Register Oriented Instruction Set
Rois24_24 variant

James Brakefield

Contents

1. Instruction Set

Register to register	**XXXXXX DDDDDD RRRRRR SSSSSS**
Immediate1	**XXXXXX DDDDDD RRRRRR sNNNNN**
Immediate2	**XXXXXX DDDDDD sNNNNN NNNNNN**
Immediate3	**XXXXXX sNNNNN NNNNNN NNNNNN**

X: Op-code D: Result register
R: 1st operand register S: 2nd operand register
N: immediate operand s: N's sign (2's complement)

2. Op-codes **XXXXXX DDDDDD RRRRRR SSSSSS**

Register 0 always reads as zero
PFX instruction loads 18-bits into prefix register. Used for concatenation by next sN instruction
Three address arithmetic:

ADD	Add
SUB	Subtract
ADC	Add with carry
SBC	Subtract with borrow
AND	And
ANDN	And with second operand complemented
OR	Or
XOR	Exclusive Or
MUL	Multiply, least significant 24 bits
MULU	Multiply, most significant 24 bits unsigned
MULUS	Multiply, most significant 24 bits signed
DIV	Divide, unspecified
FADD	Floating point add
FSUB	Floating point subtract
FMUL	Floating point multiple
FDIV	Floating point divide

Immediates (immediate 1 format):

ADDI, ADCI, ANDI, ORI, XORI, MULI,

LSRI	Logical shift right
ASRI	Arithmetic shift right
LSLI	Logical shift left
RORI	Rotate right

Immediate (immediate 2 format):
- LDI Load register with sNNNNN NNNNNN

Immediate (immediate 3 format):
- PFX Load prefix register with 18 bits from sNNNNN NNNNNN NNNNNN

Load/Store:
- LD Load register D with memory location R+S
- ST Store register D at memory location R+S

Load/Store (immediate 1 format):
- LDsN Load register D with memory location R+sNNNNN
- STsN Store register D at memory location R+sNNNNN
- IN Place contents of input port R+sNNNNN into D
- OUT Transfer D to output port R+sNNNNN

Relative branches (immediate2 format):
- BRCC Branch to PC+sNNNNNNNNNNN if condition DDDDDD true
- CALLR Branch to PC+sNNNNNNNNNNN and save PC at register D

Three address branches:
- CALL Branch to address R+S, save PC+1 at register D
- CALLsN Branch to address R+sNNNNN, save PC+1 at register D
- JMPCC Jump to R+S if condition DDDDDDD true
- JMPCCsN Jump to R+sNNNNN if condition DDDDDDD true

2. Variations

The main constraints are LUT RAM for the register file(s) and Block RAM for program and data memory. In the larger memory space versions block RAM becomes a cache for main memory (which is external to the FPGA). LUT RAM for a single register file is constrained to have a single write port and may have as many read ports as needed.

The variation dimensions are RTL description (pipeline-depth, data-path, control signals), instruction encoding (op-codes, modes, number of register fields), instruction bit size (24, 16 or 12-bit) and addressable unit (8-bit, 12-bit or word).

All the variations listed below assume that one or more instructions are processed per clock. The single stage pipe versions of load and store instructions take two clocks. Two stage pipe versions of conditional instructions take two clocks if the condition code is modified by the pervious instruction. To the extent that they implemented; multiply, divide and floating-point instructions may take more than two clock cycles depending on the pipeline implementation and result delay interlocks.

2.1. Minimal implementation (MI)

Only a few instructions are implemented and the program is expressed by a case statement which allows further optimization (program and instruction logic get folded together, optimization includes removing unused instructions and modes)[1]. Fmax very good and LUT count is minimal.

[1]Further expansion of the "case statement as the program" approach is covered in the paper: "Lutiac – Small Soft Processors for Small Programs", Galloway and Lewis, 2010.
http://www.eecg.utoronto.ca/~jayar/FPGAseminar/2010/lutiac---small-processors-2.html

2.2. Basic implementation (BI)

The basic architecture is a single stage pipe, triple port LUT RAM, single port block RAM. Instruction implementation is a simple case statement, one entry for each op-code. Then there is a second case statement to handle the second clock cycle for memory read/write. Program typically in block RAM.

Synthesis results in many adders and multiplexors which get optimized by translate & map. Clock speed is good and LUT count is high.

The great advantage of BI is the ease of adding or removing instructions (one can simply comment out unused instructions).

2.3. Data path with control signals

Number of signal names increases and debugging is more difficult.

2.3.1. Many adders, multiplexors mostly after the adders

Variation is unexplored at this time. Variation needed for comparison against DP. Possibly superior Fmax at small increase in LUT count.

2.3.2. Few adders, multiplexors on both sides of adders (DP)

Preliminary indications are that LUT count is dramatically lower than that of BI. Good control over synthesis results.

2.4. Two stage pipeline

The case statement for instruction specification now generates control signals rather than data operations.

2.4.1. With single port block RAM

Not likely to be viable.

2.4.2. With dual port block RAM

This variation is expected to give the best performance metric (Instruction work[2] per LUT per millisecond). Use of more than two pipe stages likely to give marginal improvement to Fmax and result in increased complexity and LUT count.

2.5. Five bit register designators

2.5.1. Additional instruction bits

With three five bit register designators there is room for nine instruction bits. Planned usage for the additional bits are: return bit and condition code save bit. The third bit remains for six-bit immediates. The condition code save bit is the MSB of the D register field and return bit is the MSB of the R register field.

2.5.2. Register designation recode

Rather than have 32 fixed location registers it is possible to provide a number of register offsets that correspond to: return address stack, data stack(s) and frame pointer. Thus one can specify a register relative to these pointers.

