|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  | 🞂Table888  Develop your own processor |
|  |  |
|  | **robfinch@finitron.ca** 🞂🞂5/18/2014 |

Table of Contents

[Preface 10](#_Toc388688450)

[Qualifications: 10](#_Toc388688451)

[Choosing an Implementation Language 11](#_Toc388688452)

[Support Tools 11](#_Toc388688453)

[Documenting the Design 12](#_Toc388688454)

[Building the System 12](#_Toc388688455)

[Compilers for the Target Architecture 12](#_Toc388688456)

[Testing and Debugging 13](#_Toc388688457)

[Test Benches 13](#_Toc388688458)

[Bootstrap Code vs the “Real Code” 13](#_Toc388688459)

[Disabling Interrupts 14](#_Toc388688460)

[IRQ Live Indicator 14](#_Toc388688461)

[Disable Caching 14](#_Toc388688462)

[Stuck on a Bug ? 14](#_Toc388688463)

[The Rare Chance 14](#_Toc388688464)

[Design Choices 15](#_Toc388688465)

[Little Endian vs big Endian 15](#_Toc388688466)

[Deciding on the Degree of Pipelining 15](#_Toc388688467)

[Choosing a Bus Standard 16](#_Toc388688468)

[Choosing an ISA 16](#_Toc388688469)

[Readability 16](#_Toc388688470)

[Planning for the future 17](#_Toc388688471)

[Opcode / Instruction Size: 17](#_Toc388688472)

[Data Size 18](#_Toc388688473)

[Number of Registers: 18](#_Toc388688474)

[Register Access 19](#_Toc388688475)

[Segment Registers 19](#_Toc388688476)

[Other Registers 20](#_Toc388688477)

[Handling Immediate Values 20](#_Toc388688478)

[SETHI 22](#_Toc388688479)

[IMMxx 22](#_Toc388688480)

[LW Table 23](#_Toc388688481)

[Half-Operand Instructions 23](#_Toc388688482)

[The Branch Set 24](#_Toc388688483)

[Branch Targets 25](#_Toc388688484)

[Branch Instruction Format: 26](#_Toc388688485)

[Branch Prediction 26](#_Toc388688486)

[Looping Constructs 26](#_Toc388688487)

[Other Control Flow Instructions 26](#_Toc388688488)

[Subroutine Calls 26](#_Toc388688489)

[Returning From Subroutines 27](#_Toc388688490)

[Returning from Interrupt Routines 28](#_Toc388688491)

[Jumps 28](#_Toc388688492)

[Conditional Moves 29](#_Toc388688493)

[Predicated Instruction Execution 29](#_Toc388688494)

[Comparison Results: 29](#_Toc388688495)

[Arithmetic Operations 30](#_Toc388688496)

[Immediate Operate Functions 30](#_Toc388688497)

[Single Register Functions (The R table) 31](#_Toc388688498)

[Logical Operations 31](#_Toc388688499)

[Immediate Operate Functions 31](#_Toc388688500)

[Dual Register Functions (the RR table) 32](#_Toc388688501)

[Single register functions (The R table) 32](#_Toc388688502)

[Shift Instructions 32](#_Toc388688503)

[Other Instructions Reserved for Future Implementations 33](#_Toc388688504)

[Exception Handling 33](#_Toc388688505)

[Hardware Interrupts 34](#_Toc388688506)

[Getting and Putting Data 34](#_Toc388688507)

[Load / Store Instructions 36](#_Toc388688508)

[The Stack 36](#_Toc388688509)

[Data Caching 37](#_Toc388688510)

[Address Modes: 37](#_Toc388688511)

[Support for Semaphores 38](#_Toc388688512)

[Pipeline Design 40](#_Toc388688513)

[Processor Stages / States 40](#_Toc388688514)

[RESET 40](#_Toc388688515)

[IFETCH 40](#_Toc388688516)

[DECODE 41](#_Toc388688517)

[EXECUTE 41](#_Toc388688518)

[Memory Stage: 41](#_Toc388688519)

[LOAD1 41](#_Toc388688520)

[LOAD2 41](#_Toc388688521)

[LOAD3 41](#_Toc388688522)

[LOAD4 41](#_Toc388688523)

[STORE1 41](#_Toc388688524)

[STORE2 41](#_Toc388688525)

[STORE3 41](#_Toc388688526)

[STORE4 42](#_Toc388688527)

[Instruction Fetch: 42](#_Toc388688528)

[Instruction Cache 42](#_Toc388688529)

[Decode 43](#_Toc388688530)

