The next several options are pipeline optimization options. They make no sense in a single instruction fetch mode, hence they are all disabled if OPT_SINGLE_FETCH is defined.

OPT_PIPELINED is the natural result and opposite of using the single instruction fetch unit. It is an internal parameter that doesn't need user adjustment, but if you look through the cpudefs.v file you may see and notice it. If you have not set the OPT_SINGLE_FETCH parameter, cpudefs.v will set the OPT_PIPELINED option. This is more for readability than anything else, since OPT_PIPELINED makes more intuitive readability sense than OPT_SINGLE_FETCH. In other words, define or comment out OPT_SINGLE_FETCH, and let OPT_PIPELINED be taken care of automatically.

Assuming you have chosen not to define OPT_SINGLE_FETCH, OPT_TRADITIONAL_PFCACHE allows you to switch between one of two prefetch cache modules. If enabled (recommended), a more traditional cache will be implemented in the CPU. This more traditional cache reduces the stall count tremendously over the alternative pipeline cache, and its LUT usage is quite competitive. As there is little downside to defining this option if pipelining is enabled, I would recommend including it.

The alternative prefetch and cache, sometimes called the pipeline cache, tries to read instructions ahead of where they are needed, while maintaining what it has read in a cache. That cache is cleared anytime you jump outside of its window, and it often competes with the CPU for access to the bus. These two characteristics make this alternative bus often less than optimal.

OPT_EARLY_BRANCHING is an attempt to execute a BRA (branch or jump) statement as early in the pipeline as possible, to avoid as many pipeline stalls on a branch as possible. As an example, if you have OPT_TRADITIONAL_PFCACHE defined as well, then branches within the cache will only cost a single stall cycle. Indeed, using early branching, a BRA instruction can be used as the compiler's branch prediction optimizer: BRA's barely stall, while branches on conditions will always suffer about

6 stall cycles. Setting this option causes the parameter, EARLY_BRANCHING, to be set to one, so it can be overridden upon instantiation.

Given the performance benefits achieved by early branching, setting this flag is highly recommended.

OPT_PIPELINED_BUS_ACCESS controls whether or not LOD/STO instructions can take advantage of the pipelined wishbone bus. To be eligible, the operations to be pipelined must be adjacent, must be all LODs or all STOs, and the addresses must all use the same base address register and either have identical immediate offsets, or immediate offsets that increase by one for each instruction. Further, the LOD/STO string of instructions must all have the same conditional (if any). Currently, this approach and benefit is most effectively used when saving registers to or restoring registers from the stack at the beginning/end of a procedure, when using assembly optimized programs, or when doing a context swap.

I recommend setting this flag, for performance reasons, especially if your wishbone bus implementation can handle pipelined bus accesses. The logic impact of this setting is minimal, the performance impact can be significant.

OPT_VLIW includes within the instruction set the Very Long Instruction Word packing, which packs up to two instructions within each instruction word. Non–packed instructions will still execute as normal, this just enables the decoding and running of packed instructions.

The two next options, INCLUDE_DMA_CONTROLLER and INCLUDE_ACCOUNTING_COUNTERS control whether the DMA controller is included in the ZipSystem, and whether or not the eight accounting timers are also included. Set these to include the respective peripherals, comment them out not to. These only affect the ZipSystem implementation, and not any ZipBones implementations.

Finally, if you find yourself needing to debug the core and specifically needing to get a trace from the core to find out why something specifically failed, you may find it useful to define DEBUG_SCOPE. This will add a 32–bit debug output from the core, as the last argument to the core, to the ZipSystem, or even to ZipBones. The actual definition and composition of this debugging bit–field changes from one implementation to the next, depending upon needs and necessities, so please look at the code at the bottom of zipcpu.v for more details.

That ends our discussion of CPU options, but there remain several implementation parameters that can be defined with the CPU as well. Some of these, such as IMPLEMENT_MPY, IMPLEMENT_DIVIDE, IMPLEMENT_FPU, and EARLY_BRANCHING have already been discussed. The remainder shall be discussed quickly here.

The RESET_ADDRESS parameter controls what address the CPU attempts to fetch its first instruction from upon any CPU reset. The default value is not likely to be particularly useful, so overriding the default is recommended for every implementation.

The ADDRESS_WIDTH parameter can be used to trim down the width of addresses used by the CPU. For example, although the Wishbone Bus definition used by the CPU has 32–address lines, particular implementations may have fewer. By setting this value to the actual number of wires in the address bus, some logic can be spared within the CPU. The default is a 32–bit wide bus.

The LGICACHE parameter specifies the log base two of the instruction cache size. If no instruction cache is used, this option has no effect. Otherwise it sets the size of the instruction cache to be 2^{LGICACHE} words. The traditional prefetch cache, if used, will split this cache size into up to thirty two separate cache lines.

The IMPLEMENT_LOCK parameter controls whether or not the LOCK instruction is implemented. If set to zero, the LOCK instruction will cause an illegal instruction exception, otherwise it will be implemented if pipelining is enabled.

Supervisor Register Set #'s 0-15			User Register Set #'s 16-31	
sR0(LR)	sR8		uR0(LR)	uR8
sR1	sR9		uR1	uR9
sR2	sR10		uR2	uR10
sR3	sR11		uR3	uR11
sR4	sR12(FP)		uR4	uR12(FP)
sR5	sSP		uR5	uSP
sR6	sCC		uR6	uCC
sR7	sPC		uR7	uPC
Interrupts Disabled			Interrupts Enabled	

Figure 2.1: ZipCPU Register File

Other parameters are defined within the ZipSystem parent module, and affect the performance of the system as a whole.

The `START_HALTED` parameter, if set to non–zero, will cause the CPU to be halted upon startup. This is useful for debugging, since it prevents the CPU from doing anything without supervision. Of course, once all pieces of your design are in place and proven, you'll probably want to set this to zero.

The `EXTERNAL_INTERRUPTS` parameter controls the number of interrupt wires coming into the CPU. This number must be between one and sixteen, or if the performance counters are disabled, between one and twenty four.

2.2 Internal Architecture

This section discusses the general architecture of the CPU itself, separated from its environment. As such, it focuses on the instruction set layout and how those instructions are implemented.

2.2.1 Register Set

Fundamental to the understanding of the ZipCPU is its register set, and the performance model associated with it. The ZipCPU register set contains two sets of sixteen 32-bit registers, a supervisor and a user set as shown in Fig. 2.1. The supervisor set is used when interrupts are disabled, whereas the user set is used any time interrupts are enabled. This choice makes it easy to set up a working context upon any interrupt, as the supervisor register set remains what it was when interrupts were enabled. This sets up one of two modes the CPU can run within: a *supervisor mode*, which runs with interrupts disabled using the supervisor register set, and *user mode*, which runs with interrupts enabled using the user register set.

This separation is so fundamental to the CPU that it is impossible to enable interrupts without switching to the user register set. Further, on any interrupt, exception, or trap, the CPU simply clears the pipeline and switches instruction sets.

In each register set, the Program Counter (PC) is register 15, whereas the status register (SR) or condition code register (CC) is register 14. All other registers are identical in their hardware

Bit #	Access	Description
31...23	R	Reserved for future uses
22...16	R/W	Reserved for future uses
15	R	Reserved for MMU exceptions
14	W	Clear I-Cache command, always reads zero
13	R	VLIW instruction phase (1 for first half)
12	R	(Reserved for) Floating Point Exception
11	R	Division by Zero Exception
10	R	Bus-Error Flag
9	R	Trap Flag (or user interrupt). Cleared on return to userspace.
8	R	Illegal Instruction Flag
7	R/W	Break–Enable (sCC), or user break (uCC)
6	R/W	Step
5	R/W	Global Interrupt Enable (GIE)
4	R/W	Sleep. When GIE is also set, the CPU waits for an interrupt.
3	R/W	Overflow
2	R/W	Negative. The sign bit was set as a result of the last ALU instruction.
1	R/W	Carry
0	R/W	Zero. The last ALU operation produced a zero.

Table 2.1: Condition Code Register Bit Assignment

functionality.[1] By convention, the stack pointer is register 13 and noted as (SP)–although there is nothing special about this register other than this convention. Also by convention, if the compiler needs a frame pointer it will be placed into register 12, and may be abbreviated by FP. Finally, by convention, R0 will hold a subroutine's return address, sometimes called the link register (LR).

When the CPU is in supervisor mode, instructions can access both register sets via the `MOV` instruction, whereas when the CPU is in user mode, `MOV` instructions will only offer access to user registers. We'll discuss this further in subsection. 2.2.9.

2.2.2 The Status Register, CC

The status register (CC) is special, and bears further mention. As shown in Fig. 2.1, the lower sixteen bits of the status register form a set of CPU state and condition codes. The other bits are reserved for future uses.

Of the condition codes, the bottom four bits are the current flags: Zero (Z), Carry (C), Negative (N), and Overflow (V). These flags maintain their usual definition from other CPUs that use them, for all but the shift right instructions. On those instructions that set the flags, these flags will be set based upon the output of certain instructions. If the result is zero, the Z (zero) flag will be set. If the high order bit is set, the N (negative) flag will be set. If the instruction caused a bit to

[1] Jumps to `R0`, an instruction used to implement a return from a subroutine, may be optimized in the future within the early branch logic.

fall off the end, the carry bit will be set. In comparisons, this is equivalent to a less–than unsigned comparison. Finally, if the instruction causes a signed integer overflow, the V (overflow) flag will be set afterwards.

We'll walk through the next many bits of the status register in order from least significant to most significant.

4. The next bit is a sleep bit. Set this bit to one to disable instruction execution and place the CPU to sleep, or to zero to keep the pipeline running. Setting this bit will cause the CPU to wait for an interrupt (if interrupts are enabled), or to completely halt (if interrupts are disabled). This leads to the `WAIT` and `HALT` opcodes which will be discussed more later. In order to prevent users from halting the CPU, only the supervisor is allowed to both put the CPU to sleep and disable interrupts. Any user attempt to do so will simply result in a switch to supervisor mode.

5. The sixth bit is a global interrupt enable bit (GIE). This bit also forms the top, or fifth, bit of any register address. When this sixth bit is a '1' interrupts will be enabled, else disabled. When interrupts are disabled, the CPU will be in supervisor mode, otherwise it is in user mode. Thus, to execute a context switch, one only need enable or disable interrupts. (When an interrupt line goes high, interrupts will automatically be disabled, as the CPU goes and deals with its context switch.) Special logic has been added to keep the user mode from setting the sleep register and clearing the GIE register at the same time, with clearing the GIE register taking precedence.

   Whenever read, the supervisor CC register will always have this bit cleared, whereas the user CC register will always have this bit set.

6. The seventh bit is a step bit in the user CC register, and zero in the supervisor CC director. This bit can only be set from supervisor mode. After setting this bit, should the supervisor mode process switch to user mode, it would then accomplish one instruction in user mode before returning to supervisor mode. This bit has no effect on the CPU while in supervisor mode.

   This functionality was added to enable a userspace debugger functionality on a user process, working through supervisor mode of course.

   The CPU can be stepped in supervisor mode. Doing so requires the CPU debug functionality, not the step bit.

7. The eighth bit is a break enable bit. When applied to the supervisor CC register, this controls whether a break instruction in user mode will halt the processor for an external debugger (break enabled), or whether the break instruction will simply send send the CPU into interrupt mode. This bit can only be set within supervisor mode. However, when applied to the user CC register, from supervisor mode, this bit will indicate whether or not the reason the CPU entered supervisor mode was from a break instruction or not. This break reason bit is automatically cleared upon any transition to user mode, although it can also be cleared by the supervisor writing to the user CC register.

   Encountering a break in supervisor mode will halt the CPU independent of the break enable bit.

This functionality was added to enable a debugger to set and manage breakpoints in a user mode process.

8. The ninth bit is an illegal instruction bit. When the CPU tries to execute either a non-existent instruction, or an instruction from an address that produces a bus error, the CPU will (if implemented) switch to supervisor mode while setting this bit. The bit will automatically be cleared upon any return to user mode.

9. The tenth bit is a trap bit. It is set whenever the user requests a soft interrupt, and cleared on any return to userspace command. This allows the supervisor, in supervisor mode, to determine whether it got to supervisor mode from a trap, from an external interrupt or both.

10. The eleventh bit is a bus error flag. If the user program encountered a bus error, this bit will be set in the user CC register and the CPU will switch to supervisor mode. The bit may be cleared by the supervisor, otherwise it is automatically cleared upon any return to user mode. If the supervisor encounters a bus error, this bit will be set in the supervisor CC register and the CPU will halt. In that case, either a CPU reset or a write to the supervisor CC register will clear this register.

11. The twelfth bit is a division by zero exception flag. This operates in a fashion similar to the bus error flag. If the user attempts to use the divide instruction with a zero denominator, the system will switch to supervisor mode and set this bit in the user CC register. The bit is automatically cleared upon any return to user mode, although it can also be manually cleared by the supervisor. In a similar fashion, if the supervisor attempts to execute a divide by zero, the CPU will halt and set the zero exception flag in the supervisor's CC register. This will automatically be cleared upon any CPU reset, or it may be manually cleared by the external debugger writing to this register.

12. The thirteenth bit will operate in a similar fashion to both the bus error and division by zero flags, only it will be set upon a (yet to be determined) floating point error.

13. In the case of VLIW instructions, if an exception occurs after the first instruction but before the second, the fourteenth bit of the CC register will be set to indicate this fact.

14. The fifteenth bit references a clear cache bit. The supervisor may write a one to this bit in order to clear the CPU instruction cache. The bit always reads as a zero.

15. Last, but not least, the sixteenth bit is reserved for a page not found memory exception to be created by the memory management unit.

Some of the upper bits have been temporarily assigned to indicate CPU capabilities. This is not a permanent feature, as these upper bits officially remain reserved.

2.2.3 Instruction Format

All ZipCPU instructions fit in one of the formats shown in Fig. 2.2. The basic format is that some operation, defined by the OpCode, is applied if a condition, Cnd, is true in order to produce a result which is placed in the destination register (DR). There are three basic exceptions to this model. The first is the `MOV` instruction, which steals bits 13 and 18 to allow supervisor access to user registers.

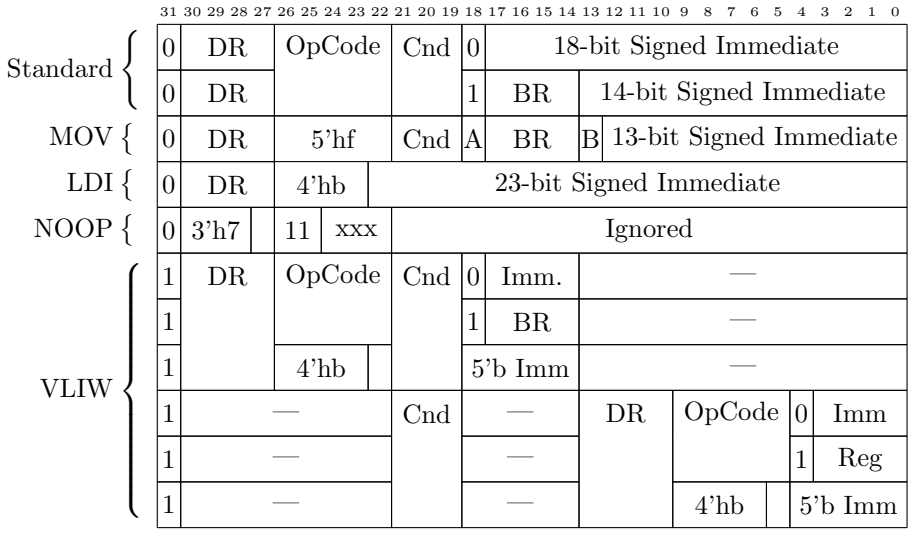

Figure 2.2: Zip Instruction Set Format

The second is the load 23–bit signed immediate instruction (`LDI`), in that it accepts no conditions and uses only a 4-bit opcode. The last exception is the `NOOP` instruction group, containing the `NOOP`, `BREAK`, and `LOCK` opcodes. These instructions ignore their register and immediate settings.[2]

The ZipCPU also supports a very long instruction word (VLIW) set of instructions. These aren't truly VLIW instructions in the sense that the CPU still only issues one instruction at a time, but they do pack two instructions into a single instuction word. The number of bits used by the immediate field are adjusted to make space for these instruction words. Other than instruction format, the only basic difference between VLIW and normal instructions is that the CPU will not switch to interrupt mode in between the two instructions, unless an exception is generated by the first instruction. Likewise a new job given to the assembler is that of automatically packing as many instructions as possible into the VLIW format.

The disassembler will represent VLIW instructions by placing a vertical bar between the two components, but still leaving them on the same line.

2.2.4 Instruction OpCodes

With a 5–bit opcode field, there are 32–possible instructions as shown in Tbl. 2.2. Of these opcodes, `ROL` and `POPC` are experimental and may be replaced in future revisions. (If you have a reason to like or wish to keep these opcodes, please contact me. If you know of alternatives that might be better, please let me know as well.) There is also room for six more register-less instructions in the `NOOP` instruction space, and two floating point instruction opcodes have been reserved for future use.

[2] A future version of the CPU may repurpose the immediate bits within the `NOOP` instruction to be simulator commands, while the immediate/register bits within the `BREAK` instruction may be used by the debugger for whatever purpose it chooses to use them for–such as a breakpoint table index.