[2] Instruction work is the average amount of computation done by a single instruction, typically normalized to a VAX MIPS (e.g. the Drystone MIPS or the CoreMark). The target performance metric value is several hundred. To a first approximation instruction work is pro-rated by word size: 8:33%, 16:67%, 24:83%, 32:100%, 48:150%, 64:200% and by average clocks per instruction. Further pro-rating if instruction set is weak, average or strong.

Additionally it is useful to specify modification of the offset pointers. Thus an instruction can specify stack pops (from a source stack) and pushes (to a destination stack).

2.6. Code compression

The MIPS instruction set uses up to three five bit register designators and a variable size instruction. Here, the baseline instruction is three six bit register designators and a six bit op-code. By being 75% of the MIPS instruction size (32-bits), it is 33% more compact and within the realm of "densely coded" instruction sets[3].

2.7. Addressable unit size

2.7.1. 12-bit

Original design is for 12, 24 or 48 bit word sizes. With 12-bit addressability the functionality closely matches 8/16/32 uP. Practical considerations may necessitate 8-bit addressability (UTF-8 in particular).

2.7.2. 8-bit

One can dispense with 12-bit items being addressable. Thus instructions would be either 24 or 16-bit and memory would be 8, 16, 32 or 64-bits wide. Memory would be in two banks with separate address adders for each bank. Instruction readout for 24-bit instructions requires a shift circuit to align the instruction.

2.8. LUT RAM word size

2.8.1. 12-bit

The intent of a 4K memory space with 12-bit words is small micro-controllers.

2.8.1. 24-bit

The intent of a 16M item memory space is embedded systems with program and data that fit. There is a cost savings over 32-bit uP.

2.8.2. 48-bit

The intent of a 48-bit uP is all applications short of very large super-computers. Its 48-bit floating-point is adequate for many applications. It holds the sweet spot between 32-bit and 64-bit uP.

2.8.3. 16-bit

TBD

2.8.4. 32-bit

The 32-bit uP provides 4GB of address space, sufficient for most general purpose computing including full operating systems and large libraries of code.

3. Implementation decisions

3.1. Go for a minimal design

There are arguments for going for either the smallest possible design or going for the highest performance design. The figure of merit is DMIPS per LUT. Or equivalently, DMIPS per unit silicon area, which factors in the area taken by multiplier/DSP units and block RAMs. Of course LUTs from

[3] PDP-11, x86, MSP430 and ARM Thumb instruction sets are considered some of the densest.

different FPGA families are not necessarily equivalent. The ultimate figure of merit is DMIPS per dollar using the prorated chip area.

The DMIPS per dollar and DMIPS per unit silicon area metrics also allow comparison with FPGAs with hard core processors such as the PowerPC, ARM Cortex A9 or Cortex A3, or with non-FPGA embedded processors.

Naturally some applications call for very little data RAM or instruction memory. Other applications call for a full 32-bit processor with floating-point, caches and/or memory management. Interestingly enough there are examples of high performance FPGA soft core processors at the minimal level, at the single processor pipelined level and at the multi-processor/barrel approach. Thus it is possible to get a high figure of merit for any of the three design approaches.

Given that **rois24_24uP** is a new architecture, a simple or minimal implementation is the lowest risk and fastest to implement choice. A minimal implementation also offers the greatest insight into LUT count versus instruction set complexity or cycle time.

3.2. Use a single pipeline stage

This mode of operation uses LUT RAM with asynchronous read for fetching operands. Thus in one clock cycle an instruction is read, is decoded, the operands are read, the operation performed and the results along with the updated PC registered at the beginning of the next clock cycle.

A pipeline design with additional stages will result in a higher clock speed and greater performance. However, there is greater complexity which is best left to when after the single pipeline stage design is up and running. A full pipelined “barrel processor” design, with one processor per pipeline stage, is a straight forward way to get maximum clock rate.

3.3. Do not use a data path

For a simple processor implementation with single pipeline stage, the instruction decode is a large case statement with each case condition expressing the instruction calculation.

Normally one would code the data path and have the instruction decode generate the control signals for the data paths. For sizeable instruction sets, this saves LUTs and results in a faster clock cycle.

3.4. Condition code register (CCR)

The conditional branch instructions specify a condition code. There are arguments for and against condition code registers. Here, it was decided to stay with a full condition code register. Thus, the CCR contains carry, signed overflow, zero and sign for each accumulator. Also included is a least significant bit (even/odd). And four bits for: all ones, all zeroes for both the floating point exponent and the floating point mantissa fields (the overall zero bit becomes redundant).

Interrupt enable and Interrupted bits also present.

3.5. Condition code register contains the ALU result

It takes additional logic to generate the some condition code bits from the ALU result (such as the zero bit). Instead save the entire ALU result and do the additional logic during the logic for conditional branches. To save the CCR, the ALU result is reduced. To restore the CCR, the all zeros and all ones bits are “unreduced”. CCR saved/restored via IO instructions.

3.6. Use block RAM for instruction memory

The smallest block RAM allows for several hundred instructions, a reasonable number. Reading instructions from a synchronous RAM is compatible with a single pipe stage implementation.

3.7. Using instruction memory for data memory

Given that instruction memory is block RAM and therefore has synchronous reads, and supports dual port operation (first port for instruction reads, second port used for data RAM read/write), the single stage pipe design does not allow a block RAM data read to occur in the same cycle as the instruction read. However a block RAM data write can occur at the end of the cycle.

Given that it takes one cycle to post the read address and the next cycle to read from the block RAM, a variety of arrangements can be made for data read: Use two instructions, one to post the read address and the next instruction to process the data. Or use a memory pointer register (in IO space) which auto increments after each data read and fetches data at the next location

3.8. Interrupts

Interrupts shall push sufficient information (program counter, condition code register, etc.) to transparently resume the program at the point of interrupt.