[Register File Access 43](#_Toc388688531)

[Execute 43](#_Toc388688532)

[Nice-to-Have Hardware Features 44](#_Toc388688533)

[Implementing the Processor 45](#_Toc388688534)

[Convenience Tasks 45](#_Toc388688535)

[next\_state(); 45](#_Toc388688536)

[wb\_xxxx(); 45](#_Toc388688537)

[Implementing Processor Reset 45](#_Toc388688538)

[Implementing the IFETCH stage 46](#_Toc388688539)

[Implementing the Program Counter 46](#_Toc388688540)

[Implementing the Instruction Cache 48](#_Toc388688541)

[Implementing Uncached Instruction Access 51](#_Toc388688542)

[Implementing Hardware Interrupts 53](#_Toc388688543)

[Implementing the DECODE stage 54](#_Toc388688544)

[Implementing Immediates 54](#_Toc388688545)

[Implementing Target Register Selection 57](#_Toc388688546)

[Implementing the EXECUTE Stage 57](#_Toc388688547)

[Implementing Branches 57](#_Toc388688548)

[Implementing the JMP Instruction 58](#_Toc388688549)

[Implementing the JSR Instruction 58](#_Toc388688550)

[Implementing the JSR (address,Rn) and JMP (address,Rn) Instructions 60](#_Toc388688551)

[Implementing the CMP Instruction 60](#_Toc388688552)

[Implementing Arithmetic and Logical Instructions 61](#_Toc388688553)

[Implementing Multiply and Divide 62](#_Toc388688554)

[Implementing Shift Operations 65](#_Toc388688555)

[Implementing the Memory Stage 66](#_Toc388688556)

[Implementing Loads 66](#_Toc388688557)

[Implementing Stores 68](#_Toc388688558)

[Implementing the Stack Pointer 70](#_Toc388688559)

[Implementing Stack PUSH / POP operations 72](#_Toc388688560)

[Implementing the Writeback Stage 73](#_Toc388688561)

[Implementing Register Updates 73](#_Toc388688562)

[Instruction Set Description 74](#_Toc388688563)

[ADD - addition 75](#_Toc388688564)

[Instruction Formats 75](#_Toc388688565)

[Operation 75](#_Toc388688566)

[AND – bitwise logical ‘and’ 76](#_Toc388688567)

[Instruction Formats 76](#_Toc388688568)

[Operation 76](#_Toc388688569)

[ANDN – bitwise logical ‘and’ with complement 77](#_Toc388688570)

[Instruction Formats 77](#_Toc388688571)

[Operation 77](#_Toc388688572)

[ASR – Arithmetic Shift Right 78](#_Toc388688573)

[Instruction Formats 78](#_Toc388688574)

[Operation 78](#_Toc388688575)

[Bcc – Branches 79](#_Toc388688576)

[Instruction Formats 79](#_Toc388688577)

[Operation 79](#_Toc388688578)

[BRK – Breakpoint 80](#_Toc388688579)

[Instruction Formats 80](#_Toc388688580)

[Operation 80](#_Toc388688581)

[BSR – Branch to Subroutine 81](#_Toc388688582)

[Instruction Formats 81](#_Toc388688583)

[Operation 81](#_Toc388688584)

[CLI – Clear Interrupt Mask 82](#_Toc388688585)

[Instruction Formats 82](#_Toc388688586)

[Operation 82](#_Toc388688587)

[CMP - Comparison 83](#_Toc388688588)

[Instruction Formats 83](#_Toc388688589)

[Operation 83](#_Toc388688590)

[COM – bitwise ones complement 84](#_Toc388688591)

[Instruction Formats 84](#_Toc388688592)

[Operation 84](#_Toc388688593)

[DIV - Division 85](#_Toc388688594)

[Instruction Formats 85](#_Toc388688595)

[Operation 85](#_Toc388688596)

[EOR – bitwise logical exclusive ‘or’ 86](#_Toc388688597)

[Instruction Formats 86](#_Toc388688598)

[Operation 86](#_Toc388688599)

[ENOR – complement bitwise logical exclusive ‘or’ 87](#_Toc388688600)

[Instruction Formats 87](#_Toc388688601)

[Operation 87](#_Toc388688602)

[IMMx – Immediate Prefix 88](#_Toc388688603)

[Instruction Formats 88](#_Toc388688604)

[Operation 88](#_Toc388688605)

[JMP – Jump 89](#_Toc388688606)

[Instruction Formats 89](#_Toc388688607)

[Operation 89](#_Toc388688608)