OpCode		Instruction	Sets CC
5'h00	SUB	Subtract	Y
5'h01	AND	Bitwise And	
5'h02	ADD	Add two numbers	
5'h03	OR	Bitwise Or	
5'h04	XOR	Bitwise Exclusive Or	
5'h05	LSR	Logical Shift Right	
5'h06	LSL	Logical Shift Left	
5'h07	ASR	Arithmetic Shift Right	
5'h08	MPY	32x32 bit multiply	Y
5'h09	LDILO	Load Immediate Low	N
5'h0a	MPYUHI	Upper 32 of 64 bits from an unsigned 32x32 multiply	Y
5'h0b	MPYSHI	Upper 32 of 64 bits from a signed 32x32 multiply	
5'h0c	BREV	Bit Reverse B operand into result	
5'h0d	POPC	Population Count	
5'h0e	ROL	Rotate Ra left by OpB bits	
5'h0f	MOV	Move OpB into Ra	N
5'h10	CMP	Compare (Ra-OpB) to zero	Y
5'h11	TST	Test (AND w/o setting result)	
5'h12	LOD	Load Ra from memory (OpB)	N
5'h13	STO	Store Ra into memory at (OpB)	
5'h14	DIVU	Divide, unsigned	Y
5'h15	DIVS	Divide, signed	
5'h16/7	LDI	Load 23–bit signed immediate	N
5'h18	FPADD	Floating point add	Y
5'h19	FPSUB	Floating point subtract	
5'h1a	FPMPY	Floating point multiply	
5'h1b	FPDIV	Floating point divide	
5'h1c	FPI2F	Convert integer to floating point	
5'h1d	FPF2I	Convert floating point to integer	
5'h1e		*Reserved for future use*	
5'h1f		*Reserved for future use*	
5'h18		NOOP (A-register = PC)	N
5'h19		BREAK (A-register = PC)	
5'h1a		LOCK (A-register = PC)	

Table 2.2: ZipCPU OpCodes

Code	Mnemonic	Condition
3'h0	None	Always execute the instruction
3'h1	`.LT`	Less than ('N' set)
3'h2	`.Z`	Only execute when 'Z' is set
3'h3	`.NZ`	Only execute when 'Z' is not set
3'h4	`.GT`	Greater than ('N' not set, 'Z' not set)
3'h5	`.GE`	Greater than or equal ('N' not set, 'Z' irrelevant)
3'h6	`.C`	Carry set (Also known as less-than unsigned)
3'h7	`.V`	Overflow set

Table 2.3: Conditions for conditional operand execution

```
CMP 1,R0
; Condition codes are now set based upon R0-1
CMP.Z 2,R1
; If R0 ≠ 1, conditions are unchanged, Z is still false.
; If R0 = 1, conditions are now set based upon R1-2.
; Now some instruction could be done based upon the conjunction
; of both conditions.
; While we use the example of a STO, it could easily be any instruction.
STO.Z R0,(R2)
```

Table 2.4: An example of a double conditional

2.2.5 Conditional Instructions

Most, although not quite all, instructions may be conditionally executed. The 23–bit load immediate instruction, together with the `NOOP`, `BREAK`, and `LOCK` instructions are the exceptions to this rule. All other instructions may be conditionally executed.

From the four condition code flags, eight conditions are defined for standard instructions. These are shown in Tbl. 2.3. There is no condition code for less than or equal, not C or not V—there just wasn't enough space in 3–bits. Ways of handling non–supported conditions are discussed in Sec. 2.2.6.

With the exception of `CMP` and `TST` instructions, conditionally executed instructions will not further adjust the condition codes. Conditional `CMP` or `TST` instructions will adjust conditions whenever they are executed. In this way, multiple conditions may be evaluated without branches, creating a sort of logical and–but only if all the conditions are the same. For example, to do something if `R0` is one and `R1` is two, one might try code such as Tbl. 2.4.

The real utility of conditionally executed instructions is that, unlike conditional branches, conditionally executed instructions will not stall the bus if they are not executed.

In the case of VLIW instructions, only four conditions are defined as shown in Tbl. 2.5. Further, the first bit of the three is given a special meaning: If the first bit is set, the conditions apply to the second half of the instruction, otherwise the conditions will only apply to the first half of a

Code	Mnemonic	Condition
2'h0	None	Always execute the instruction
2'h1	`.LT`	Less than ('N' set)
2'h2	`.Z`	Only execute when 'Z' is set
2'h3	`.NZ`	Only execute when 'Z' is not set

Table 2.5: VLIW Conditions

Original	Modified	Name
`CMP Rx,Ry` `BLE label`	`CMP 1+Rx,Ry` `BLT label`	Less-than or equal (signed, `Z` or `N` set)
`CMP Rx,Ry` `BLEU label`	`CMP 1+Rx,Ry` `BC label`	Less-than or equal unsigned
`CMP Rx,Ry` `BGTU label`	`CMP Ry,Rx` `BC label`	Greater-than unsigned
`CMP Rx,Ry` `BGEU label`	`CMP 1+Ry,Rx` `BC label`	Greater-than equal unsigned
`CMP A+Rx,Ry` `BGEU label`	`CMP (1-A)+Ry,Rx` `BC label`	Greater-than equal unsigned (with offset)

`CMP A,Ry`
`BGEU label`
	`LDI (A-1),Rx`
	`CMP Ry,Rx`
	`BC label`
		Greater-than equal comparison with a constant

Table 2.6: Modifying conditions

conditional instruction. Of course, the other conditions are still available by mingling the non–VLIW instructions with VLIW instructions.

2.2.6 Modifying Conditions

A quick look at the list of conditions supported by the ZipCPU and listed in Tbl. 2.3 reveals that the ZipCPU does not have a full set of conditions. In particular, only one explicit unsigned condition is supported. Therefore, Tbl. 2.6 shows examples of how these unsupported conditions can be created simply by adjusting the compare instruction, for no extra cost in clocks. Of course, if the compare originally had an immediate within it, that immediate would need to be loaded into a register in order to do make some of these adjustments. That case is shown as the last case above.

Many of these alternate conditions are chosen by the compiler implementation.

Users should be aware of any signed overflow that might take place within the modified conditions, especially when numbers close to the limit are used.

2.2.7 Operand B

Many instruction forms have a 19-bit source "Operand B", or OpB for short, associated with them. This "Operand B" is shown in Fig. 2.2 as part of the standard instructions. An Operand B is

18	17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0	18-bit Signed Immediate
1	Reg / 14-bit Signed Immediate

Table 2.7: Bit allocation for Operand B

either equal to a register plus a 14–bit signed immediate offset, or an 18–bit signed immediate offset by itself. This value is encoded as shown in Tbl. 2.7. This format represents a deviation from many other RISC architectures that use `R0` to represent zero, such as OpenRISC and RISC-V. Here, instead, we use a bit within the instruction to note whether or not an immediate is used. The result is that ZipCPU instructions can encode larger immediates within their instruction space.

In those cases where a fourteen or eighteen bit immediate doesn't make sense, such as for `LDILO`, the extra bits associated with the immediate are simply ignored. (This rule does not apply to the shift instructions, `ASR`, `LSR`, and `LSL`–which all use all of their immediate bits.)

VLIW instructions still use the same operand B as regular instructions, only there was no room for any instruction plus immediate addressing. Therefore, VLIW instructions have either a register or a 4–bit signed immediate as their operand B. The only exception is the load immediate instruction, which permits a 5–bit signed operand B.[3]

2.2.8 Address Modes

The ZipCPU supports two addressing modes: register plus immediate, and immediate addressing. Addresses are encoded in the same fashion as Operand B's, discussed above.

The VLIW instruction set only offers register addressing.

2.2.9 Move Operands

The previous set of operands would be perfect and complete, save only that the CPU needs access to non–supervisory registers while in supervisory mode. The MOV instruction has been modified to fit that purpose. The two bits, shown as `A` and `B` in Fig. 2.2 above, are designed to contain the high order bit of the 5–bit register index. If the `B` bit is a '1', the source operand comes from the user register set. If the `A` bit is a '1', the destination operand is in the user register set. A zero bit indicates the current register set.

This encoding has been chosen to keep the compiler simple. For the most part, the extra bits are quietly set to zero by the compiler. Assembly instructions, or particular built–in instructions, can be used to get access to these cross register set move instructions.

Further, the `MOV` instruction lacks the full OpB capability to use a register or a register plus immediate as a source, since a load immediate instruction already exists. As a result, all moves come from a register plus a potential offset.

[3] Although the space exists to extend this VLIW load immediate instruction to six bits, the 5–bit limit was chosen to simplify the disassembler. This may change in the future.

2.2.10 Multiply Operations

The ZipCPU supports three separate 32x32-bit multiply instructions: `MPY`, `MPYUHI`, and `MPYSHI`. The first of these produces the low 32-bits of a 32x32-bit multiply result. The second two produce the upper 32-bits. The first, `MPYUHI`, produces the upper 32-bits assuming the multiply was unsigned, whereas `MPYSHI` assumes it was signed. Each multiply instruction is independent of every other in execution, although the compiler is likely to use them in a dependent fashion.

In an effort to maintain a fast clock speed, all three of these multiplies have been slowed down in logic. Thus, depending upon the setting of `OPT_MULTIPLY` within `cpudefs.v`, or the corresponding `IMPLEMENT_MPY` parameter that may override it, the multiply instructions will either 1) cause an ILLEGAL instruction error (`OPT_MULTIPLY=0`, or no multiply support), 2) take one additional clock (`OPT_MULTIPLY=2`), or 3) take two additional clock cycles (`OPT_MULTIPLY=3`).[4]

2.2.11 Divide Unit

The ZipCPU also has an optional divide unit which can be built alongside the ALU. This divide unit provides the ZipCPU with another two instructions that cannot be executed in a single cycle: `DIVS`, or signed divide, and `DIVU`, the unsigned divide. These are both 32–bit divide instructions, dividing one 32–bit number by another. In this case, the Operand B field, whether it be register or register plus immediate, constitutes the denominator, whereas the numerator is given by the other register.

As with the multiply, the divide instructions are also a multi–clock instructions. While the divide is running, the ALU, any memory loads, and the floating point unit (if installed) will be idle. Once the divide completes, other units may continue.

Of course, any divide instruction can result in a division by zero exception. If this happens the CPU will either suddenly transition from user mode to supervisor mode, or it will halt if the CPU is already in supervisor mode. Upon exception, the divide by zero bit will be set in the CC register. In the case of a user mode divide by zero, this will be cleared by any return to user mode command. The supervisor bit may be cleared either by a reboot or by the external debugger.

2.2.12 NOOP, BREAK, and Bus LOCK Instruction

Three instructions within the opcode list in Tbl. 2.2, are somewhat special. These are the `NOOP`, `BREAK`, and bus `LOCK` instructions. These are encoded according to Fig. 2.3.

The `NOOP` instruction is just that: an instruction that does not perform any operation. While many other instructions, such as a move from a register to itself, could also fit this role, only the NOOP instruction guarantees that it will not stall waiting for a register to be available. For this reason, it gets its own place in the instruction set. Bits 21–0 of this instruction are reserved for commands which may be given to a simulator, such as simulator exit, should the code be run from a simulator. However, such simulation codes have not yet been defined.

The `BREAK` instruction is useful for creating a debug instruction that will halt the CPU without executing. If in user mode, depending upon the setting of the break enable bit, it will either switch to supervisor mode or halt the CPU–depending upon where the user wishes to do his debugging. The lower 22 bits of this instruction are likewise reserved for the debuggers use.

[4]Support also exists for a one clock multiply (no clock slowdown), or a four clock multiply, and I am anticipating supporting a much longer multiply for FPGA architectures with no accelerated hardware multiply support.

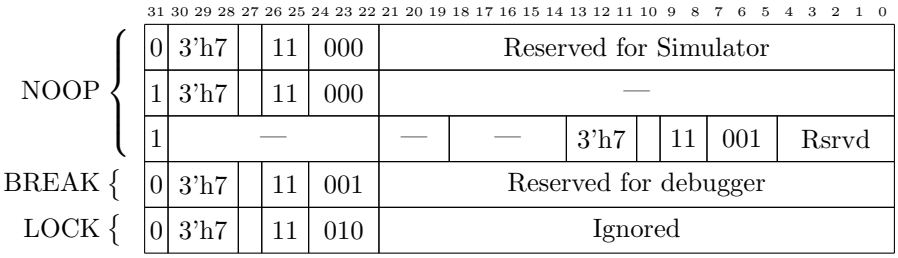

Figure 2.3: NOOP/Break/LOCK Instruction Format

Finally, the `LOCK` instruction was added in order to provide for atomic operations. The `LOCK` instruction only works when the CPU is configured for pipeline mode. It works by stalling the ALU pipeline stack until all prior stages are filled, and then it guarantees that once a bus cycle is started, the wishbone `CYC` line will remain asserted until the LOCK is deasserted. This allows the execution of three instructions, one memory (ex. `LOD`), one ALU (ex. `ADD`), and another memory instruction (ex. `STO`), to take place in an unbreakable fashion. Example uses of this capability include an atomic increment, such as `LOCK`, `LOD (Rx),Ry`, `ADD #,Ry`, and then `STO Ry,(Rx)`, or even a two instruction pair such as a test and set sequence: `LDI 1,Rz`, `LOCK`, `LOD (Rx),Ry`, `STO Rz,(Rx)`.

2.2.13 Floating Point

Although the ZipCPU does not (yet) have a floating point unit, the current instruction set offers eight opcodes for floating point operations, and treats floating point exceptions like divide by zero errors. Once this unit is built and integrated together with the rest of the CPU, the ZipCPU will support 32–bit floating point instructions natively. Any 64–bit floating point instructions will either need to be emulated in software, or else they will need an external floating point peripheral.

Until the FPU is built and integrated, of even afterwards if the floating point unit is not installed by option, floating point instructions will trigger an illegal instruction exception, which may be trapped and then implemented in software.

2.2.14 Load/Store byte

One difference between the ZipCPU and many other architectures is that there are no load byte `LB`, store byte `SB`, load halfword `LH` or store halfword `SH` instructions. This lack is by design in an attempt to keep the 32–bit bus simple.

Because the ZipCPU's addresses refer to 32–bit values, i.e. address one will refer to a completely different 32–bit value than address two, simulating these load and store byte instructions is difficult.

This is just the nature of the ZipCPU, as a result of the design choices that were made.

2.2.15 Derived Instructions

The ZipCPU supports many other common instructions by construction, although not all of them are single cycle instructions. Tables 2.8, 2.9, 2.10 and 2.11 show how these other instructions may

be implemented on the ZipCPU. Many of these instructions will have assembly equivalents, such as the branch instructions, to facilitate working with the CPU.

2.2.16 Interrupt Handling

The ZipCPU does not maintain any interrupt vector tables. If an interrupt takes place, the CPU simply switches to from user to supervisor (interrupt) mode. The supervisor code then continues from where it left off after executing a return to userspace `RTU` instruction.

Since the CPU may return from userspace after either an interrupt, a trap, or an exception, it is up to the supervisor code that handles the transition to determine which of the three has taken place.

2.2.17 Pipeline Stages

As mentioned in the introduction, and highlighted in Fig. 1.1, the ZipCPU supports a five stage pipeline.

1. **Prefetch**: Reads instructions from memory. If the CPU has been configured with a cache, the cache has been integrated into the prefetch. Stalls are also created here if the instruction isn't in the prefetch cache.

   The ZipCPU supports one of three prefetch methods, depending upon the flags set at build time within the `cpudefs.v` file.

   The simplest is a non–cached implementation of a prefetch. This implementation is fairly small, and ideal for users of the ZipCPU who need the extra space on the FPGA fabric. However, because this non–cached version has no cache, the maximum number of instructions per clock is limited to about one per eight–depending upon the bus/memory delay. This prefetch option is set by leaving the `OPT_SINGLE_FETCH` line uncommented within the `cpudefs.v` file. Using this option will also turn off the ZipCPU pipeline.

   The second prefetch module is a non–traditional pipelined prefetch with a cache. This module tries to keep the instruction address within a window of valid instruction addresses. While effective, it is not a traditional cache implementation. A disappointing feature of this implementation is that it needs an extra internal pipeline stage to be implemented.

   The third prefetch and cache module implements a more traditional cache. This cache provides for the fastest CPU speed. The only drawback is that, when a cache line is loading, the CPU will be stalled until the cache is completely loaded.

2. **Decode**: Decodes an instruction into it's OpCode, register(s) to read, condition code, and immediate offset. This stage also determines whether the flags will be read or set, whether registers will be read (and hence the pipeline may need to stall), or whether the result will be written back. In many ways, simplifying the CPU also meant simplifying this pipeline stage and hence the instruction set architecture.

3. **Read Operands**: Read from the register file and applies any immediate values to the result. There is no means of detecting or flagging arithmetic overflow or carry when adding the immediate to the operand. This stage will stall if any source operand is pending and the immediate value is non–zero.

Mapped	Actual	Notes
`ABS Rx`	`TST -1,Rx` `NEG.LT Rx`	Absolute value, depends upon the derived `NEG` instruction below, and so this expands into three instructions total.

`ADD Ra,Rx`
`ADDC Rb,Ry`
	`Add Ra,Rx`
	`ADD.C $1,Ry`
	`Add Rb,Ry`
		Add with carry

`BRA.`x `+/-$Addr`	`ADD.`x `$Addr+PC,PC`	Branch or jump on condition x. Works for 18–bit signed address offsets.
`BUSY`	`ADD $-1,PC`	Execute an infinite loop. This is used within ZipCPU simulations as the execute simulation on error instruction.
`CLRF.NZ Rx`	`XOR.NZ Rx,Rx`	Clear Rx, and flags, if the Z-bit is not set
`CLR Rx`	`LDI $0,Rx`	Clears Rx, leaving the flags untouched. This instruction cannot be conditional.
`CLR.NZ Rx`	`BREV.NZ $0,Rx`	Clears Rx, leaving the flags untouched. This instruction can be executed conditionally. The assembler will quietly choose between `LDI` and `BREV` depending upon the existence of the condition.
`EXCH.W Rx`	`ROL $16,Rx`	Exchanges the top and bottom 16'bit words of Rx
`HALT`	`Or $SLEEP,CC`	This only works when issued in interrupt/supervisor mode. In user mode this is simply a wait until interrupt instruction. This is also used within the simulator as an exit simulation on success instruction.
`INT`	`LDI $0,CC`	This is also known as a trap instruction
`IRET`	`OR $GIE,CC`	Also known as an RTU instruction (Return to Userspace)
`JMP R6+$Offset`	`MOV $Offset(R6),PC`	Only works for 13–bit offsets. Other offsets require adding the offset first to R6 before jumping.
`LJMP $Addr`	`LOD (PC),PC` *Address*	Although this only works for an unconditional jump, and it only works in an architecture with a unified instruction and data address space, this instruction combination makes for a nice combination that can be adjusted by a linker at a later time.

`LJMP.x $Addr`	`LOD.x 2(PC),PC`
	`ADD 1,PC`
	Address
		Long jump, works for a conditional long jump.

Table 2.8: Derived Instructions

Mapped	Actual	Notes

`LJSR $Addr`	`MOV $2+PC,R0`
	`LOD (PC),PC`
	Address
		Similar to LJMP, but it handles the return address properly.

`JSR PC+$Offset`	`MOV $1+PC,R0` `ADD $Offset,PC`	This is similar to the jump and link instructions from other architectures, save only that it requires a specific link instruction, seen here as the `MOV` instruction on the left.
`LDI $val,Rx`	`BREV REV(`*val*`)&0x0ffff, Rx` `LDILO (`*val*`&0x0ffff),Rx`	Sadly, there's not enough instruction space to load a complete immediate value into any register. Therefore, fully loading any register takes two cycles. The `LDILO` (load immediate low) instruction has been created to facilitate this together with `BREV`. This is also the appropriate means for setting a register value to an arbitrary 32–bit value in a post–assembly link operation.

`LOD.b $addr,Rx`	`LDI $addr,Ra`
	`LDI $addr,Rb`
	`LSR $2,Ra`
	`AND $3,Rb`
	`LOD (Ra),Rx`
	`LSL $3,Rb`
	`SUB $32,Rb`
	`ROL Rb,Rx`
	`AND $0ffh,Rx`
		This CPU is designed for 32'bit word length instructions. Byte addressing is not supported by the CPU or the bus, so it therefore takes more work to do.
		Note also that in this example, $Addr is a bytewise address, where all other addresses in this document are 32-bit wordlength addresses. For this reason, we needed to drop the bottom two bits. This also limits the address space of character accesses using this method from 16 MB down to 4MB.

`LSL $1,Rx`
`LSLC $1,Ry`
	`LSL $1,Ry`
	`LSL $1,Rx`
	`OR.C $1,Ry`
		Logical shift left with carry. Note that the instruction order is now backwards, to keep the conditions valid. That is, LSL sets the carry flag, so if we did this the other way with Rx before Ry, then the condition flag wouldn't have been right for an `OR` correction at the end.

`LSR $1,Rx`
`LSRC $1,Ry`
	`CLR Rz`
	`LSR $1,Ry`
	`BREV.C $1,Rz`
	`LSR $1,Rx`
	`OR Rz,Rx`
		Logical shift right with carry. Unlike the shift left, this approach doesn't extend well to numbers larger than two words.

Table 2.9: Derived Instructions, continued

`NEG Rx`	`XOR $-1,Rx` `ADD $1,Rx`	Negates Rx
`NEG.C Rx`	`MOV.C $-1+Rx,Rx` `XOR.C $-1,Rx`	Conditionally negates Rx
`NOT Rx`	`XOR $-1,Rx`	One's complement
`POP Rx`	`LOD $(SP),Rx` `ADD $1,SP`	The compiler avoids the need for this instruction and the similar `PUSH` instruction when setting up the stack by coalescing all the stack address modifications into a single instruction at the beginning of any stack frame.
`PUSH Rx`	`SUB $1,SP` `STO Rx,$(SP)`	Note that for pipelined operation, it helps to coalesce all the `SUB`'s into one command, and place the `STO`'s right after each other. Further, to avoid a pipeline stall, the immediate value for the first store must be zero.

`PUSH Rx-Ry`	`SUB $`n`,SP`
	`STO Rx,$(SP) ...`
	`STO Ry,$`$(n-1)$`(SP)`
		Multiple pushes at once only need the single subtract from the stack pointer. This derived instruction is analogous to a similar one on the Motorola 68k architecture, although the Zip Assembler does not support the combined instruction. This instruction also supports pipelined memory access.

`RESET`	`STO`
	`$1,$watchdog(R12)`
	`BUSY`
		This depends upon the existence of a watchdog peripheral, and the peripheral base address being preloaded into `R12`. The BUSY instructions are required because the CPU will continue until the `STO` has completed.
		Another opportunity might be to jump to the reset address from within supervisor mode.

`RET`	`MOV R0,PC`	This depends upon the form of the `JSR` given on the previous page that stores the return address into R0.
`SEX.b Rx`	`LSL 24,Rx` `ASR 24,Rx`	Signed extend an 8–bit value into a full word.
`SEX.h Rx`	`LSL 16,Rx` `ASR 16,Rx`	Sign extend a 16–bit value into a full word.
`STEP Rr,Rt`	`LSR $1,Rr` `XOR.C Rt,Rr`	Step a Galois implementation of a Linear Feedback Shift Register, Rr, using taps Rt
`STEP`	`OR $Step\|$GIE,CC`	Steps a user mode process by one instruction

Table 2.10: Derived Instructions, continued

`STO.b Rx,$addr`	`LDI $addr,Ra`
	`LDI $addr,Rb`
	`LSR $2,Ra`
	`AND $3,Rb`
	`SUB $32,Rb`
	`LOD (Ra),Ry`
	`AND $0ffh,Rx`
	`AND ~$0ffh,Ry`
	`ROL Rb,Rx`
	`OR Rx,Ry`
	`STO Ry,(Ra)`
		This CPU and its bus are *not* optimized for byte-wise operations.
		Note that in this example, $addr is a byte-wise address, whereas in all of our other examples it is a 32-bit word address. This also limits the address space of character accesses from 16 MB down to 4MB. Further, this instruction implies a byte ordering, such as big or little endian.

`SUBR Rx,Ry`	`SUB 1+Rx,Ry` `XOR -1,Ry`	Ry is set to Rx-Ry, rather than the normal subtract which sets Ry to Ry-Rx.

`SUB Ra,Rx`
`SUBC Rb,Ry`
	`SUB Ra,Rx`
	`SUB.C $1,Ry`
	`SUB Rb,Ry`
		Subtract with carry. Note that the overflow flag may not be set correctly after this operation.

`SWAP Rx,Ry`	`XOR Ry,Rx`
	`XOR Rx,Ry`
	`XOR Ry,Rx`
		While no extra registers are needed, this example does take 3-clocks.

`TRAP #X`	`LDI $x,R1` `AND ~$GIE,CC`	This works because whenever a user lowers the $GIE flag, it sets a TRAP bit within the uCC register. Therefore, upon entering the supervisor state, the CPU only need check this bit to know that it got there via a TRAP. The trap could be made conditional by making the LDI and the AND conditional. In that case, the assembler would quietly turn the LDI instruction into a `BREV`/`LDILO` pair, but the effect would be the same.

`TS Rx,Ry,(Rz)`	`LDI 1,Rx`
	`LOCK`
	`LOD (Rz),Ry`
	`STO Rx,(Rz)`
		A test and set instruction. The `LOCK` instruction insures that the next two instructions lock the bus between the instructions, so no one else can use it. Thus guarantees that the operation is atomic.

`TST Rx`	`TST $-1,Rx`	Set the condition codes based upon Rx without changing Rx. Equivalent to a CMP $0,Rx.
`WAIT`	`Or $GIE | $SLEEP,CC`	Wait until the next interrupt, then jump to supervisor/interrupt mode.

Table 2.11: Derived Instructions, continued

4. At this point, the processing flow splits into one of four tracks: An **ALU** track which will accomplish a simple instruction, the **MemOps** stage which handles `LOD` (load) and `STO` (store) instructions, the **divide** unit, and the **floating point** unit.

   - Loads will stall instructions in the read operands stage until the entire memory is complete, lest a register be read only to be updated unseen by the Load.
   - Condition codes are set upon completion of the ALU, divide, or FPU stage. (Memory operations do not set conditions.)
   - Issuing a non–pipelined memory instruction to the memory unit while the memory unit is busy will stall the entire pipeline until the memory unit is idle and ready to accept another instruction.

5. **Write-Back**: Conditionally write back the result to the register set, applying the condition and any special CC logic. This routine is quad-entrant: either the ALU, the memory, the divide, or the FPU may commit a result. The only design rule is that no more than a single register may be written in any given clock cycle.

   This is also the stage where any special condition code logic takes place.

The ZipCPU does not support out of order execution. Therefore, if the memory unit stalls, every other instruction stalls. The same is true for divide or floating point instructions–all other instructions will stall while waiting for these to complete. Memory stores, however, can take place concurrently with non–memory operations, although memory reads (loads) cannot. This is likely to change with the integration of an memory management unit (MMU), in which case a store instruction must stall the CPU until it is known whether or not the store address can be mapped by the MMU.

2.2.18 Pipeline Stalls

The processing pipeline can and will stall for a variety of reasons. Some of these are obvious, some less so. These reasons are listed below:

- When the prefetch cache is exhausted

  This reason should be obvious. If the prefetch cache doesn't have the instruction in memory, the entire pipeline must stall until an instruction can be made ready. In the case of the `pipefetch` windowed approach to the prefetch cache, this means the pipeline will stall until enough of the prefetch cache is loaded to support the next instruction. In the case of the more traditional `pfcache` approach, the entire cache line must fill before instruction execution can continue.

- While waiting for the pipeline to load following any taken branch, jump, return from interrupt or switch to interrupt context (4 stall cycles, minimum)

  Fig. 2.4 illustrates the situation for a conditional branch. In this case, the branch instruction, `BC`, is nominally followed by instructions `I1` and so forth. However, since the branch is taken, the next instruction must be `IA`. Therefore, the pipeline needs to be cleared and reloaded. Given that there are five stages to the pipeline, that accounts for the four stalls. (Were the `pipefetch` cache chosen, there would be another stall internal to the `pipefetch` cache.)

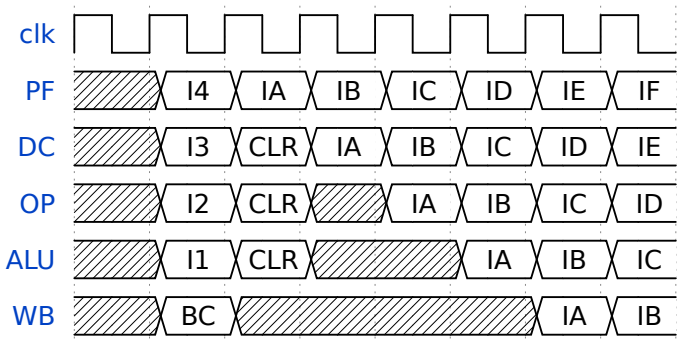

Figure 2.4: A conditional branch generates 4 stall cycles

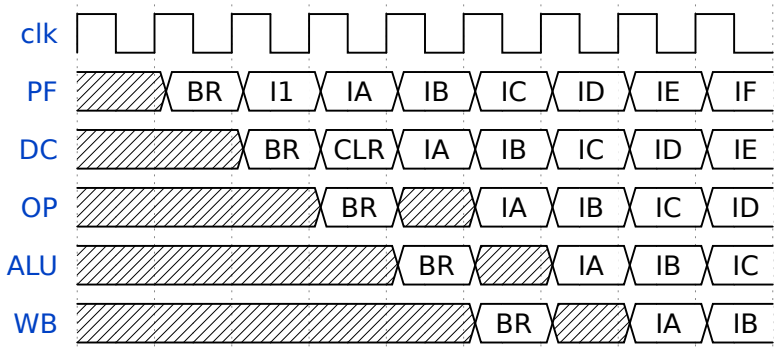

Figure 2.5: An expedited branch costs a single stall cycle

The decode stage can handle the `ADD $X,PC`, `LDI $X,PC`, and `LOD (PC),PC` instructions specially, however, when `EARLY_BRANCHING` is enabled. These instructions, when not conditioned on the flags, can execute with only a single stall cycle (two for the `LOD(PC),PC` instruction), such as is shown in Fig. 2.5. In this example, `BR` is a branch always taken, `I1` is the instruction following the branch in memory, while `IA` is the first instruction at the branch address. (`CLR` denotes a clear–pipeline operation, and does not represent any instruction.)

- When reading from a prior register while also adding an immediate offset

  1. `OPCODE ?,RA`
  2. *(stall)*
  3. `OPCODE I+RA,RB`

  Since the addition of the immediate register within OpB decoding gets applied during the read operand stage so that it can be nicely settled before the ALU, any instruction that will write back an operand must be separated from the opcode that will read and apply an immediate offset by one instruction. The good news is that this stall can easily be mitigated by proper

scheduling. That is, any instruction that does not add an immediate to `RA` may be scheduled into the stall slot.

This is also the reason why, when setting up a stack frame, the top of the stack frame is used first: it eliminates this stall cycle.[5] Hence, to save registers at the top of a procedure, one would write:

1. `SUB 2,SP`
2. `STO R1,(SP)`
3. `STO R2,1(SP)`

Had `R1` instead been stored at `1(SP)` as the top of the stack, there would've been an extra stall in setting up the stack frame.

- When reading from the CC register after setting the flags

  1. `ALUOP RA,RB` *; Ex: a compare opcode*
  2. *(stall)*
  3. `TST sys.ccv,CC`
  4. `BZ somewhere`

  The reason for this stall is simply performance: many of the flags are determined via combinatorial logic *during* the writeback cycle. Trying to then place these into the input for one of the operands for an ALU instruction during the same cycle created a time delay loop that would no longer execute in a single 100 MHz clock cycle. (The time delay of the multiply within the ALU wasn't helping either . . .).

  This stall may be eliminated via proper scheduling, by placing an instruction that does not set flags in between the ALU operation and the instruction that references the CC register. For example, `MOV $addr+PC,uPC` followed by an `RTU` (`OR $GIE,CC`) instruction will not incur this stall, whereas an `OR $BREAKEN,CC` followed by an `OR $STEP,CC` will incur the stall, while a `LDI $BREAKEN|$STEP,CC` will not since it doesn't read the condition codes before executing.

- When waiting for a memory read operation to complete

  1. `LOD address,RA`
  2. *(multiple stalls, bus dependent, 4 clocks best)*
  3. `OPCODE I+RA,RB`

  Remember, the ZipCPU does not support out of order execution. Therefore, anytime the memory unit becomes busy both the memory unit and the ALU must stall until the memory unit is cleared. This is illustrated in Fig. 2.6, since it is especially true of a load instruction, which must still write its operand back to the register file. Further, note that on a pipelined memory operation, the instruction must stall in the decode operand stage, lest it try to read a result from the register file before the load result has been written to it. Finally, note that there is an extra stall at the end of the memory cycle, so that the memory unit will be idle for two clocks before an instruction will be accepted into the ALU. Store instructions are different,

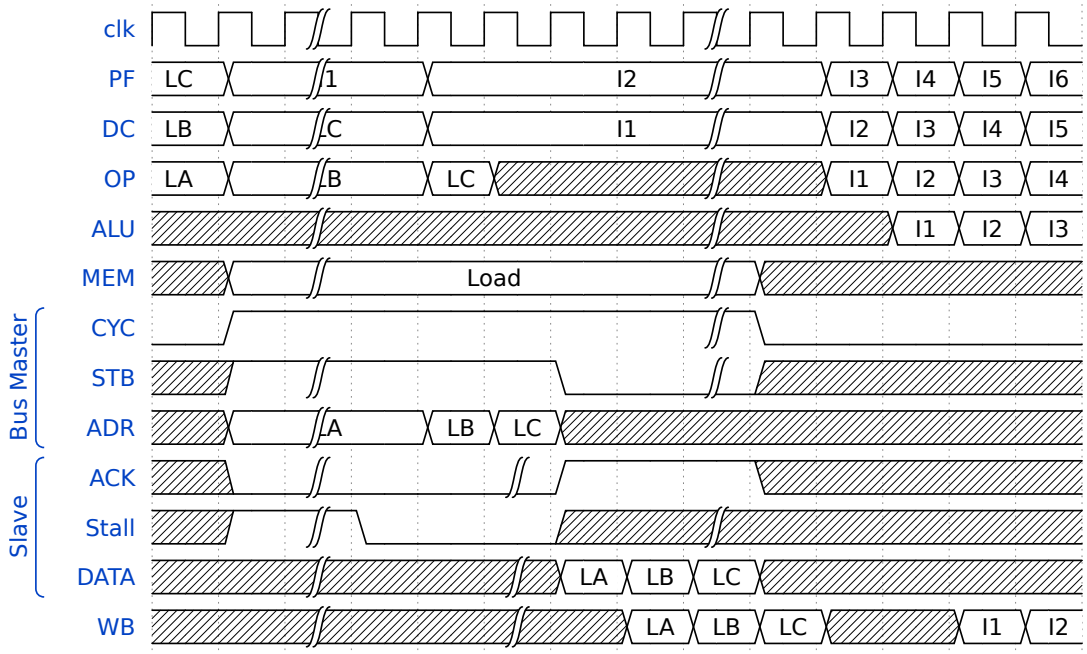

Figure 2.6: Pipeline handling of a load instruction

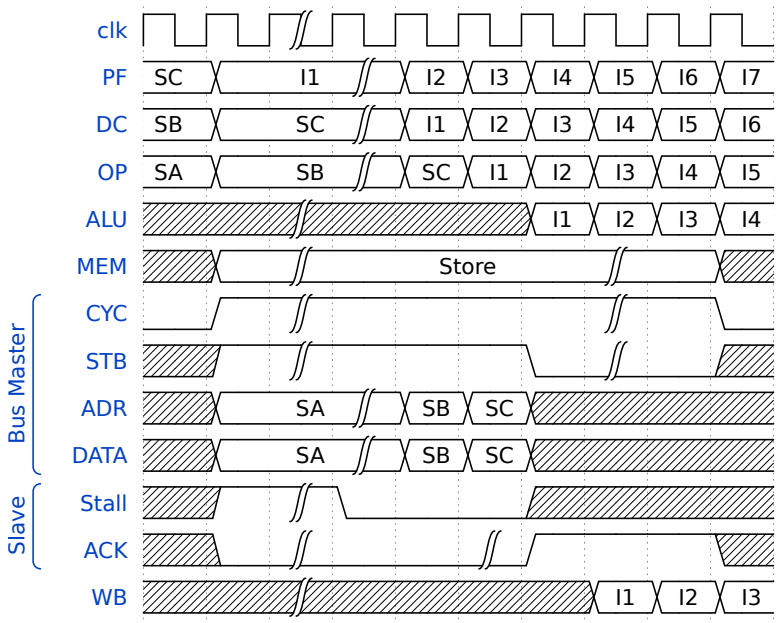

Figure 2.7: Pipeline handling of a store instruction

as shown in Fig. 2.7, since they can be busy with the bus without impacting later write back pipeline stages. Hence, only loads stall the pipeline.