[JSR – Jump to Subroutine 90](#_Toc388688609)

[Instruction Formats 90](#_Toc388688610)

[Operation 90](#_Toc388688611)

[LB – Load Byte with Sign Extend 91](#_Toc388688612)

[Instruction Formats 91](#_Toc388688613)

[Operation 91](#_Toc388688614)

[LBU – Load Byte with Zero Extend 92](#_Toc388688615)

[Instruction Formats 92](#_Toc388688616)

[Operation 92](#_Toc388688617)

[LC – Load Character with Sign Extend 93](#_Toc388688618)

[Instruction Formats 93](#_Toc388688619)

[Operation 93](#_Toc388688620)

[LCU – Load Character with Zero Extend 94](#_Toc388688621)

[Instruction Formats 94](#_Toc388688622)

[Operation 94](#_Toc388688623)

[LDI – Load Immediate 95](#_Toc388688624)

[Instruction Formats 95](#_Toc388688625)

[Operation 95](#_Toc388688626)

[LH – Load Half-Word with Sign Extend 96](#_Toc388688627)

[Instruction Formats 96](#_Toc388688628)

[Operation 96](#_Toc388688629)

[LHU – Load Half-Word with Zero Extend 97](#_Toc388688630)

[Instruction Formats 97](#_Toc388688631)

[Operation 97](#_Toc388688632)

[LW – Load Word 98](#_Toc388688633)

[Instruction Formats 98](#_Toc388688634)

[Operation 98](#_Toc388688635)

[MOD – Signed Modulus 99](#_Toc388688636)

[Instruction Formats 99](#_Toc388688637)

[Operation 99](#_Toc388688638)

[MODU – Unsigned Modulus 100](#_Toc388688639)

[Instruction Formats 100](#_Toc388688640)

[Operation 100](#_Toc388688641)

[MUL – Signed Multiply 101](#_Toc388688642)

[Instruction Formats 101](#_Toc388688643)

[Operation 101](#_Toc388688644)

[MULU – Unsigned Multiply 102](#_Toc388688645)

[Instruction Formats 102](#_Toc388688646)

[Operation 102](#_Toc388688647)

[NAND – Complement Bitwise Logical ‘And’ 103](#_Toc388688648)

[Instruction Formats 103](#_Toc388688649)

[Operation 103](#_Toc388688650)

[NEG – Negate 104](#_Toc388688651)

[Instruction Formats 104](#_Toc388688652)

[Operation 104](#_Toc388688653)

[NOP – No Operation 105](#_Toc388688654)

[Instruction Formats 105](#_Toc388688655)

[Operation 105](#_Toc388688656)

[NOR – Complement Bitwise Logical ‘Or’ 106](#_Toc388688657)

[Instruction Formats 106](#_Toc388688658)

[Operation 106](#_Toc388688659)

[NOT – Not 107](#_Toc388688660)

[Instruction Formats 107](#_Toc388688661)

[Operation 107](#_Toc388688662)

[OR – bitwise logical or 108](#_Toc388688663)

[Instruction Formats 108](#_Toc388688664)

[Operation 108](#_Toc388688665)

[ORN – Bitwise Logical Or with Complement 109](#_Toc388688666)

[Instruction Formats 109](#_Toc388688667)

[Operation 109](#_Toc388688668)

[PHP – Push Processor Status 110](#_Toc388688669)

[Instruction Formats 110](#_Toc388688670)

[Operation 110](#_Toc388688671)

[PLP – Pull Processor Status 111](#_Toc388688672)

[Instruction Formats 111](#_Toc388688673)

[Operation 111](#_Toc388688674)

[POP – Pop Register 112](#_Toc388688675)

[Instruction Formats 112](#_Toc388688676)

[Operation 112](#_Toc388688677)

[PUSH – Push Register 113](#_Toc388688678)

[Instruction Formats 113](#_Toc388688679)

[Operation 113](#_Toc388688680)

[ROL – Rotate Left 114](#_Toc388688681)

[Instruction Formats 114](#_Toc388688682)

[Operation 114](#_Toc388688683)

[ROR – Rotate Right 115](#_Toc388688684)

[Instruction Formats 115](#_Toc388688685)

[Operation 115](#_Toc388688686)

[RTI – Return From Interrupt 116](#_Toc388688687)

[Instruction Formats 116](#_Toc388688688)

[Operation 116](#_Toc388688689)

[RTS – Return From Subroutine 117](#_Toc388688690)

[Instruction Formats 117](#_Toc388688691)

[Operation 117](#_Toc388688692)