This, of course, also assumes that the memory being accessed is a single cycle memory and that there are no stalls to get to the memory. Slower memories, such as the Quad SPI flash, will take longer–perhaps even as long as forty clocks. During this time the CPU and the external bus will be busy, and unable to do anything else. Likewise, if it takes a couple of clock cycles for the bus to be free, as shown in both Figs. 2.6 and 2.7, there will be stalls.

- Memory operation followed by a memory operation

  1. `STO address,RA`
  2. *(multiple stalls, bus dependent, 4 clocks best)*
  3. `LOD address,RB`
  4. *(multiple stalls, bus dependent, 4 clocks best)*

  In this case, the LOD instruction cannot start until the STO is finished, as illustrated by Fig. 2.8. With proper scheduling, it is possible to do something in the ALU while the memory unit is busy with the STO instruction, but otherwise this pipeline will stall while waiting for it to complete before the load instruction can start.

[5]This only applies if there is no local memory to allocate on the stack as well.

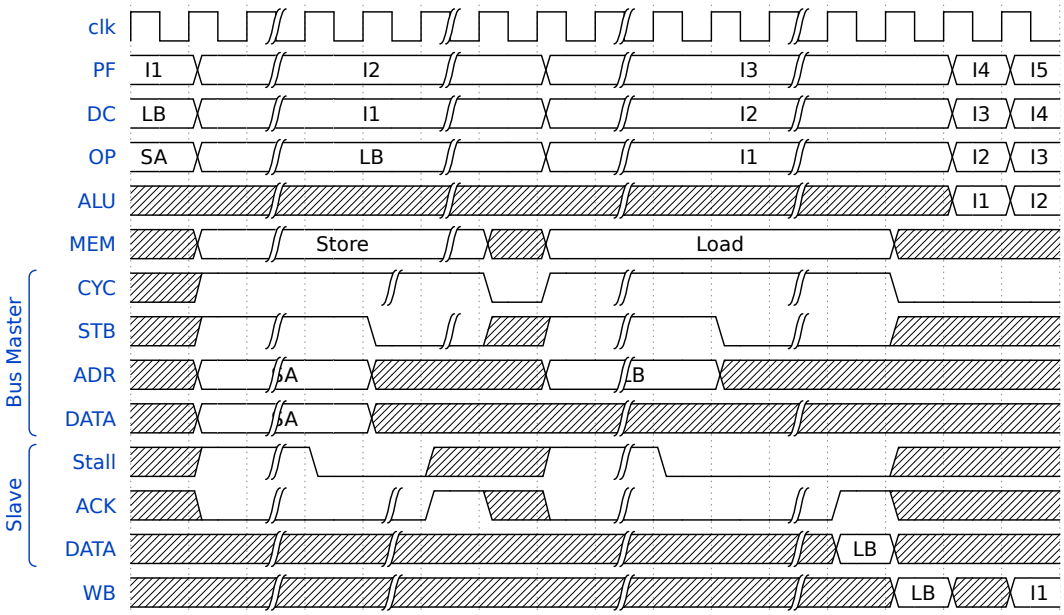

Figure 2.8: Pipeline handling of a store followed by a load instruction

The ZipCPU has the capability of supporting a form of burst memory access, often called pipelined memory access within this document due to its use of the Wishbone B4 pipelined access mode. When using this mode, the CPU may issue multiple loads or stores at a time, to the extent that all but the first take only a single clock. Doing this requires several conditions to be true:

1. Aall accesses within the burst must all be reads or all be writes,
2. All must use the same base register for their address, and
3. There can be no stalls or other instructions between memory access instructions within the burst.
4. Further, the immediate offset to memory must be either indentical or increasing by one address each instruction.

These conditions work well for saving or storing registers to the stack in a burst operation. Indeed, if you noticed, both Fig. 2.6 and Fig. 2.7 illustrated pipelined memory accesses. Beyond saving and restoring registers to the stack, the compiler does not optimize well (yet) for using this burst mode.

2.3 External Architecture

Having now described the CPU registers, instructions, and instruction formats, we now turn our attention to how the CPU interacts with the rest of the world. Specifically, we shall discuss how the bus is implemented, and the memory model assumed by the CPU.

2.3.1 Simplified Wishbone Bus

The bus architecture of the ZipCPU is that of a simplified, pipelined, WISHBONE bus built according to the B4 specification. Several changes have been made to simplify this bus. First, all unnecessary ancillary information has been removed. This includes the retry, tag, lock, cycle type indicator, and burst indicator signals. It also includes the select lines which would enable the CPU to act on less than 32–bit words. As a result all operations on the bus are 32–bit operations. The bus is neither little endian nor big endian. For this reason, all words are 32–bits. All instructions are also 32–bits wide. Everything has been built around the 32–bit word. Even the byte size (the size of the minimum addressable unit) is 32–bits. Second, we insist that all accesses be pipelined, and simplify that further by insisting that pipelined accesses not cross peripherals—although we leave it to the user to keep that from happening in practice. Finally, we insist that the wishbone strobe line be zero any time the cycle line is inactive. This makes decoding simpler in slave logic: a transaction is initiated whenever the strobe line is high and the stall line is low. For those peripherals that do not generate stalls, only the strobe line needs to be tested for access. The transaction completes whenever either the ACK or the ERR lines go high.

2.3.2 Memory Model

The memory model of the ZipCPU is that of a uniform 32–bit address space. The CPU knows nothing about which addresses reference on–chip or off-chip memory, or even which reference peripherals. Indeed, there is no indication within the CPU if a particular piece of memory can be cached or not, save that the CPU assumes any and all instruction words can be cached.

The one exception to this rule revolves around addresses beginning with `2'b11` in their high order word. These addresses are used to access a variety of optional peripherals that will be discussed more in Sec. 2.3.3, but that are only present within the `ZipSystem`. When used with the bare `ZipBones`, these addresses will cause a bus error.

The prefetch cache currently has no means of detecting an instruction that was changed, save by clearing the instruction cache. This may be necessary when loading programs into previously used memory, or when creating self–modifying code.

Should the memory management unit (MMU) be integrated into the ZipCPU, the MMU will be able to be configured to instruct the ZipCPU as to which addresses may be cached and which not.

This topic is discussed further in the linker section, Sec. 3.6.1 of the ABI chapter, Chap. 3.

2.3.3 ZipSystem

While the previous chapter describes a CPU in isolation, the ZipSystem includes a small minimum set of peripherals that can be tightly integrated into the CPU. These peripherals are shown in Fig. 2.9 and described here. They are designed to make the ZipCPU more useful in an Embedded Operating System environment.

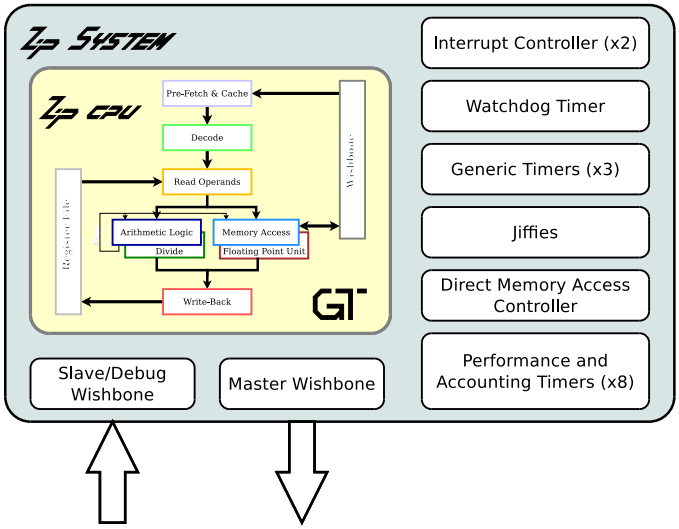

Figure 2.9: ZipSystem Peripherals

Interrupt Controller

Perhaps the most important peripheral within the ZipSystem is the interrupt controller. While the ZipCPU itself can only handle one interrupt, and has only the one interrupt state: disabled or enabled, the interrupt controller can make things more interesting.

The ZipSystem interrupt controller module supports up to 15 interrupts, all controlled from one register. Further, it has been designed so that individual interrupts can be enabled or disabled individually without having any knowledge of the rest of the controller setting. To enable an interrupt, write to the register with the high order global enable bit set and the respective interrupt enable bit set. No other bits will be affected. To disable an interrupt, write to the register with the high order global enable bit cleared and the respective interrupt enable bit set. To clear an interrupt, write a '1' to that interrupt's status pin. A zero written to the register has the sole effect of disabling the master interrupt enable bit.

As an example, suppose you wished to enable interrupt #4. You would then write to the register a `0x80100010` to enable interrupt #4 and to clear any past active state. When you later wish to disable this interrupt, you would write a `0x00100010` to the register. This both disables the interrupt and clears the active indicator. This also has the side effect of disabling all interrupts, so a second write of `0x80000000` may be necessary to re-enable any other interrupts.

The ZipSystem hosts two interrupt controllers: a primary and a secondary. The primary interrupt controller is the one that interrupts the CPU. It has six local interrupt lines, the rest coming from external interrupt sources. One of those interrupt lines to the primary controller comes from the secondary interrupt controller. This controller maintains an interrupt state for the process accounting counters, and any other external interrupts that didn't fit into the primary interrupt controller.

As a word of caution, because the interrupt controller is an external peripheral, and because memory writes take place concurrently with any following instructions, any attempt to clear interrupts on one instruction followed by an immediate Return to Userspace (`RTU`) instruction, may not have the effect of having interrupts cleared before the `RTU` instruction executes.

Counter

The Zip Counter is a very simple counter: it just counts. It cannot be halted. When it rolls over, it issues an interrupt. Writing a value to the counter just sets the current value, and it starts counting again from that value.

Eight counters are implemented in the ZipSystem for process accounting if the `INCLUDE_ACCOUNTING_COUNTERS` define is set within `cpudefs.v`. Four of those measure the performance of the system as a whole, four are used for measuring user CPU usage. This may change in the future, as nothing as yet uses these counters.

Timer

The Zip Timer is also very simple: it is a 31–bit counter that simply counts down to zero. When it transitions from a one to a zero it creates an interrupt.

Writing any non-zero value to the timer starts the timer. If the high order bit is set when writing to the timer, the timer becomes an interval timer and reloads its last start time on any interrupt. Hence, to mark seconds, one might set the 31–bits of the timer to the number of clocks per second and the top bit to one. Ever after, the timer will interrupt the CPU once per second–until a non–interrupt interval is set in the timer. This reload capability also limits the maximum timer value to $2^{31}-1$, rather than $2^{32}-1$.

Watchdog Timer

The watchdog timer has only two differences from the of the other timers. The first difference is that it is a one–shot timer. The second difference, though, is critical: the interrupt line from the watchdog timer is tied to the reset line of the CPU. Hence writing a '1' to the watchdog timer will always reset the CPU. To stop the Watchdog timer, write a '0' to it. To start it, write any other number to it—as with the other timers.

Bus Watchdog

There is an additional watchdog timer on the Wishbone bus of the ZipSystem. This timer, however, is hardware configured and not software configured. The timer is reset at the beginning of any bus transaction, and only counts clocks during such bus transactions. If the bus transaction takes longer than the number of counts the timer allots, it will raise a bus error flag to terminate the transaction. This is useful in the case of any peripherals that are misbehaving. If the bus watchdog terminates a bus transaction, the CPU may then read from its port to find out which memory location created the problem.

Aside from its unusual configuration, the bus watchdog is just another implementation of the fundamental timer described above–stripped down for simplicity.

Jiffies

This peripheral is motivated by the Linux use of 'jiffies' whereby a process can request to be put to sleep until a certain number of 'jiffies' have elapsed. Using this interface, the CPU can read the number of 'jiffies' from the peripheral (it only has the one location in address space), add the sleep length to it, and write the result back to the peripheral. The `zipjiffies` peripheral will record the value written to it only if it is nearer the current counter value than the last current waiting interrupt time. If no other interrupts are waiting, and this time is in the future, it will be enabled. (There is currently no way to disable a jiffie interrupt once set, other than to disable the interrupt line in the interrupt controller.) The processor may then place this sleep request into a list among other sleep requests. Once the timer expires, it would write the next Jiffy request to the peripheral and wake up the process whose timer had expired.

Indeed, the Jiffies register is nothing more than a glorified counter with an interrupt. Unlike the other counters, the internal Jiffies counter can only be read, never set. Writes to the jiffies register create an interrupt time. When the Jiffies register later equals the value written to it, an interrupt will be asserted and the register then continues counting as though no interrupt had taken place.

Finally, if the new value written to the Jiffies register is within the past 2^{31-1} clock ticks, the Jiffies register will immediately cause an interrupt and otherwise ignore the new request.

The purpose of this register is to support alarm times within a CPU. To set an alarm for a particular process N clocks in advance, read the current Jiffies value, add N, and write it back to the Jiffies register. The O/S must also keep track of values written to the Jiffies register. Thus, when an 'alarm' trips, it should be removed from the list of alarms, the list should be resorted, and the next alarm in terms of Jiffies should be written to the register–possibly for a second time.

Direct Memory Access Controller

The Direct Memory Access (DMA) controller can be used to either move memory from one location to another, to read from a peripheral into memory, or to write from a peripheral into memory all without CPU intervention. Further, since the DMA controller can issue (and does issue) pipeline wishbone accesses, any DMA memory move will by nature be faster than a corresponding program accomplishing the same move. To put this to numbers, it may take a program running on the CPU 18 clocks per word transferred, whereas this DMA controller can move one word in eight clocks–provided it has bus access[6] (The CPU gets priority over the bus, but once bus access is granted to the DMA peripheral, it will not be revoked mid–read or mid–write.)

When copying memory from one location to another, the DMA controller will copy in units of a given transfer length–up to 1024 words at a time. It will read that transfer length into its internal buffer, and then write to the destination address from that buffer.

When coupled with a peripheral, the DMA controller can be configured to start a memory copy when any interrupt line goes high. Further, the controller can be configured to issue reads from (or to) the same address instead of incrementing the address at each clock. The DMA completes once the total number of items specified (not the transfer length) have been transferred.

In each case, once the transfer is complete and the DMA unit returns to idle, the DMA will issue an interrupt.

[6] The pipeline cost of the DMA controller, including setup cost, is a minimum of $14 + 2N$ clocks.

2.4 Debug Interface

The ZipCPU supports an external debug port. Access to the port is the same as accessing a two register peripheral on a wishbone bus, so the basic interface is fairly simple. Using this interface, it is possible to both control the CPU, as well as read register values and current status from the CPU.

While a more detailed discussion will be reserved for Sec. 5.2, here we'll just discuss how it is put together. The debug interface allows a controller access to the CPU reset line, and a halt line. By raising the reset line, the CPU will be caused to clear it's cache, to clear any internal exception or error conditions, and then to start execution at the `RESET_ADDRESS`–just like a normal reboot. In a similar fashion, the debug interface allows you to control the `cpu_halt` line into the CPU. Holding this line high will hold the CPU in an externally halted state. Toggling the line low for one clock allows one to step the CPU by one instruction. Lowering the line causes the CPU to go. A final control wire, controlled by the debug interface, will force the CPU to clear its cache. All of these control wires are set or cleared from the debug control register.

The two debug command registers also make it possible to read and write all 32 registers within the CPU. In this fashion, a debugger can halt the CPU, investigate its state, and even modify registers so as to have the CPU restart from a different state.

Finally, without halting the CPU, the debug controller can read from any single register, and it can see if the CPU is still actively running, whether it is in user or supervisor modes, and whether or not it is sleeping. This alone is useful for detecting deadlocks or other difficult problems.

3.

Application Binary Interface

This chapter discusses not the CPU itself, but rather how the GCC and binutils toolchains have been configured to support the ZipCPU.

3.1 Executable File Format

ZipCPU executable files are stored in the Executable and Linkable Format (ELF), prior to being placed in flash, or whatever memory they will be executed from. All addresses within this format are ZipCPU addresses, referencing 32–bit quantities, whereas all offsets internal to the ELF file represent 8–bit quantities. Thus, when running the `zip-objdump` utility on a ZipCPU ELF file, the addresses are properly set.

The ZipCPU does not (yet) have a dynamic linker/loader. All linking is currently static, and done prior to run time.

3.2 Stack

Although nothing in the hardware requires this, the compiler back end implementation uses `R13` (also known as the `SP` register) as a stack register, and grows the stack from high addresses to lower addresses. That means that the stack will usually start out set to a very large value, such as one past the last RAM address, and it will grow to lower and lower values–hopefully never mixing with the heap. Memory at the current stack position is assumed to be allocated.

When creating a stack frame for a function, the compiler will subtract the size of the stack frame from the stack register. It will then store any registers used by the function, from `R5` to `R12` (including the link register `R0`) onto offsets given by the stack pointer plus a constant. If a frame pointer is used, the compiler uses `R12` (or `FP`) for this purpose. The frame pointer is set by moving the stack pointer plus an offset into `FP`. This `MOV` instruction effectively limits the size of any individual stack frame to $2^{12} - 1$ words.

Once a subroutine is complete, the frame is unwound. If the frame pointer, `FP` was used, then `FP` is copied directly to the stack pointer, `SP`. Registers are restored, starting with `R0` all the way to `R12` (`FP`). This also restores, and obliterates, the subroutine frame pointer. Once complete, a value is added to the stack pointer to return it to its original value, and a jump is made to the value located within `R0`.

3.3 Relocations

The ZipCPU binutils back end supports two several relocations, although the two most common are the 32–bit relocations for register load and long jump.

The first of these is for loading an arbitrary 32–bit value into a register. Such instructions are broken into a pair of `BREV` and `LDILO` instructions, and once the value of the parameter is known their immediates are filled in.

The second type of 32–bit relocation is for jumps to arbitrary addresses. These jumps are supported by the `LOD (PC),PC` instruction, followed by the 32–bit address to be filled in later by the linker. If the jump is conditional, then a conditional `LOD.`x `1(PC),PC` instruction is used, followed by a `BRA 1(PC),PC` and then the 32–bit relocation value.

If the branch distance is known and within reach, branches will be implemented with `ADD #,PC` instructions, possibly conditional, as necessary.

While other relocations are supported, they tend not to be used nearly as much as these two.

3.4 Call format

One feature of the ZipCPU is that it has no JSR instruction. Jumps to subroutine's therefore take three assembly instructions: The first is a `MOV .Lcall##(PC),R0`, which places the return address into R0. `.Lcall##` in this case is a label, where ## is a unique number filled in by the compiler. This instruction is followed by a `BRA subroutine` instruction. Finally, the third assembly "instruction" of any call sequence is the label `.Lcall##`.

While this works well in practice, GCC's implementation prevents such things as `JSR`'s followed by `BRA`'s from being combined together.

Finally, the first five operands passed to the subroutine will be placed into registers R1–R5. Any additional operands are placed upon the stack.

3.5 Built-ins

The ZipCPU ABI supports the a number of built in functions. The compiler maps these functions directly to assembly language equivalents, essentially providing the C programmer with access to several assembly language instructions. These are:

1. `zip_bitrev(int)` reverses the bits in the given integer, returning the result. This utilizes the internal `BREV` instruction, and is designed to be used with FFT's as necessary.

2. `zip_busy()` executes an `ADD -1,PC` function, essentially forcing the CPU into a very tight infinite loop.

3. `zip_cc()` returns the value of the current CC register. This may be used within both user and supervisor code to determine in which mode the CPU is within.

4. `zip_halt()` executes an `OR $SLEEP,CC` instruction to place the processor to sleep. If the processor is in supervisor mode, this halt's the processor.

5. `zip_rtu()` executes an `OR $GIE,CC` instruction. This will place the CPU into user mode, and has no effect if the CPU is already in user mode.

6. `zip_step()` executes an `OR $STEP|$GIE,CC` instruction. This will place the CPU into user mode in order to step one instruction, and then return to supervisor mode. It has no effect if the CPU is already in user mode.

7. `zip_system()` executes an `AND ~$GIE,CC` instruction to return the CPU to supervisor mode. This essentially executes a trap, setting the trap bit for the supervisor to examine. What this instruction does not do is arrange for the trap arguments to be placed into the registers `R1` through `R5`. Since this is a wholly inadequate solution, a function call may be made to an assembly routine that executes a trap if necessary.

8. `zip_wait()` executes a `$SLEEP|$GIE,CC` instruction. Unlike `zip_halt()`, this `zip_wait()` instruction places the CPU into a wait state regardless of whether or not the CPU is in supervisor mode or not. When this function, i.e. instruction, completes, it will leave the CPU in supervisor mode upon an interrupt having taken place.

   You may wish to set the user program counter prior to this instruction, as the prefetch unit will try to load instructions from the address contained within the user program counter. Attempts to read from addresses with sideeffects may not produce the desired outcome. However, once that cache fails (or succeeds), the CPU will have been put to sleep and will do no more.

9. `zip_restore_context(context *)` inserts the 32 assembly instructions necessary to copy all sixteen user registers to a memory region pointed to by the given context pointer, starting with `uR0` on up to `uPC`.

10. `zip_save_context(context *)` inserts the 32 assembly instructions necessary to copy all sixteen user registers to a memory region pointed to by the given context pointer argument, starting with `uR0` on up to `uPC`.

11. `zip_ucc()`, returns the value of the user CC register.

3.6 Linker Scripts

The ZipCPU makes no assumptions about its memory layout. The result, though, is that the memory layout of a given project is board specific. This is accomplished via a board specific linker script. This section will discuss some of the specifics of a ZipCPU linker script.

Because the ZipCPU uses a modified binutils package as part of its tool chain, the format for this linker script is defined by the GNU LD project within binutils. Further details on that format may be found within the GNU LD documentation within the binutils package.

This discussion will focus on those parts of the script specific to the ZipCPU.

3.6.1 Memory Types

Of the FPGA boards that the ZipCPU has been applied to, most of them have some combination of three types of memory: flash, block RAM, and Synchronous Dynamic RAM (SDRAM). Of these three, only the flash is non–volatile. The block RAM is the fastest, and the SDRAM the largest. While other memory types are available, such as files on an external media such as an SD card or a network drive, these three types have so far been sufficient for our purposes.

To support these memories, the linker script has three memory lines identifying where each memory exists on the bus, the size of the memory, and any protections associated with it. For example,

blkram (wx) : ORIGIN = 0x0008000, LENGTH = 0x0008000

specifies that there is a region of memory, called blkram, that can be read and written, and that programs can execute from. This section starts at address `0x8000` and extends for another `0x8000` words. The other memories are defined in a similar manner, with names `flash` and `sdram`.

Following the memory section, three specific symbols are defined: `_flash`, defining the beginning of flash memory, `_blkram`, defining the beginning of on–chip block RAM, and `_sdram`, defining the beginning of SDRAM. These symbols are used to make the bootloader's task easier.

3.6.2 The Entry Function

The ZipCPU has, as a parameter, a `RESET_ADDRESS`. It is important that this address contain a valid instruction (or more), since this is the first instruction the ZipCPU will execute. Traditionally, this address is also the first address in instruction memory as well.

To make this happen, the ZipCPU defines two additional segments: the `.start` and the `.boot` segments. The `.start` segment is to have nothing in it but the very initial startup code. This code needs to run from flash (or other ROM). By placing this segment at the very beginning of the ZipCPU's flash address space, and in particular at the first valid flash address, the ZipCPU will boot from this address. This is the purpose of the `.start` section.

The `.boot` section has a similar purpose. This section includes anything associated with the bootloader. It is a special section because, when loading from flash, the bootloader *cannot* be placed in RAM, but must be placed in flash–since it is the code that loads things from flash into RAM.

It may also make sense to place any code executed once only within flash as well. Such code may run slower than the main system code, but by leaving it in flash it can be kept from consuming any higher speed RAM. To do this, place this other code into the `.boot` section.

You may also find that large data structures that are best left in flash can also be placed into this `.boot` section as well for that purpose.

3.6.3 Bootloader Tags

The bootloader needs to know a couple things from the linker script. It needs to know what code/data to copy to block RAM from the flash, what code/data to copy to SDRAM, and finally what initial data area needs to be zeroed. Four additional pointers, set within a linker script, can define these regions.

1. `_kernel_image_start`

   This is the first location in flash containing data that the bootloader needs to move.

2. `_kernel_image_end`

   This is a pointer to one past the last location in block RAM to place things into. If this pointer is equal to `_kernel_image_start`, then no information is placed into block RAM.

3. `_sdram_image_start`

   This should be equal to `_kernel_image_end`. It is a pointer, within block RAM address space, of the first location to be moved into SDRAM. By adding the difference between `_sdram_image_start` and `_blkram` to the flash address in `_kernel_image_start`, the actual source address within the flash of the code/data that needs to be copied into SDRAM can be determined.

4. `_sdram_image_end`

   This is the ending address of any code/data to be copied into SDRAM. The distance between this pointer and `_sdram` should be the amount of data to be placed into SDRAM.

5. `_bss_image_end`

   The BSS segment contains data the starts with an initial value of zero. Such data are usually not placed in the executable file, nor are they placed into any flash image. This address points to the last location in SDRAM used by the BSS segment. The bootloader is responsible then for clearing the SDRAM between `_sdram_image_end` and `_bss_image_end`.

   The bootloader must also be robust enough to handle the cases where 1) there is no SDRAM, 2) there is no block RAM, and 3) where there is non requirement to move memory at all—such as when the program is placed into memory and started from there.

3.6.4 Other required linker symbols

Two other symbols need to be defined in the linker script, which are used by the startup code. These are:

1. `_top_of_stack`

   This is the address that the startup code will set the stack pointer to point to. It may be one past the last location of a RAM memory, whether block RAM or SDRAM.

2. `_top_of_heap`

   This is the first location past the end of the `.bss` segment. Equivalently, this is the address of the first unused piece of memory, or the location from whence to start any dynamic memory subsystem.

3.7 Loading ZipCPU Programs

There are two basic ways to load a ZipCPU program, depending upon whether or not the ZipCPU is active within the current configuration. If the ZipCPU is not a part of the current FPGA configuration, one need only write the flash and then switch configurations. It will be the CPU's responsibility to place itself in RAM then.

The more practical alternative is a little more involved, and there are several steps to it.

1. Halt the CPU by writing 0x0440 to the CPU control register. This both halts and resets the CPU. It then prevents both bus contention, while writing the new instructions to memory, as well as preventing the CPU from running some instructions from one program and other instructions from another.

2. Load the program into memory. For many programs this will involve loading the program into flash, and necessitate having and using a flash controller. The ZipCPU also supports being loaded straight into RAM address as well, as though the bootloader had completed it's task.
3. You may optionally, at this point, clear all of the CPUs registers, to make certain the reboot is clean.
4. Set the sPC register to the starting address.
5. Clear the instruction cache in order to force the CPU to reload its cache upon start.
6. Release the CPU by writing to the CPU debug control register a number between 0 and 31. This number will correspond to the register number of the register that can be "peeked" at while the CPU is running.

3.8 Starting a ZipCPU program

3.8.1 CRT0

Most computers have a section of code, conventionally called `crt0`, which loads a program into memory. On the ZipCPU, this code starts at `_start`. It is responsible for setting the stack pointer, calling the boot loader, and then calling the main entry function, `entry()`.

Because `_start` *must* be the first symbol in a program, and because that first symbol is located at the boot address for the CPU, the `_start` is placed into the `.start` segment. It is the only routine placed there.

On those CPU's that don't have enough logic space for a debugger, it may be useful to place a routine to dump any registers, stack values and/or kernel traces to an output routine at this time. That way, on any kernel fault, the kernel can be brought back up with a debug trace. This works because rebooting the CPU doesn't reset any register values save the `sCC` and `sPC`.

3.8.2 The Bootloader

As discussed in Sec. 3.6.3, the bootloader must be placed into flash if it is used. It can be a small C program (it need not be assembly, like `_start`), and it only needs to copy memory. First, it copies any memory from flash to block RAM. Second, it copies any necessary memory from flash to SDRAM. Then, it zeros any memory necessary in SDRAM (or block RAM, if there is no SDRAM).

These memory copies may be done with the DMA, or they may be done one–at–a time for a performance penalty.

3.8.3 Kernel Entry

After calling the boot loader, execution returns to the `_start` routine which then calls the main program entry function, `entry()`. No requirements are laid upon this entry function regarding where it must reside. The simplest place to put it is in Block RAM–and just to put all code and variables there. In reality, this entry function may easily be left in flash. It often doesn't need to run particularly fast, since there may easily be one–time setup functions that are independent of the programs main loop.

3.8.4 Kernel Main

If the kernel entry function, `entry()`, is placed in flash, it should call a separate function to run the main while loop once it has been set up. In this fashion, the main while loop may be kept in the fastest memory necessary (that it will fit within), to ensure good performance.

4.

Operation

This chapter will explore how to perform common tasks with the ZipCPU, offering examples in both C and assembly for those tasks.

4.1 CRT0

Of course, the one task that every CPU must do is start the CPU for other tasks. The ZipCPU is no different. This is the one ZipCPU task that must take place in assembly, since no assumptions can be made about the state of the ZipCPU upon entry. In particular, the stack pointer, SP, needs to be loaded with a valid memory location before any higher level language can work. Once that has taken place, it is then possible to call other higher level routines.

Table. 4.1 presents an example of one such initialization routine that first sets up the stack, then calls a bootloader routine. Upon completion, the initialization routine then calls the main entry point for the CPU. Should that main entry point ever return, a short routine following halts the CPU.

The example also highlights one optimization that didn't take place. Instead of placing the `_after_bootloader` address into `R0`, this script could have placed the `entry()` address into `R0`. Had it done so, the CPU would not have suffered the pipeline stalls associated with two long jumps: the first to `R0`, and the second to `entry`. Instead, it would have suffered such stalls only once: when jumping to `entry()`.

4.2 System High

The easiest and simplest way to run the ZipCPU is in the system high mode. In this mode, the CPU runs your program in supervisor mode from reboot to power down, and is never interrupted. You will need to poll the interrupt controller to determine when any external condition has become active. This mode is incredibly useful, and can handle many microcontroller–type tasks.

Even better, in system high mode, all of the user registers are available to the system high program as variables. Accessing these registers can be done in a single clock cycle, which would move them to the active register set or move them back. While this may seem like a load or store instruction, none of these register accesses will suffer from memory delays.

The one thing that cannot be done in supervisor mode is a wait for interrupt instruction. This, however, is easily rectified by jumping to a user task within the supervisors memory space, such as Tbl. 4.2.

```
; By starting our loader in the .start section, we guarantee through our
; linker script that these are the very first instructions the CPU sees.
    .section .start
    .global _start
; _start is to be placed at our reboot/reset address, so it will be
; called upon any reboot.
_start:
    ; The most important step: creating a stack pointer. The value
    ; _top_of_stack is created by the linker based upon the linker script.
    LDI _top_of_stack,SP
    ; We then call the bootloader to load our code into memory.
    MOV _after_bootloader(PC),R0
    BRA bootloader
_after_bootloader:
    ; Just in case the bootloader messed up the stack, we'll reset it here.
    LDI _top_of_stack,SP
    ; Finally, we call the entry function for the entire design.
    MOV _kernel_exit(PC),R0
    BRA entry
; The _kernel_exit routine that follows isn't strictly necessary,
; since the CPU should never return from the entry() function. However,
; since returning from such a function is valid C code, and just in case
; it does return, we provide this function as a fail safe to make sure
; the kernel halts upon completion.
_kernel_exit:
    HALT
    BRA _kernel_exit
```

Table 4.1: Setting up a stack frame and starting the CPU

```
supervisor_idle:
    ; While not strictly required, the following move helps to
    ; ensure that the prefetch doesn't try to fetch an instruction
    ; outside of the CPU's address space when it switches to user
    ; mode.
    MOV supervisor_idle_continue,uPC
    ; Put the processor into user mode and to sleep in the same
    ; instruction.
    OR $SLEEP|$GIE,CC
supervisor_idle_continue:
    ; Now, if we haven't done this inline, we need to return
    ; to whatever function called us.
    RETN
```

Table 4.2: Executing an idle from supervisor mode

There are some problems with this model, however. For example, even though the user registers can be accessed in a single cycle, there is currently no way to do so other than with assembly instructions.

An alternative to this approach is to use the `zip_wait()` built–in function. This places the ZipCPU into an idle/sleep mode to wait for interrupts. Because the supervisor puts the CPU to sleep, rather than the user, no user context needs to be set up.

4.3 A Programmable Delay

One common task in microcontrollers, whether in a user task or supervisor task, is to wait for a programmable amount of time. Using the ZipSystem, there are several peripherals that can be used to create such a delay. It can be done with one of the three timers, the jiffies, or even an off-chip ZipCounter.

Here, in Tbl. 4.3, we present one means of waiting for a programmable amount of time using a timer. If exact timing is important, you may wish to calibrate the method by subtracting from the counts number the counts it takes to actually do the routine. Otherwise, the timer is guaranteed to at least `counts` ticks.

Notice that the routine clears the PIC early on. While one might expect that this could be done in the instruction immediately before `zip_rtu()`, this isn't the case. The reason is a race condition created by the fact that the write to the PIC completes after the `zip_rtu()` instruction. As a result, you might find yourself with a zero delay simply because the timer had tripped some time earlier.

The routine is also careful not to clear any other interrupts beyond the timer interrupt, lest some other condition trip that the user was also waiting on.

```
#define EINT(A) (0x80000000|(A<<16)) // Enable interrupt A
#define DINT(A) (A<<16)              // Just disable the interrupts in A
#define DISABLEALL 0x7fff0000        // Disable all interrupts
#define CLEARPIC 0x7fff7fff          // Clears and disables all interrupts
#define SYSINT_TMA 0x10              // The Timer–A interrupt mask

void timer_delay(int nclocks) {
    // Clear the PIC. We want to exit from here on timer counts alone
    zip->pic = DISABLEALL|SYSINT_TMA;
    if (nclocks > 10) {
        // Set our timer to count down the given number of counts
        zip->tma = counts
        zip->pic = EINT(SYSINT_TMA);
        zip_wait();
        zip->pic = CLEARPIC;
    } // else anything less has likely already passed }
```

Table 4.3: Waiting on a timer

4.4 Traditional Interrupt Handling

Although the ZipCPU does not have a traditional interrupt architecture, it is possible to create the more traditional interrupt approach via software. In this mode, the programmable interrupt controller is used together with the supervisor state to create the illusion of more traditional interrupt handling.