[SB – Store Byte 118](#_Toc388688693)

[Instruction Formats 118](#_Toc388688694)

[Operation 118](#_Toc388688695)

[SC – Store Character 119](#_Toc388688696)

[Instruction Formats 119](#_Toc388688697)

[Operation 119](#_Toc388688698)

[SH – Store Half-Word 120](#_Toc388688699)

[Instruction Formats 120](#_Toc388688700)

[Operation 120](#_Toc388688701)

[SHL – Shift Left 121](#_Toc388688702)

[Instruction Formats 121](#_Toc388688703)

[Operation 121](#_Toc388688704)

[SHR – Shift Right 122](#_Toc388688705)

[Instruction Formats 122](#_Toc388688706)

[Operation 122](#_Toc388688707)

[SW – Store Word 123](#_Toc388688708)

[Instruction Formats 123](#_Toc388688709)

[Operation 123](#_Toc388688710)

[SXB – Sign Extend Byte 124](#_Toc388688711)

[Instruction Formats 124](#_Toc388688712)

[Operation 124](#_Toc388688713)

[SXC – Sign Extend Character 125](#_Toc388688714)

[Instruction Formats 125](#_Toc388688715)

[Operation 125](#_Toc388688716)

[SXH – Sign Extend Half-Word 126](#_Toc388688717)

[Instruction Formats 126](#_Toc388688718)

[Operation 126](#_Toc388688719)

[Glossary 128](#_Toc388688720)

[FPGA: 128](#_Toc388688721)

[HDL 128](#_Toc388688722)

[Instruction Bundle: 128](#_Toc388688723)

[ISA: 128](#_Toc388688724)

[Program Counter: 128](#_Toc388688725)

[SIMD: 128](#_Toc388688726)

[Stack Pointer 128](#_Toc388688727)

[Major Opcode Table 130](#_Toc388688728)

[Func Table for RR instructions 131](#_Toc388688729)

[Func Table for R instructions 132](#_Toc388688730)

[01 Func Table 133](#_Toc388688731)

[02 Func Table 134](#_Toc388688732)

# Preface

One might think with a name like ‘Table888’ that this is a book about a diner or dinner date, but it’s really a book about developing a homebrew processor. As I sat down to develop yet another processor I named a table, table888. Then I thought to turn the table into a book, rather than just another ISA description.

I get to say here things that I’d never post in a hyper-technical document. Develop your own 64 bit processor ? Yeah right. One has to be somewhat nuts to consider it. But, it doesn’t take billions of dollars to develop a processor of one’s own at home; it just takes a lot of time and dedication. If you seek to be an expert on the personal computer or laptop sitting on your desk, there’s nothing like trying to develop your own processor to learn things. It’s possible these days to develop something simple and rudimentary using a small FPGA board available from several different vendors. For an outlay of a few hundred dollars one can begin to become a real expert on home-grown processors. FPGA stands for ‘Field Programmable Gate Array’, which is a chip with lots of small memories interconnected with a connection network. I’m currently using the Atlys board from Digilent. But I’ve used boards from Terasic and BurchEd in the past. Of course it’s also possible to make your own board if you have the skills. The first board I used was one I wired up myself but it didn’t work very reliably. Be sure to recycle the boards appropriately; I sell my older boards on Ebay to budding students.

The processor presented here isn’t the smallest and fastest RISC processor. That wasn’t one of my goals. Instead it offers reasonable performance with an easy to understand state machine. It’s also designed around the idea of using a simple compiler. Some operations like multiply and divide could be supported with software generated by a compiler rather than having hardware support. But I was after a simple compiler design. There’s lots of room for expansion in the future. I chose 64 bits in part anticipating more than 4GB of memory available sometime down the road. A 64-bit architecture is doable FPGA’s today, although it uses double or more the resources that a 32 bit design would.

## Qualifications:

First a warning: I’m not a professional cpu designer. I’ve simply spent a lot of time at home doing research and implementing several soft-core processors. One of the first cores I worked on was a 6502 emulation. I then went on to develop the Butterfly32 core. Later the Raptor64. I have about 20 years professional experience working on banking applications at a variety of language levels including assembler. So I have some real world experience developing complex applications. I also have a degree in electronics engineering technology. Some of the cores I work on these days are really too complex and too large to do at home on an inexpensive FPGA. I await bigger, better, faster boards yet to come.

# Choosing an Implementation Language

You will need a high-level hardware description language of some sort in order to develop a processor.

Choosing a language is somewhat of a personal choice, one should choose whatever works best for themselves. There are two popular HDL languages (Verilog, and VHDL) and number of others, I encourage you to search the web for HDL languages and find something you’re comfortable with. Not everybody speaks the same language as easily as everybody else, and it does have a little bit to do with linguistics. I know some people who will only work with schematics. My personal favorite is Verilog. VHDL is more verbose than Verilog and has tighter control of types. Table888 is implemented in the Verilog HDL language.

# Support Tools

One wouldn’t be able to achieve anything without the appropriate supporting toolsets. If you can’t get your hands on the tools (or roll your own) required to do the work maybe you shouldn’t bother. Many thanks to the vendors who supply free toolsets for use with their FPGA’s. One may have to develop one’s own tools to some extent. It’s almost like a circus performance in order to get one’s own toolsets working well. Is it the processor that’s broken ? or the toolset ? That program didn’t work because the assembler didn’t assemble it correctly, it wasn’t a bug in the processor. Keeping everything ‘in sync’ is like a dance, one goes around and around in circles. I’ve had to develop my own assembler, disassembler, compiler, glyph editing program and other things. It’s more involved than one might anticipate to begin with. For instance in order to get character display on-screen a glyph editor was needed. I looked at a couple of free ones available on the net, but they didn’t quite do what I needed. I needed something that could output FPGA vendor compatible files, and the free glyph editors were geared towards graphics files formats. After spending about a day trying to modify an existing editor I gave up, and decided to roll my own. I first developed a simple assembler about 25 years ago for use at school; I still use the same source code with many, many updates. The assembler has become quite powerful now.

## Documenting the Design

Any processor design is likely to have a number of documents associated with it. One needs to be able to refer to things like what opcode does what, outside of the implementation code itself. For general tasks I’m using MS Office. Word for word processing, and Excel for spreadsheets. Excel is handy for representing tables like opcode tables. One will likely need some sort of word processor that supports tables for documentation purposes. A simple text editor probably isn’t enough.

## Building the System

In order to actually produce an implementation some sort of FPGA developer tools will be required. I’ve used both free toolsets from Altera and Xilinx. The most recent release (14.7) of the free Webpack tools seems fairly stable under Window 8.0.

## Compilers for the Target Architecture

There are several toolsets available that can be utilized during development of soft-core processors. One of these is the LCC compiler. I used the LCC compiler for the Butterfly32 project. It’s fairly straightforward to implement the compiler for a new ISA especially if your ISA is similar to an existing one. Another toolset is the gcc compiler. I haven’t actually put this toolset to use yet, but I’ve had a look at it. It seems somewhat daunting. GCC is very general in nature and supports a lot of target architectures. People have put a lot of work into making this compiler available for any architecture. I know a number of people have been turned off by the complexity however. The compiler I use a fair bit is a modified 68000 ‘C’ compiler that I found on the net a while ago. One may have to study compilers for a while before being able to modify one or create one oneself. Compilers tend to be complex, and if you want good results for an original ISA you will have to write a good part of a compiler yourself. Not to worry, many homebrew projects get by without a compiler. There are other languages that may be useful and easy to adapt. I’ve adapted a version of Tiny Basic to several different homebrew projects now. Forth is another language popular with small systems.

# Testing and Debugging

This section seems short for the amount of testing I do. 90% of the work is in the testing. But this is a book about implementing or developing a processor, not a book about testing. Whole books could easily be written about testing. If you don’t like testing this isn’t the occupation for you. Every bug fix is a test. When one bug is fixed, the next one shows up. Good testing skills are a requirement for developing and debugging a processor. Sometimes the processor and programming cannot help you to find a bug in the processor itself. You have to be able to think in terms of ‘what test can I do ?’ to fix the bug. There are usually a least several wow-zzy bugs. For example I had a bug where a register exchange instruction only failed on a cache miss, when the instruction was at the end of a cache line. Many programs actually worked fine, and the processor seemed not to work intermittently. It took quite a while to find. I finally noticed the instruction failed when the cache was turned off. So one thing to try for testing is turning the cache on or off.

## Test Benches

If your gonna build it there must be some way to perform testing. I’d recommend writing a test-bench first and trying the code in a simulator before trying out the code in an FPGA. It is extremely unlikely that one would get the code perfect the first time. The processor is not likely to be working, so how do you fix it up ? One needs debugging dumps of course, and those are only available from a simulator. Judiciously placed debug output can be real aid to getting the cpu working. Unless a fix-up is really minor and well-known, I run simulator traces before attempting to run the code in an FPGA.