To set this up, upon reboot the supervisor task:

1. Creates a (single) user context, a user stack, and sets the user program counter to the entry of the user task
2. Creates a task table of ISR entries
3. Enables the master interrupt enable via the interrupt controller, albeit without enabling any of the fifteen potential underlying interrupts.
4. Switches to user mode, as the first part of the while loop in Tbl. 4.4.

We can work through the interrupt handling process by examining Tbl. 4.4. First, remember, the CPU is always running either the user or the supervisor context. Once the supervisor switches to user mode, control does not return until either an interrupt, a trap, or an exception has taken place. Therefore, if neither the trap bit nor any of the exception bits have been set, then we know an interrupt has taken place.

It is also possible that an interrupt will occur coincident with a trap or exception. If this is the case, the subsequent `zip_rtu()` instruction will return immediately, since the interrupt has yet to be cleared.

```
while(true) {
    zip_rtu();
    if (zip_ucc() & CC_TRAPBIT) { // Here, we allow users to install ISRs, or
        // whatever else they may wish to do in supervisor mode.
        ...
    } else (zip_ucc() & (CC_BUSERR|CC_FPUERR|CC_DIVERR)) {
        // Here we handle any faults that the CPU may have encountered
        // The easiest solution is often to print a trace and reboot
        // the CPU.
        _start();
    } else {
        // At this point, we know an interrupt has taken place: Ask the programmable
        // interrupt controller (PIC) which interrupts are enabled and which are active.
        int picv = zip->pic;
        // Turn off all active interrupts
        // Globally disable interrupt generation in the process
        int active = (picv >> 16) & picv & 0x07fff;
        zip->pic = (active<<16);
        // We build a mask of interrupts to re-enable in picv.
        picv = 0;
        for(int i=0,msk=1; i<15; i++, msk<<=1) {
            if ((active & msk)&&(isr_table[i])) {
                // Here we call our interrupt service routine.
                tmp = (isr_table[i])();
                // The tricky part is that, because of how the PIC is built, the ISR cannot
                // re-enable its own interrupts without re-enabling all interrupts. Hence, we
                // look at the return value from the ISR to know if an interrupt needs to be
                // re-enabled.
                if (tmp)
                    picv |= msk;
            }
        }
        // Re-activate, but do not clear, all (requested) interrupts
        zip->pic = picv | 0x80000000;
    }
}
```

Table 4.4: Traditional Interrupt handling

```
idle_task:
    ; Wait for the next interrupt, then switch to supervisor task
    WAIT
    ; When we come back, it's because the supervisor wishes to
    ; wait for an interrupt again, so go back to the top.
    BRA idle_task
```

Table 4.5: Example Idle Task in Assembly

As Sec. 2.3.3 discusses, the top of the PIC register stores which interrupts are enabled, and the bottom stores which have tripped. (Interrupts may trip without being enabled, they just will not generate an interrupt to the CPU.) Our first step is to query the register to find out our interrupt state, and then to disable any interrupts that have tripped. To do that, we write a one to the enable half of the register while also clearing the top bit (master interrupt enable). This has the consequence of disabling any and all further interrupts, not just the ones that have tripped. Hence, upon completion, we re–enable the master interrupt bit again. Finally, we keep track of which interrupts have tripped.

Using the bit mask of interrupts that have tripped, we walk through all fifteen possible interrupts. If there is an ISR installed, we simply call it here. The ISR, however, cannot re–enable its interrupt without re-enabling the master interrupt bit. Thus, to keep things simple, when the ISR is finished it returns a boolean into R0 to indicate whether or not the interrupt needs to be re-enabled. Put together, this tells the kernel which interrupts to re–enable. As a final instruction, interrupts are re–enabled before returning continuing the while loop.

There you have it: the ZipCPU, with its non-traditional interrupt architecture, can still process interrupts in a very traditional fashion.

4.5 Idle Task

One task every operating system needs is the idle task, the task that takes place when nothing else can run. On the ZipCPU, this task is quite simple, and it is shown in assembly in Tbl. 4.5, or equivalently in C in Tbl. 4.6.

When this task runs, the CPU will fill up all of the pipeline stages up the ALU. The `WAIT` instruction, upon leaving the ALU, places the CPU into a sleep state where nothing more moves. Then, once an interrupt takes place, control passes to the supervisor task to handle the interrupt. When control passes back to this task, it will be on the next instruction. Since that next instruction sends us back to the top of the task, the idle task thus does nothing but wait for an interrupt.

This should be the lowest priority task, the task that runs when nothing else can. It will help lower the FPGA power usage overall—at least its dynamic power usage.

For those highly interested in reducing power consumption, the clock could even be disabled at this time–requiring only some small modifications to the core.

```
void idle_task(void) {
    while(true) { // Never exit
    // Wait for the next interrupt, then switch to supervisor task
    zip_wait();
    //
    // When we come back, it's because the supervisor wishes to
    // wait for an interrupt again, so go back to the top.
    }
}
```

Table 4.6: Example Idle Task in C

```
void memcpy(void *dest, void *src, int len) {
     for(int i=0; i<len; i++)
         *dest++ = *src++;
}
```

Table 4.7: Example Memory Copy code in C

4.6 Memory Copy

One common operation is that of a memory move or copy. This section will present several methods available to the ZipCPU for performing a memory copy, starting with the C code shown in Tbl. 4.7. Each successive example will further speed up the memory copying process.

This memory copy code can be translated in Zip Assembly as shown in Tbl. 4.8. This example points out several things associated with the ZipCPU. First, a straightforward implementation of a for loop is not the fastest loop structure. For this reason, we have placed the test to continue at the end. Second, all pointers are `void` pointers to arbitrary 32–bit data types. The ZipCPU does not have explicit support for smaller or larger data types, and so this memory copy cannot be applied at an 8–bit level. Third, notice that we can use `R4` without storing it, since the C ABI allows for subroutines to use `R1`–`R4` without saving them. This means that we can return from this subroutine using conditional jumps to `R0`.

Still, there's more that could be done. Suppose we wished to use the pipeline bus capability? We might then write something closer to Tbl. 4.9. This pipeline memory example, though, provides some neat things to discuss about optimizing code using the ZipCPU.

First, note that all of the loads and stores, except the three following `memcpy_finish`, are pipelined. To do this, we needed to unroll the copy loop by a factor of four. This means that each time through the loop, we can read and store four values. At 17 clocks to copy four values, that's roughly three times faster than our previous example. The down side, though, is that the loop now needs a final cleanup section where the last 0–3 values will be copied.

Second, note that we used our remaining length minus one as our loop variable. This was done so that the conditions set by subtracting four from our loop variable could be used without a

```
memcpy:
        ; R0 = return address, R1 = *dest, R2 = *src, R3 = LEN
        ; The following will operate in 6 (N = 0), or 2 + 12N clocks (N ≠ 0).
        CMP 0,R3
        JMP.Z R0        ; A conditional return
        ; No stack frame needs to be set up to use R4, since the compiler
        ; assumes R1-R4 may be used and changed by any subroutine
memcpy_loop:
        LOD (R2),R4
        ; (4 stalls, cannot be scheduled away)
        STO R4,(R1)     ; (4 schedulable stalls, has no impact now)
        ; Update our count of the number of remaining values to copy
        SUB 1,R3        ; This will be zero when we have copied our last
        JMP.Z R0        ; + 4 stalls, if taken
        ADD 1,R1        ; Implement the destination pointer
        ADD 1,R2        ; Implement the source pointer
        BRA memcpy_loop
        ; (1 stall on a BRA instruction)
```

Table 4.8: Example Memory Copy code in Zip Assembly, Unoptimized

separate compare. Speaking of the compare, note that we have chosen to use a branch if carry (BC) comparison, which is equivalent to a less–than unsigned comparison.

Third, notice how we packed four ALU instructions, two adds, a subtract, and a conditional branch, after the four store instructions. These instructions can complete while the memory unit is busy, preparing us to start the subsequent load without any further stalls (unless the memory is particularly slow.)

Next, you may wish to notice that the four memory loads within the loop are followed by the early branching instruction. As a result, the branch costs no extra clocks, and the time between the loads at the bottom of the loop is dominated by the load to store time frame.

Finally, notice how the comparison at the end has been stacked. By comparing against one, we can return when there are zero items left, or one item left, without needing a new comparison. Hence, zero to three separate values can be copied using only two compares.

However, this discussion wouldn't be complete without an example of how this memory operation would be made even simpler using the direct memory access controller. In that case, we can return to C with the code in Tbl. 4.10. For large memory amounts, the cost of this approach will scale at roughly 2 clocks per word transferred.

Notice how much simpler this memory copy has become to write by using the DMA. But also consider, the system has only one direct memory access controller. What happens if one task tries to use the controller when it is already in use by another task? The result is that the direct memory access controller may need some special protections to make certain that only one task uses it at a time—much like any other hardware peripheral.

```
; Upon entry, R0 = return address, R1 = *dest, R2 = *src, R3 = LEN
; Achieves roughly 32 + 17 ⌊N/4⌋ clocks, after the initial pipeline delay
memcpy_opt:
     CMP 4,R3            ; Check for small short lengths, len < 4
     BC memcpy_finish    ; Jump to the end if so
     SUB 3,SP            ; Otherwise, create a stack frame, storing the registers
     STO R5,(SP)         ; we will be using. Note that this is a pipelined store, so
     STO R6,1(SP)        ; subsequent stores only cost 1 clock.
     STO R7,2(SP)
     ADD 4,R2            ; Pre-Increment our pointers, for a 4-stage pipeline. This
     ADD 4,R1            ; also fills up the 3 of the 4 stall states following the
     SUB 5,R3            ; stores. Also, leave R3 as the number left minus one.
     LOD -4(R2),R4       ; Load the first four values into
     LOD -3(R2),R5       ; registers, using a pipelined load.
     LOD -2(R2),R6
     LOD -1(R2),R7
mcopy_next_four_chars:   ; Here's the top of our copy loop
     STO R4,-4(R1)       ; Store four values, using a burst memory operation.
     STO R5,-3(R1)       ; One clock for subsequent stores.
     STO R6,-2(R1)       ; None of these effect the flags, that were set when
     STO R7,-1(R1)       ; we last adjusted R3
     BC preend_memcpy    ; +4 stall cycles, but only when taken
     ADD 4,R1            ; ALU ops don't stall during stores, so
     ADD 4,R2            ; increment our pointers here.
     SUB 4,R3            ; Calculate whether or not we have a next round
     LOD -4(R2),R4       ; Preload the values for the next round
     LOD -3(R2),R5       ; Notice that these are also pipelined
     LOD -2(R2),R6       ; loads, as before.
     LOD -1(R2),R7       ; The four stall cycles, though, are concurrent w/ the branch.
     BRA mcopy_next_char ; Early branching avoids the full memory pipeline stall
preend_memcpy:
     ADD 1,R3            ; R3 is now the remaining length, rather than one less than it
     LOD (SP),R5         ; Restore our saved registers, since the remainder of the routine
     LOD 1(SP),R6        ; doesn't use these registers
     LOD 2(SP),R7        ;
     ADD 3,SP            ; Adjust the stack pointer back to what it was
memcpy_finish:           ; At this point, there are 0 ≤ R3< 4 words left
     CMP 1,R3            ; Check if any ops are remaining
     JMP.LT R0           ; Return now if nothing is left
     LOD (R1),R4         ; Load and store the first item
     STO R4,(R1)         ;
     JMP.Z R0            ; Return if that was our only value
     LOD 1(R1),R4        ; Load and store the second item (if necessary)
     STO R4,1(R1)
     CMP 2, R3
     JMP.LT R0
     LOD 2(R1),R4        ; Load and store the second item (if necessary)
     STO R4,2(R1)        ; LOD, STO, JMP R0 will cost 10 cycles
     JMP R0              ; Finally, we return
```

Table 4.9: Example Memory Copy code in Zip Assembly, Hand Optimized

```
#define DMACOPY 0x0fed0000 // Copy memory, largest chunk at a time possible

void memcpy_dma(void *dest, void *src, int len) {
     // This assumes we have access to the DMA, that the DMA is not
     // busy, and that we are running in system high mode ...
     zip->dma.len = len; // Set up the DMA
     zip->dma.rd = src;
     zip->dma.wr = dst;
     // Command the DMA to start copying
     zip->dma.ctrl= DMACOPY;
     // Note that we take two clocks to set up our PIC. This is
     // required because the PIC takes at least a clock cycle to clear.
     zip->pic = DISABLEALL|SYSINT_DMA;
     // Now that our PIC is actually clear, with no more DMA
     // interrupt within it, now we enable the DMA interrupt, and
     // only the DMA interrupt.
     zip->pic = EINT(SYSINT_DMA);
     // And wait for the DMA to complete.
     zip_wait();
}
```

Table 4.10: Example Memory Copy code using the DMA

```
void  *memset(void *s, int c, size_t n) {
      for(size_t i=0; i<n; i++)
            *s++ = c;
      return s;
}
```

Table 4.11: Example Memset code

4.7 Memset

Another example worth discussing is the `memset()` library function. A straightforward implementation of this function in C might look like Tbl. 4.11. The function is simple enough to handle compile into the assembling listing in Tbl. 4.12. Note that we grab `R4` as a local variable, so that we can maintain the source pointer in `R1` as our result upon return. This is valid, since the compiler assumes that `R1`–`R4` will be clobbered upon any function call and so they are not saved.

You can also see that this straight forward implementation costs about six clocks per value to be set.

Were we to pipeline the memory accesses, we might choose to unroll the loop and do something more like Tbl. 4.13. Note that, in this example as with the `memcpy` example, our loop variable is one less than the number of operations remaining. This is because the ZipCPU has no less than or equal comparison, but only a less than comparison. Further, because the length is given as an unsigned

```
; Upon entry, R0 = return address, R1 = s, R2 = c, R3 = len
; Cost: Roughly 4 + 6N clocks
memset:
    TST R3          ; Return immediately if len (R3) is zero
    JMP.Z R0
    MOV R1,R4       ; Keep our return value in R1, use R4 as a local
memset_loop:        ; Here, we know we have at least one more to go
    STO R2,(R4)     ; Store one value (no pipelining)
    SUB 1,R3        ; Subtract during the store
    JMP.Z R0        ; Return (during store) if all done
    ADD 1,R4        ; Otherwise increment our pointer
    BRA memset_loop ; and repeat
```

Table 4.12: Example Memset code, minimally optimized

```
; Upon entry, R0 = return address, R1 = s, R2 = c, R3 = len
; Cost: roughly 20 + 9 ⌊N/4⌋
memset_pipe:
    MOV     R1,R4           ; Make a local copy of *s, so we can return R1
    CMP     4,R3            ; Jump to non–unrolled section
    JMP.C   memset_pipe_tail
    SUB     1,R3            ; R3 is now one less than the number to finish
memset_pipe_unrolled:       ; Here, we know we have at least four more to go
    STO     R2,(R4)         ; Store our four values, pipelining our
    STO     R2,1(R4)        ; access across the bus
    STO     R2,2(R4)
    STO     R2,3(R4)
    SUB     4,R3            ; If there are zero left, this will be a -1 result
    JMP.C   prememset_pipe_tail   ; So we can use our LT condition
    ADD     4,R4            ; Otherwise increment our pointer
    BRA     memset_pipe_loop ; and repeat using an early branchable instruction
prememset_pipe_tail:
    ADD     1,R3            ; Return our counts left to the run number
memset_pipe_tail:           ; At this point, we have R3=0-3 remaining
    CMP     1,R3            ; If there's less than one left
    JMP.C   R0              ; then return early.
    STO     R2,(R4)         ; If we've got one left, store it
    STO.GT  R2,1(R4)        ; if two, do a burst store
    CMP     3,R3            ; Check if we have another left
    STO.Z   R2,2(R4)        ; and store it if so.
    JMP     R0              ; Return now that we are complete.
```

Table 4.13: Example Memset after loop unrolling, using pipelined memory ops

```
#define DMA_CONSTSRC 0x20000000 // Don't increment the source address
void * memset_dma(void *s, int c, size_t len) {
        // As before, this assumes we have access to the DMA, and that
        // we are running in system high mode ...
        zip->dma.len = len; // Set up the DMA
        zip->dma.rd = &c;
        zip->dma.wr = s;
        // Command the DMA to start copying, but not to increment the
        // source address during the copy.
        zip->dma.ctrl= DMACOPY|DMA_CONSTSRC;
        // Note that we take two clocks to set up our PIC. This is
        // required because the PIC takes at least a clock cycle to clear.
        zip->pic = DISABLEALL|SYSINT_DMA;
        // Now that our PIC is actually clear, with no more DMA
        // interrupt within it, now we enable the DMA interrupt, and
        // only the DMA interrupt.
        zip->pic = EINT(SYSINT_DMA);
        // And wait for the DMA to complete.
        zip_wait();
}
```

Table 4.14: Example Memset code, only this time with the DMA

quantity, we *only* have a less than comparison. By subtracting one from the loop variable, that's all our comparison needs to be–at least, until the end of the loop. For that, we jump to a section one instruction earlier and return our counts value to the true remaining length.

You may also notice that, despite the four possibilities in the end game, we can carefully rearrange the logic to only use two compares. The first compare tests against less than one and returns if there are no more sets left. Using the same compare, though, we can also know if we have one or more stores left. Hence, we can create a burst memory operation with one or two stores.

As one final example, we might also use the DMA for this operation, as with Tbl. 4.14. This is almost identical to the `memcpy` function above that used the DMA, save that the pointer for the value read is given to be the address of c, and that the DMA is instructed not to increment its source pointer. The DMA will still do `len` reads, so the asymptotic performance will never be less than $2N$ clocks per transfer.