As a first test running code in the FPGA try something really simple like turning an LED on or off. One of the first lines of code Table888 executes is:

|  |
| --- |
| start  sei ; disable interrupts  ld r1,#$FF  st r1,LEDS |

which turns on all the LEDs on the board.

## Bootstrap Code vs the “Real Code”

The next thing to do after getting simpler I/O tests working is more complex I/O like a video display. Being able to display things on-screen can be invaluable (a character LCD display or LED display works good too). Also being able to get a keystroke can be valuable too. One of the first routines my processors execute is the clear-screen routine. If it can’t clear the screen I know something’s seriously wrong in the start-up. While the blue screen-of-death may be a bad sign, it’s a good sign at least the processor is working that much. When setting the processor software up (bootstrapping) don’t go for the most complex algorithms to begin with. Go with really simple things. I have two versions of keyboard routines. The one that ‘works the right way’ and the one I use for bootstrapping. The bootstrapping routine goes directly to the keyboard port to read a character. It’s really simple, and pauses the whole machine waiting for a character.

## Disabling Interrupts

Another thing nice to be able to do is disable interrupts using an external switch. There are times when one wants to know if the processor is capable of executing a linear sequence of instructions, without the interference of interrupts. Debugging the processor with interrupts enabled can be tricky. I would leave the development of an interrupt system to later stage of development. Get the processor running longer sequences of code successfully first before trying to deal with interrupts. One may want to try something like a co-operative multi-tasker that polls for external events before interrupts are working.

## IRQ Live Indicator

An indicator that IRQ’s are happening seems like a friendly image. It can be useful to see that IRQ’s are happening on a regular basis. An IRQ indicator can let one know if the machine is just busy, or really, really stuck. This can be accomplished by incrementing a character at a fixed location on-screen. IF that character stops flipping around one knows there’s real trouble.

## Disable Caching

As mentioned before, it sometimes necessary to disable the cache. Nice-to-have instructions are a cache-on and cache-off instruction. The processor should end up with the same results regardless of whether or not caching is enabled. If results seem flaky try disabling the cache.

## Stuck on a Bug ?

Try changing the code around in the area of the bug. Sometimes just by changing the code you will be able to spot a bug that wasn’t readily apparent. It’s a bit like moving your eyes around on the horizon to try and spot an enemy. The action of changing or simply moving the code causes a bug to pop out, out of the shadows.

## The Rare Chance

There is a rare chance that it’s a problem in the toolset, a problem like this can make things really difficult, especially if it’s a free toolset with no technical support. In about 10 years or so, of using toolsets I’ve found a few bugs. The toolsets generally speaking are superb, so the chance of it being a bug in a toolset is extremely remote but not impossible. The one bug I ran into was in extending a complement of a single bit value. The toolset returned “10” the value two when a single bit was being inverted. It should have returned a zero. I was able to work around this problem by zero extending the value manually. I found the bug by tracking the location of it down and dumping values using debug outputs.

# Design Choices

## Little Endian vs big Endian

One choice to make is whether the architecture is little endian or big endian. There’s a never ending argument by computer folks as to which endian is better. In reality they are both equally about the same or there wouldn’t be an argument. In a little endian architecture the least significant byte is stored at the lowest memory address. In a big-endian architecture the most significant byte is stored at the lowest memory address. I’m partial to little endian machines, it just seems more natural to me. Whichever endian is chosen, often the machine has instructions(s) for converting from one endian to the other. Some implementations even allow the endian of the machine to be set by the user. The endian of data is important because some file types depend on data being in little or big endian format.

## Deciding on the Degree of Pipelining

How much pipelining is going to be done can impact the instruction set architecture (ISA). Some things are easier or harder to do depending on the pipelining present. For instance handling large constants in an overlapped-pipelined design can be tricky, so one may want to stick with specific approaches. If one wants to support complex addressing mode such a memory indirect indexed it may be a lot easier to implement with a non-overlapped pipeline. The pipeline for Table888 is basically a non-overlapped pipeline, a couple of goals for the processor were a high clock frequency and complex instructions. I wanted to be able to implement complex instructions easily using state machines. I’ve found non-pipelined designs easier to debug as well.

## Choosing a Bus Standard

The processor interacts with the outside world using a bus. I would encourage one to use one of the commonly known bus standards to interface to the outside world. It makes it possible to use peripheral cores developed by others.