4.8 String Operations

Perhaps one of the immediate questions most folks will have is, how does one handle string operations on a CPU that only handles 32–bit numbers? Here we offer a couple of possibilities.

The first possibility is the easy and natural choice: just define characters to be 32–bit numbers and ignore the upper 24 bits. This is the choice made by the compiler. Hence, if you compile a simple string compare function, such as Tbl. 4.15, string length function, such as Tbl. 4.16, or string copy function, such as Tbl. 4.17, this is what you will get.

```
int strcmp(const char *s1, const char *s2) {
    while(*s1 == *s2)
        s1++, s2++;
    return *s2 - *s1;
}
```

Table 4.15: Example string compare function

```
unsigned strlen(const char *s) {
        int ln = 0;
        while(*s++ != 0)
            ln++;
        return ln;
}
```

Table 4.16: Example string compare function

```
char * strcpy(char *dest, const char *src) {
       char *d = dest; // Make a working copy of the dest ptr
       do {
           *d++ = *src;
       } while(*src++);
       return dest;
}
```

Table 4.17: Example string copy function

```
void packstr(char *s) {
     char *d = s; // Pack our string in place
     int w;       // A holding word to pack things into
     int k=0;     // A count to know when to move to the next word
     while(*s) {
         w = (w<<8)|(*s & 0x0ff);
         // After four of these octets, write the result out
         if (((++k)&3)==0) *d++ = w;
     }
     // But what happens if we never got to the fourth octet
     // in our last word? We need to clean that up here.

     // First, shift the partial value all the way up
     w = (w<<(32-((k&3)<<3)); // Shift up the last word
     *d++ = w; // Store any remaining partial value
     // If we want to make sure our strings end in zero, we need
     // one more step:
     *d = 0; // Make sure string ends in a zero.
}
```

Table 4.18: String packing function

```
int pstrcmp(const char *s1, const char *s2) {
    while(*s1 == *s2)
        s1++, s2++;
    return *s2 - *s1;
}
```

Table 4.19: Packed string compare function

A little work with these functions, and you should be able to optimize them in a fashion similar to that with memcpy. This doesn't solve the fundamental problem, though, of why am I wasting 32–bits for 8–bit quantities?

An alternative would be to use a packed string structure. To pack a string, one might do something like Tbl. 4.18. Notice that our packed string places its first byte in the high order octet of our first word, that any excess octets in the last word are zeros, and that there remains a zero word following our string. With this packed string approach, compares and copies can proceed four times faster. As an example, Tbl. 4.19 presents a string compare function for a packed string. You'll notice that it doesn't look all that different from a string compare for a non-packed string. This is on purpose. Another example might be a string copy, which again, wouldn't look all that different. Getting the number of used 8–bit octets within a string is a touch more difficult. In that case, one might try something like Tbl. 4.20.

```
unsigned pstrlen(const char *s) {
        int ln = 0;
        while(*s++ != 0)
            ln+=4;
        if (ln) {
            // Touch up the length in case of an incomplete last word
            int lastval = s[-1];

            if ((lastval & 0x0ff)==0) ln--;
            if ((lastval & 0x0ffff)==0) ln--;
            if ((lastval & 0x0ffffff)==0) ln--;
        }
        return ln;
}
```

Table 4.20: Packed string subcharacter length function

4.9 Context Switch

Fundamental to any multiprocessing system is the ability to switch from one task to the next. In the ZipSystem, this is accomplished in one of a couple of ways. The first step is that an interrupt, trap, or exception takes place. This will pull the CPU out of user mode and into supervisor mode. At this point, the CPU needs to execute the following tasks:

1. Check for the reason, why did we return from user mode? Did the user execute a trap instruction, or did some other user exception such as a break, bus error, division by zero error, or floating point exception occur. That is, if the user process needs attending then we may not wish to adjust the context, check interrupts, or call the scheduler. Tbl. 4.21 shows the rudiments of this code, while showing nothing of how the actual trap would be implemented.

   You may also wish to note that the instruction before the first instruction in our context swap *must be* a return to userspace instruction. Remember, the supervisor process is re–entered where it left off. This is different from many other processors that enter interrupt mode at some vector or other. In this case, we always enter supervisor mode right where we last left.

2. Capture user accounting counters. If the operating system is keeping track of system usage via the accounting counters, those counters need to be copied and accumulated into some master counter at this point.

3. Preserve the old context. This involves recording all of the user registers to some supervisor memory structure, such as is shown in Tbl. 4.22. Since this task is so fundamental, the ZipCPU compiler back end provides the `zip_save_context(int *)` function.

4. Reset the watchdog timer. If you are using the watchdog timer, it should be reset on a context swap, to know that things are still working.

5. Interrupt handling. How you handle interrupts on the ZipCPU are up to you. You can activate a sleeping task if you like, or for smaller faster interrupt routines, such as copying a character

```
while(true) {
    // The instruction before the context switch processing must
    // be the RTU instruction that enacted user mode in the first
    // place. We show it here just for reference.
    zip_rtu();

    if (zip_ucc() & (CC_FAULT)) {
        // The user program has experienced an unrecoverable fault and must die.
        // Do something here to kill the task, recover any resources
        // it was using, and report/record the problem.
        ...
    } else if (zip_ucc() & (CC_TRAPBIT)) {
        // Handle any user request
        zip_restore_context(userregs);
        // If the request ID is in uR1, that is now userregs[1]
        switch(userregs[1]) {
        case x: // Perform some user requested function
            break;
        }
    }

}
```

Table 4.21: Checking for whether the user task needs our attention

```
save_context:
    SUB 1,SP        ; Function prologue: create a stack
    STO R5,(SP)     ; frame and save R5. (R1-R4 are assumed
    MOV uR0,R2      ; to be used and in need of saving. Then
    MOV uR1,R3      ; copy the user registers, four at a time to
    MOV uR2,R4      ; supervisor registers, where they can be
    MOV uR3,R5      ; stored, while exploiting memory pipelining
    STO R2,(R1)     ; Exploit memory pipelining:
    STO R3,1(R1)    ; All instructions write to same base memory
    STO R4,2(R1)    ; All offsets increment by one
    STO R5,3(R1)
    ...; Need to repeat for all user registers
    MOV uR12,R2     ; Finish copying ...
    MOV uSP,R3
    MOV uCC,R4
    MOV uPC,R5
    STO R2,12(R1)   ; and saving the last registers.
    STO R3,13(R1)   ; Note that even the special user registers
    STO R4,14(R1)   ; are saved just like any others.
    STO R5,15(R1)
    LOD (SP),R5     ; Restore our one saved register
    ADD 1,SP        ; our stack frame,
    JMP R0          ; and return
```

Table 4.22: Example Storing User Task Context

to or from a serial port or providing a sample to an audio port, you might choose to do the task within the kernel main loop. The difference may depend upon how you have your hardware set up, and how fast the kernel main loop is.

6. Calling the scheduler. This needs to be done to pick the next task to switch to. It may be an interrupt handler, or it may be a normal user task. From a priority standpoint, it would make sense that the interrupt handlers all have a higher priority than the user tasks, and that once they have been called the user tasks may then be called again. If no task is ready to run, run the idle task to wait for an interrupt.

   This suggests a minimum of four task priorities:

   (a) Interrupt handlers, executed with their interrupts disabled

   (b) Device drivers, executed with interrupts re-enabled

   (c) User tasks

   (d) The idle task, executed when nothing else is able to execute

7. Restore the new tasks context. Given that the scheduler has returned a task that can be run at this time, the user registers need to be read from the memory at the user context pointer and then placed into the user registers. An example of this is shown in Tbl. 4.23, Because this is such an important task, the ZipCPU GCC provides a built–in function, `zip_restore_context(int *)`, which can be used for this task.

8. Clear the userspace accounting registers. In order to keep track of per process system usage, these registers need to be cleared before reactivating the userspace process. That way, upon the next interrupt, we'll know how many clocks the userspace program has encountered, and how many instructions it was able to issue in those many clocks.

9. Return back to the top of our loop in order to execute `zip_rtu()` again.

```
restore_context:
    SUB 1,SP        ; Set up a stack frame
    STO R5,(SP)     ; and store a local register onto it.

    LOD (R1),R2     ; By doing four loads at a time, we are
    LOD 1(R1),R3    ; making sure we are using our pipelined
    LOD 2(R1),R4    ; memory capability.
    LOD 3(R1),R5
    MOV R2,uR1      ; Once the registers are loaded, copy them
    MOV R3,uR2      ; into the user registers that they need to
    MOV R4,uR3      ; be placed within.
    MOV R5,uR4
    ...; Need to repeat for all user registers
    LOD 12(R1),R2 ; Now for our last four registers ...
    LOD 13(R5),R3
    LOD 14(R5),R4
    LOD 15(R5),R5
    MOV R2,uR12     ; These are the special purpose ones, restored
    MOV R3,uSP      ; just like any others.
    MOV R4,uCC
    MOV R5,uPC
    LOD (SP),R5     ; Restore our saved register,
    ADD 1,SP        ; and the stack frame,
    JMP R0          ; and return to where we were called from.
```

Table 4.23: Example Restoring User Task Context

5.

Registers

This chapter covers the definitions and locations of the various registers associated with both the ZipSystem, and the ZipCPU contained within it. These registers fall into two separate categories: the registers belonging to the ZipSystem, and then the two debug port registers belonging to the CPU itself. In this chapter, we'll discuss the ZipSystem peripheral registers first, followed by the two ZipCPU registers.

5.1 ZipSystem Peripheral Registers

The ZipSystem maintains currently maintains 20 register locations, as shown in Tbl. 5.1. These

Name	Address	Width	Access	Description
PIC	0xc0000000	32	R/W	Primary Interrupt Controller
WDT	0xc0000001	32	R/W	Watchdog Timer
WBU	0xc0000002	32	R	Address of last bus timeout error
CTRIC	0xc0000003	32	R/W	Secondary Interrupt Controller
TMRA	0xc0000004	32	R/W	Timer A
TMRB	0xc0000005	32	R/W	Timer B
TMRC	0xc0000006	32	R/W	Timer C
JIFF	0xc0000007	32	R/W	Jiffies
MTASK	0xc0000008	32	R/W	Master Task Clock Counter
MMSTL	0xc0000009	32	R/W	Master Stall Counter
MPSTL	0xc000000a	32	R/W	Master Pre–Fetch Stall Counter
MICNT	0xc000000b	32	R/W	Master Instruction Counter
UTASK	0xc000000c	32	R/W	User Task Clock Counter
UMSTL	0xc000000d	32	R/W	User Stall Counter
UPSTL	0xc000000e	32	R/W	User Pre–Fetch Stall Counter
UICNT	0xc000000f	32	R/W	User Instruction Counter
DMACTRL	0xc0000010	32	R/W	DMA Control Register
DMALEN	0xc0000011	32	R/W	DMA total transfer length
DMASRC	0xc0000012	32	R/W	DMA source address
DMADST	0xc0000013	32	R/W	DMA destination address

Table 5.1: ZipSystem Internal/Peripheral Registers

Bit #	Access	Description
31	R/W	Master Interrupt Enable
30...16	R/W	Interrupt Enable lines
15	R	Current Master Interrupt State
15...0	R/W	Input Interrupt states, write '1' to clear

Table 5.2: Interrupt Controller Register Bits

registers are located in the CPU's address space, although in a special area of that space. Indeed, the area is so special, that the CPU decodes the address space location before placing the request onto the bus. For this reason, other containers for the CPU, such as the ZipBones which doesn't have these registers, will still create errors when they are referenced.

Here in this section, we'll walk through the definition of each of these registers in turn, together with any bit fields that may be associated with them, and how to set those fields.

5.1.1 Interrupt Controller(s)

Any CPU with only a single interrupt line, such as the ZipCPU, really needs an interrupt controller to give it access to more than the single interrupt. The ZipCPU is no different. When the ZipCPU is built as part of the ZipSystem, this interrupt controller comes integrated into the system.

Looking into the bits that define this controller, you can see from Tbl. 5.2, that the ZipCPU Interrupt controller has four different types of bits. The high order bit, or bit–31, is the master interrupt enable bit. When this bit is set, then any time an interrupt occurs the CPU will be interrupted and will switch to supervisor mode, etc.

Bits 30 ... 16 are interrupt enable bits. Should the interrupt line ever be high while enabled, an interrupt will be generated. Further, interrupts are level triggered. Hence, if the interrupt is cleared while the line feeding the controller remains high, then the interrupt will re–trip. To set one of these interrupt enable bits, one needs to write the master interrupt enable while writing a '1' to this the bit. To clear, one need only write a '0' to the master interrupt enable, while leaving this line high.

Bits 15...0 are the current state of the interrupt vector. Interrupt lines trip whenever they are high, and remain tripped until the input is lowered and the interrupt is acknowledged. Thus, if the interrupt line is high when the controller receives a clear request, then the interrupt will not clear. The incoming line must go low again before the status bit can be cleared.

As an example, consider the following scenario where the ZipCPU supports four interrupts, 3...0.

1. The Supervisor will first, while in the interrupts disabled mode, write a `32'h800f000f` to the controller. The supervisor may then switch to the user state with interrupts enabled.

2. When an interrupt occurs, the supervisor will switch to the interrupt state. It will then cycle through the interrupt bits to learn which interrupt handler to call.

3. If the interrupt handler expects more interrupts, it will clear its current interrupt when it is done handling the interrupt in question. To do this, it will write a '1' to the low order interrupt mask, such as writing a `32'h0000_0001`.

Bit #	Access	Description
31	R/W	Auto-Reload
30...0	R/W	Current timer value

Table 5.3: Timer Register Bits

Bit #	Access	Description
31...0	R	Current jiffy value
31...0	W	Value/time of next interrupt

Table 5.4: Jiffies Register Bits

4. If the interrupt handler does not expect any more interrupts, it will instead clear the interrupt from the controller by writing a `32'h0001_0001` to the controller.
5. Once all interrupts have been handled, the supervisor will write a `32'h8000_0000` to the interrupt register to re-enable interrupt generation.
6. The supervisor should also check the user trap bit, and possible soft interrupt bits here, but this action has nothing to do with the interrupt control register.
7. The supervisor will then leave interrupt mode, possibly adjusting whichever task is running, by executing a return from interrupt command.

5.1.2 Timer Register

Leaving the interrupt controller, we show the timer registers bit definitions in Tbl. 5.3. As you may recall, the timer just counts down to zero and then trips an interrupt. Writing to the current timer value sets that value, and reading from it returns that value. Writing to the current timer value while also setting the auto–reload bit will send the timer into an auto–reload mode. In this mode, upon setting its interrupt bit for one cycle, the timer will also reset itself back to the value of the timer that was written to it when the auto–reload option was written to it. To clear and stop the timer, just simply write a '32'h00' to this register.

5.1.3 Jiffies

The Jiffies register is first and foremost a counter. It counts up one on every clock. Reads from this register, as shown in Tbl. 5.4, always return the time value contained in the register.

The register accepts writes as well. Writes to the register set the time of the next Jiffy interrupt. If the next interrupt is between 0 and 2^{31} clocks in the past, the peripheral will immediately create an interrupt. Otherwise, the register will compare the new value against the currently stored interrupt value. The value nearest in time to the current jiffies value will be kept, and so the jiffies register will trip at that value. Prior values are forgotten.

Bit #	Access	Description
31...0	R/W	Current counter value

Table 5.5: Counter Register Bits

When the Jiffy counter value equals the value in its trigger register, then the jiffies peripheral will trigger an interrupt. At this point, the internal register is cleared. It will create no more interrupts unless a new value is written to it.

5.1.4 Performance Counters

The ZipCPU also supports several counter peripherals, mostly for the purpose of process accounting. These counters each contain a single register, as shown in Tbl. 5.5. Writes to this register set the new counter value. Reads read the current counter value.

These counters can be configured to count upwards upon any event. Using this capability, eight counters have been assigned the task of performance counting. Two sets of four registers are available for keeping track of performance. The first set tracks master performance, including both supervisor as well as user CPU statistics. The second set tracks user statistics only, and will not count in supervisor mode.

Of the four registers in each set, the first is a task counter that just counts clock ticks. The second counter is a prefetch stall counter, then an master stall counter. These allow the CPU to be evaluated as to how efficient it is. The fourth and final counter in each set is an instruction counter, which counts how many instructions the CPU has issued.

It is envisioned that these counters will be used as follows: First, every time a master counter rolls over, the supervisor (Operating System) will record the fact. Second, whenever activating a user task, the Operating System will set the four user counters to zero. When the user task has completed, the Operating System will read the timers back off, to determine how much of the CPU the process had consumed. To keep this accurate, the user counters will only increment when the GIE bit is set to indicate that the processor is in user mode.

5.1.5 DMA Controller

The final peripheral to discuss is the DMA controller. This controller has four registers. Of these four, the length, source and destination address registers should need no further explanation. They are full 32–bit registers specifying the entire transfer length, the starting address to read from, and the starting address to write to. The registers can be written to when the DMA is idle, and read at any time. The control register, however, will need some more explanation.

The bit allocation of the control register is shown in Tbl. 5.6. This control register has been designed so that the common case of memory access need only set the key and the transfer length. Hence, writing a 32'h0fed0000 to the control register will start any memory transfer. On the other hand, if you wished to read from a serial port (constant address) and put the result into a buffer every time a word was available, you might wish to write 32'h2fed8001–this assumes, of course, that you have a serial port wired to the zero bit of this interrupt control. (The DMA controller does not use the interrupt controller, and cannot clear interrupts.) As a third example, if you wished to write

Bit #	Access	Description
31	R	DMA Active
30	R	Wishbone error, transaction aborted. This bit is cleared the next time this register is written to.
29	R/W	Set to '1' to prevent the controller from incrementing the source address, '0' for normal memory copy.
28	R/W	Set to '1' to prevent the controller from incrementing the destination address, '0' for normal memory copy.
27 . . . 16	W	The DMA Key. Write a 12'hfed to these bits to start the activate any DMA transfer.
27	R	Always reads '0', to force the deliberate writing of the key.
26 . . . 16	R	Indicates the number of items in the transfer buffer that have yet to be written.
15	R/W	Set to '1' to trigger on an interrupt, or '0' to start immediately upon receiving a valid key.
14. . . 10	R/W	Select among one of 32 possible interrupt lines.
9. . . 0	R/W	Intermediate transfer length. Thus, to transfer one item at a time set this value to 1. To transfer the maximum number, 1024, at a time set it to 0.