Table888 uses a WISHBONE compatible bus to communicate with the outside world. Specs for the WISHBONE bus can be found at OpenCores.org. WISHBONE bus is straightforward and easy to understand and free. It is used by a number of other projects. The bus used by Table888 is only a 32 bit bus. This is the size of the system’s data bus. All the peripherals in the test system use a 32 bit data bus. The ROM’s and RAM’s in the system are all 32 bits wide. Also the interface to the dynamic RAM memory is only 32 bits. Table888 makes use of burst memory accesses to load the instruction cache. Since instructions are only 40 bits it works okay with a 32 bit bus. Loading or storing a word to memory requires two bus accesses.

## Choosing an ISA

I would suggest as a first project to use an existing ISA and pick something simple. Designing one’s own processor tends to be project N rather than project #1. It can be quite daunting to have to develop all the tools necessary to support one’s own ISA, and an existing ISA is likely to have ready-made tools on the web. There are a large number of projects that implement existing ISA’s. MIPS must have been done about 100 times. An existing ISA is also likely to have examples of implementations in various languages. If you want to roll your own ISA it’s a lot of fun. There are many things that factor into the choice of an ISA. What is the processor geared towards ? Is it to be designed for a specific task ? What kind of resources will be available to the processor ? Is there lots of memory available, or is the amount significantly limited ? It is said that one of the pitfalls of ISA design is not allowing for growth in memory requirements.

## Readability

One of the first issues to consider is readability. This is a human factor. Believe it or not, sometimes people read machine code. Having an instruction set that contains odd sized bit fields is difficult to read (at least for me). Byte code instruction sets were partly done the way they were in order to facilitate reading the machine code, so that it would be easier for developers to write software. These days most software is written in high-level languages. As such, there is less emphasis on producing human readable machine code and more emphasis on performance. For this processor I’ve chosen to stick to a byte oriented design because I (and maybe others) will likely be reading the machine code quite a bit.

## Planning for the future

If one leaves no room for future instructions, it’ll be difficult to upgrade the processor at a later date. This instruction set has a base of 256 opcodes available; most of the opcode space is unused, and reserved for future expansion. Future expansion includes things like floating point, vector operations, and SIMD operations. While working on the instruction set for the Raptor64, which is another 64 bit processor, I found the seven bit opcode somewhat cramped. The instruction set for that processor just fit with little room left over. If possible leave several open opcodes for future expansion; that way it’ll be possible to at least use them as prefix instructions for subsequent pages of opcodes. For an example of using page prefixes see the 6809 processor. The 65C816 processor has just a single opcode left, wisely reserved for future expansions.

Part of the reason to develop a 64 bit processor isn’t that it’s really required right now, but that it has some room to grow over the next 20 years. The typical “small” FPGA board has megabytes of RAM available. To address that much memory one needs an ISA that supports the address range.

## Opcode / Instruction Size:

What works the best ? For implementing the cpu in a small FPGA device the ISA must be relatively simple. Some of the first microprocessors (6800, 6502, Z80, 8085 and others) were byte code oriented. They would fetch the first byte of an instruction and begin processing from there, fetching additional opcode bytes as needed. For simplicity the ISA I’ve chosen to implement has a fixed instruction size of 40 bits. I would not recommend using an oddball sized instruction set; it can be done, but one would need to put a lot of work into building a toolset that understood the ISA. The instruction size should at least be a multiple of eight bits. I’ve chosen 40 bits because a lot of bits are required to represent the number of registers available in the design. The instruction size is fixed to keep the instruction fetch simple, otherwise it would be necessary to implement a table containing the size for each instruction. A sample of a processor with varying sized instructions is the RTF65003 which makes use of a table to track instruction sizes. 40 bits might sound okay but a 40 bit instruction size doesn’t work well with an instruction cache, because it results in an oddball cache line length. For simplicity, typically cache lines are a power of two in length, otherwise a fast division would be required to find out which cache line to load. 128 is a power of two and it’s close to the size of three instructions (120 bits). So in addition to having an instruction size of 40 bits, the 40 bits are packed into an 128 bit instruction bundle. The bundle format:

|  |  |  |  |
| --- | --- | --- | --- |
| 127 120 | 119 80 | 79 40 | 39 0 |
| Debug | Slot2 | Slot1 | Slot0 |

### Data Size

While the size of instructions in an instruction set may vary, typically data does not. I would strongly recommend against using unusual data sizes. One would be incompatible with everything else if an unusual data size is used. It becomes a nightmare to transport and convert data files. Primitive data types should be a multiple of two of the size of a byte (eight bits). That is 8, 16, 32, or 64 bits. There are a great many well-known file formats in existence. They all rely on common data sizes. If one were to choose a nine-bit byte for instance they would have trouble packing it into the eight bits that everybody else uses.