Table 5.6: DMA Control Register Bits

to an external FIFO anytime it was less than half full (had fewer than 512 items), and interrupt line 3 indicated this condition, you might wish to issue a 32'h1fed8dff to this port.

5.2 Debug Port Registers

Accessing the ZipSystem via the debug port isn't as straight forward as accessing the system via the wishbone bus. The debug port itself has been reduced to two addresses, as outlined earlier in Tbl. 5.7.

Name	Address	Width	Access	Description
ZIPCTRL	0	32	R/W	Debug Control Register
ZIPDATA	1	32	R/W	Debug Data Register

Table 5.7: ZipSystem Debug Registers

Access to the ZipSystem begins with the Debug Control register, shown in Tbl. 5.8.

The first step in debugging access is to determine whether or not the CPU is halted, and to halt it if not. To do this, first write a '1' to the Command HALT bit. This will halt the CPU and place it into debug mode. Once the CPU is halted, the stall status bit will drop to zero. Thus, if bit 10 is high and bit 9 low, the debug port is open to examine the internal state of the CPU.

Bit #	Access	Description
31...14	R	External interrupt state. Bit 14 is valid for one interrupt only, bit 15 for two, etc.
13	R	CPU GIE setting
12	R	CPU is sleeping
11	W	Command clear PF cache
10	R/W	Command HALT, Set to '1' to halt the CPU
9	R	Stall Status, '1' if CPU is busy (i.e., not halted yet)
8	R/W	Step Command, set to '1' to step the CPU, also sets the halt bit
7	R	Interrupt Request Pending
6	R/W	Command RESET
5...0	R/W	Debug Register Address

Table 5.8: Debug Control Register Bits

At this point, the external debugger may examine internal state information from within the CPU. To do this, first write again to the command register a value (with command halt still high) containing the address of an internal register of interest in the bottom 6 bits. Internal registers that may be accessed this way are listed in Tbl. 5.9. Primarily, these "registers" include access to the entire CPU register set, as well as the internal peripherals. To read one of these registers once the address is set, simply issue a read from the data port. To write one of these registers or peripheral ports, simply write to the data port after setting the proper address.

In this manner, all of the CPU's internal state may be read and adjusted.

As an example of how to use this, consider what would happen in the case of an external break point. If and when the CPU hits a break point that causes it to halt, the Command HALT bit will activate on its own, the CPU will then raise an external interrupt line and wait for a debugger to examine its state. After examining the state, the debugger will need to remove the breakpoint by writing a different instruction into memory and by writing to the command register while holding the clear cache, command halt, and step CPU bits high, (32'hd00). The debugger may then replace the breakpoint now that the CPU has gone beyond it, and clear the cache again (32'h500).

To leave this debug mode, simply write a '32'h0' value to the command register.

Name	Address	Width	Access	Description
sR0	0	32	R/W	Supervisor Register R0
sR1	0	32	R/W	Supervisor Register R1
sSP	13	32	R/W	Supervisor Stack Pointer
sCC	14	32	R/W	Supervisor Condition Code Register
sPC	15	32	R/W	Supervisor Program Counter
uR0	16	32	R/W	User Register R0
uR1	17	32	R/W	User Register R1
uSP	29	32	R/W	User Stack Pointer
uCC	30	32	R/W	User Condition Code Register
uPC	31	32	R/W	User Program Counter
PIC	32	32	R/W	Primary Interrupt Controller
WDT	33	32	R/W	Watchdog Timer
WBUS	34	32	R	Last Bus Error
CTRIC	35	32	R/W	Secondary Interrupt Controller
TMRA	36	32	R/W	Timer A
TMRB	37	32	R/W	Timer B
TMRC	38	32	R/W	Timer C
JIFF	39	32	R/W	Jiffies peripheral
MTASK	40	32	R/W	Master task clock counter
MMSTL	41	32	R/W	Master memory stall counter
MPSTL	42	32	R/W	Master Pre-Fetch Stall counter
MICNT	43	32	R/W	Master instruction counter
UTASK	44	32	R/W	User task clock counter
UMSTL	45	32	R/W	User memory stall counter
UPSTL	46	32	R/W	User Pre-Fetch Stall counter
UICNT	47	32	R/W	User instruction counter
DMACMD	48	32	R/W	DMA command and status register
DMALEN	49	32	R/W	DMA transfer length
DMARD	50	32	R/W	DMA read address
DMAWR	51	32	R/W	DMA write address

Table 5.9: Debug Register Addresses

6.

Wishbone Datasheets

The ZipSystem supports two wishbone ports, a slave debug port and a master port for the system itself. These are shown in Tbl. 6.1 and Tbl. 6.2 respectively. I do not recommend that you connect

Description	Specification
Revision level of wishbone	WB B4 spec
Type of interface	Slave, Read/Write, single words only
Address Width	1–bit
Port size	32–bit
Port granularity	32–bit
Maximum Operand Size	32–bit
Data transfer ordering	(Irrelevant)
Clock constraints	Works at 100 MHz on a Basys–3 board, and 80 MHz on a XuLA2–LX25
Signal Names	Signal Name / Wishbone Equivalent i_clk / CLK_I i_dbg_cyc / CYC_I i_dbg_stb / (CYC_I)&(STB_I) i_dbg_we / WE_I i_dbg_addr / ADR_I i_dbg_data / DAT_I o_dbg_ack / ACK_O o_dbg_stall / STALL_O o_dbg_data / DAT_O

Table 6.1: Wishbone Datasheet for the Debug Interface

these together through the interconnect, since 1) it doesn't make sense that the CPU should be able to halt itself, and 2) it helps to be able to reboot the CPU in case something has gone terribly wrong and the CPU is stalling the entire interconnect. Rather, the debug port of the CPU should be accessible regardless of the state of the master bus.

You may wish to notice that neither the `LOCK` nor the `RTY` (retry) wires have been connected to the CPU's master interface. If necessary, a rudimentary `LOCK` may be created by tying this wire to the `wb_cyc` line. As for the `RTY`, all the CPU recognizes at this point are bus errors—it cannot tell the difference between a temporary and a permanent bus error. Therefore, one might logically OR the bus error and bus retry flags on input to the CPU if necessary.

Description	Specification
Revision level of wishbone	WB B4 spec
Type of interface	Master, Read/Write, single cycle or pipelined
Address Width	(ZipSystem parameter, can be up to 32–bit bits)
Port size	32–bit
Port granularity	32–bit
Maximum Operand Size	32–bit
Data transfer ordering	(Irrelevant)
Clock constraints	Works at 100 MHz on a Basys–3 board, and 80 MHz on a XuLA2–LX25
Signal Names	Signal Name / Wishbone Equivalent i_clk / CLK_O o_wb_cyc / CYC_O o_wb_stb / (CYC_O)&(STB_O) o_wb_we / WE_O o_wb_addr / ADR_O o_wb_data / DAT_O i_wb_ack / ACK_I i_wb_stall / STALL_I i_wb_data / DAT_I i_wb_err / ERR_I

Table 6.2: Wishbone Datasheet for the CPU as Master

The final simplification made of the standard wishbone bus B4 specification, is that the strobe lines are assumed to be zero in any slave if CYC_I is zero, and the master is responsible for ensuring that STB_O is never true when CYC_O is true in order to make this work. All of the ZipCPU and ZipSystem masters and peripherals have been created with this assumption. Converting peripherals that have made this assumption to work with masters that don't guarantee this property is as simple as anding the slave's CYC_I and STB_I lines together. No change needs to be made to any ZipCPU master, however, in order to access any peripheral that hasn't been so simplified.

7.

Clocks

This core has now been tested and proven on the Xilinx Spartan 6 FPGA as well as the Artix–7 FPGA. I hesitate to suggest that the core can run faster than 100 MHz, since I have had struggled

Name	Source	Rates (MHz)		Description
		Max	Min	
i_clk	External	100 MHz		System clock, Artix–7/35T
		80 MHz		System clock, Spartan 6

Table 7.1: List of Clocks

with various timing violations to keep it at 100 MHz. So, for now, I will only state that it can run at 100 MHz.

On a SPARTAN 6, the clock can run successfully at 80 MHz.

A second Artix–7 design on the Digilent's Arty board is limited to 81.25 MHz by the memory interface generated core used to access SDRAM.

8.

I/O Ports

This chapter presents and outlines the various I/O lines in and out of the ZipSystem. Since the ZipCPU needs to be a component of such a larger part, this makes sense.

The I/O ports to the ZipSystem may be grouped into three categories. The first is that of the master wishbone used by the CPU, then the slave wishbone used to command the CPU via a debugger, and then the rest. The first two of these were already discussed in the wishbone chapter. They are listed here for completeness in Tbl. 8.1 and 8.2 respectively.

There are only four other lines to the CPU: the external clock, external reset, incoming external interrupt line(s), and the outgoing debug interrupt line. These are shown in Tbl. 8.3. The clock line was discussed briefly in Chapt. 7. The reset line is an active high reset. When asserted, the CPU will start running again from its `RESET_ADDRESS` in memory. Further, depending upon how the CPU is configured and specifically based upon how the `START_HALTED` parameter is set, the CPU may or may not start running automatically following a reset. The `i_ext_int` bus is for set of external interrupt lines to the ZipSystem. This line may actually be as wide as 16 external interrupts, depending upon the setting of the `EXTERNAL_INTERRUPTS` parameter. Finally, the ZipSystem produces one external interrupt whenever the entire CPU halts to wait for the debugger.

The I/O lines to the ZipBones package are identical to those of the ZipSystem, with the only exception that the ZipBones package has only a single interrupt line input. This means that the ZipBones implementation practically depends upon an external interrupt controller.

Port	Width	Direction	Description
`o_wb_cyc`	1	Output	Indicates an active Wishbone cycle
`o_wb_stb`	1	Output	WB Strobe signal
`o_wb_we`	1	Output	Write enable
`o_wb_addr`	32	Output	Bus address
`o_wb_data`	32	Output	Data on WB write
`i_wb_ack`	1	Input	Slave has completed a R/W cycle
`i_wb_stall`	1	Input	WB bus slave not ready
`i_wb_data`	32	Input	Incoming bus data
`i_wb_err`	1	Input	Bus Error indication

Table 8.1: CPU Master Wishbone I/O Ports

Port	Width	Direction	Description
i_dbg_cyc	1	Input	Indicates an active Wishbone cycle
i_dbg_stb	1	Input	WB Strobe signal
i_dbg_we	1	Input	Write enable
i_dbg_addr	1	Input	Bus address, command or data port
i_dbg_data	32	Input	Data on WB write
o_dbg_ack	1	Output	Slave has completed a R/W cycle
o_dbg_stall	1	Output	WB bus slave not ready
o_dbg_data	32	Output	Incoming bus data

Table 8.2: CPU Debug Wishbone I/O Ports

Port	Width	Direction	Description
i_clk	1	Input	The master CPU clock
i_rst	1	Input	Active high reset line
i_ext_int	1...16	Input	Incoming external interrupts, actual value set by implementation parameter. This is only ever one for the ZipBones implementation.
o_ext_int	1	Output	CPU Halted interrupt

Table 8.3: I/O Ports

9.

Initial Assessment

Having now worked with the ZipCPU for a while, it is worth offering an honest assessment of how well it works and how well it was designed. At the end of this assessment, I will propose some changes that may take place in a later version of this ZipCPU to make it better.

9.1 The Good

- The ZipCPU was designed to be a simple and light weight CPU. It has achieved this end nicely. The proof of this is the full multitasking operating system built for Digilent's CMod S6 board, based around a very small Spartan 6/LX4 FPGA.

  As a result, the ZipCPU also makes a good starting point for anyone who wishes to build a general purpose CPU and then to experiment with building and adding particular features. Modifications should be simple enough.

  Indeed, a non–pipelined version of the bare ZipBones (with no peripherals) has been built that only uses 1.3k 6–LUTs. When using pipelining, the full cache, and all of the peripherals, the ZipSystem can take up to 4.5 k LUTs. Where it fits in between is a function of your needs.

  A new implementation using an iCE40 FPGA suggests that the ZipCPU will fit within the 4k 4–way LUTs of the iCE40 HK4X FPGA, but only just barely.

- The ZipCPU was designed to be an implementable soft core that could be placed within an FPGA, controlling actions internal to the FPGA. It fits this role rather nicely. It does not fit the role of a general purpose CPU replacement very well: it has no octet level access, no double–precision floating point capability, neither does it have vector registers and operations. However, it was never designed to be such a general purpose CPU but rather a system within a chip.

- The extremely simplified instruction set of the ZipCPU was a good choice. Although it does not have many of the commonly used instructions, PUSH, POP, JSR, and RET among them, the simplified instruction set has demonstrated an amazing versatility. I will contend therefore and for anyone who will listen, that this instruction set offers a full and complete capability for whatever a user might wish to do with two exceptions: bytewise character access and accelerated floating-point support.

- This simplified instruction set is easy to decode.

- The simplified bus transactions (32-bit words only) were also very easy to implement.

- The burst load/store approach using the wishbone pipelining mode is novel, and can be used to greatly increase the speed of the processor.
- The novel approach to interrupts greatly facilitates the development of interrupt handlers from within high level languages.

  The approach involves a single interrupt "vector" only, and simply switches the CPU back to the instruction it left off at. By using this approach, interrupt handlers no longer need careful assembly language scripting in order to save their context upon any interrupt.

  At the same time, if most modern systems handle interrupt vectoring in software anyway, why maintain complicated hardware support for it?
- My goal of a high rate of instructions per clock may not be the proper measure of this CPU. For example, if instructions are being read from a SPI flash device, such as is common among FPGA implementations, these same instructions may suffer stalls of between 64 and 128 cycles per instruction just to read the instruction from the flash. Executing the instruction in a single clock cycle is no longer the appropriate measure. At the same time, it should be possible to use the DMA peripheral to copy instructions from the FLASH to a temporary memory location, after which they may be executed at a single instruction cycle per access again.
- Both GCC and binutils back ends exist for the ZipCPU.

9.2 The Not so Good

- The CPU has no octet (character) support. This is both good and bad. Realistically, the CPU works just fine without it. Characters can be supported as subsets of 32-bit words without any problem. Practically, though, this creates two problems. The first is that it makes porting code from non-ZipCPU platforms to the ZipCPU is difficult–especially anything that depends upon the existence of `*int8_t`, `*int16_t`, the size difference between `sizeof(int)=4*sizeof(char)`, or that tries to create unions with characters and integers and then attempts to reference the address of the characters within that union.

  The second problem is that peripherals that depend upon character support on the bus may need to be rewritten to work on a 32–bit bus.
- The ZipCPU does not (yet) support a data cache. One is currently under development.

  The ZipCPU compensates for this lack via its burst memory capability. Further, performance tests using Dhrystone suggest that the ZipCPU is no slower than other processors containing a data cache.
- Many other instruction sets offer three operand instructions, whereas the ZipCPU only offers two operand instructions. This means that it may take the ZipCPU more instructions to do many of the same operations. The good part of this is that it gives the ZipCPU a greater amount of flexibility in its immediate operand mode, although that increased flexibility isn't necessarily as valuable as one might like.

  The impact of this lack of three operand instructions is application dependent, but does not appear to be too severe.

- The ZipCPU doesn't support out of order execution.

  I suppose it could be modified to do so, but then it would no longer be the "simple" and low LUT count CPU it was designed to be.

- Although switching to an interrupt context in the ZipCPU design doesn't require a tremendous swapping of registers, in reality it still does–since any task swap (such as swapping to a task waiting on an interrupt) still requires saving and restoring all 16 user registers. That's a lot of memory movement just to service an interrupt.

  This isn't nearly as bad as it sounds, however, since most RISC architectures have 32 registers that will need to be swapped upon any context swap.

- The ZipCPU is by no means generic: it will never handle addresses larger than 32-bits (4GW or 16GB) without a complete and total redesign. This may limit its utility as a generic CPU in the future, although as an embedded CPU within an FPGA this isn't really much of a restriction.

- While a toolchain does exist for the ZipCPU, it isn't yet fully featured. The ZipCPU has no support for soft floating point arithmetic, nor does it have support for several standard library functions. Indeed, full C library support and gdb support are still lacking.

9.3 The Next Generation

This section could also be labeled as my "To do" list. It outlines where you may expect features in the future. Currently, there are five primary items on my to do list:

1. Soft Floating Point capability

   The lack of any floating point capability, either hard or soft, makes porting math software to the ZipCPU difficult. Simply building a soft floating point library will solve this problem.

2. A C library.

   The lack of octet support has so far prevented the porting of newlib to the ZipCPU platform. In the end, it may mean that any C library implementation for the ZipCPU may be subtly different from any you are familiar with.

3. A data cache

   A preliminary data cache implemented as a write through cache has been developed. Adding this into the CPU should require few changes internal to the CPU. I expect future versions of the CPU will permit this as an option.

4. A Memory Management Unit

   The first version of such an MMU has already been written. It is available for examination in the ZipCPU repository. This MMU exists as a peripheral of the ZipCPU. Integrating this MMU into the ZipCPU will involve slowing down memory stores so that they can be accomplished synchronously, as well as determining how and when particular cache lines need to be invalidated.

5. An integrated floating point unit (FPU)

   Why a small scale CPU needs a hefty floating point unit, I'm not certain, but many application contexts require the ability to do floating point math.