## Number of Registers:

Some research reveals that typically somewhere around 24 registers is a sweet spot for performance. Machines with fewer registers start to suffer ill effects of moving data between registers and memory. Machines with more registers don’t actually improve very much in performance over having 20 or so registers. Having more registers impacts the task switch time because they have to swapped to memory during a task switch. Some common examples are the ARM processor which has a working set of the sixteen registers. Also the latest processors from INTEL support sixteen registers. The original INTEL 80x88 processor sported a register set of eight registers. Later more registers were added to the design. SPARC uses a register windowing scheme where there are eight global registers and twenty-four local registers which rotate around using a circular register buffer. A sixteen register machine is a good choice for performance reasons. Why aren’t there twenty-four registers if it’s a sweet-spot ? It’s a trade-off between using bits in the instruction set to represent the registers and performance impacts. The choice is really between 32 and 16 registers because either four or five bits must be used in an instruction to represent the register number. For my design I’ve actually chosen to use 256 registers, in part because the register number fits nicely into a byte. It was either going to be 16 or 256 registers, to make the register number readable. Also within the FPGA memory resources are allocated in blocks. These blocks are typically 512 or 2048 bytes in size. 256 registers fit nicely into a 256x64 block of memory (2kB). There are other reasons for choosing a large number of registers. One is the design of newer compiled languages, which can do whole program optimizations to make use of the registers. Another reason is to support AI programming; I plan on dealing with matrices and networks which might benefit from a larger register array. Lastly I’d simply like to experiment with having a large number of registers available.

### Register Access

Are registers going to be accessed in parallel or in sequence ? Some instructions require more than a single register. It may be desirable for performance reasons to be able to access more than one register at a time. To do this the register file must have multiple register read ‘ports’. On the other hand multiple read ports increase the size and cost of a register file. If one wants to keep a smaller register file, then the registers will have to be accessed in sequence. Many instructions require only a single register read access, for example the typical add immediate or compare immediate instructions. The most frequently used memory operation, load a register usually only needs to read a single register. With so many instructions requiring only a single register (or even no registers) accessing the register file sequentially across several clock cycles is a consideration for when multiple registers need to be read. Table888 uses three register read ports, mainly for simplicity, a few instructions read three registers (stores with indexed addressing for example); accessing registers sequentially can add complexity to the state machine and register read file path.

### Segment Registers

As part of the memory management portion of a cpu segment registers are often provided. There are usually multiple segment registers in order to support multiple segments which are typically part of a program. Common program segment are: the code segment, the data segment, the uninitialized data segment and the stack segment. There are often other segments as well. 80x88 is famous for its segment registers, but other processors like IBM’s PowerPC also use them as well. Segment registers are a fairly easy to understand , and a low cost memory management approach. The memory address is added to a value from a segment register in order to form a final address. The segment register is often shifted left as it is added in order to allow a greater physical memory range than the range directly supported by the architecture. Segment registers allow programs to be written as if they had specific memory addresses available to them, such as starting at location zero, while in reality the actual physical address of the program is much different. Once a design seems to be working well, I tend to add segment registers to the design as a first step at providing memory management features. Table888 does not include segment registers at this point.

### Other Registers

There are often other registers that are not general purpose in nature associated with a design. A common register is the status register, or machine control register as it is sometimes called. The status register often contains flags, and interrupt masks. It may contain other mode controlling bits like the decimal flag on the 6502 or the up/down flag on the 80x88. Many designs support additional registers such as an interrupt table base address register, a tick count register, debug registers, memory management control registers, cache control registers and others. Usually these other registers are handled with a simple move instruction between the register and a general purpose register.

## Handling Immediate Values

First some background information. A significant proportion of instructions (eg 40%) use immediate or constant values. Immediate values or constants vary widely in the number of bits required for representation, although most constants are small. Placing small constants using a field in the instruction works not too badly. The problem to solve is how to place and use large constants in the instruction stream. There are a few goals to achieve here. 1) Minimizing processor complexity. 2) Minimizing code and data size bloat. 3) Maximizing performance. There are four basic methods of handling immediate constants that I know of besides including the constant directly in the instruction stream.

1. SETHI / LUI – is an instruction to set the high order bits of a register
2. IMMxx – is an immediate prefix for the following instruction
3. LW table – placing constants in a table
4. Half-operand instructions – instructions operating on only half of a register

This architecture uses immediate prefixes for large constants. In some cases there may be two prefix instructions required in order to expand a constant out to 64 bits. The prefix instruction format follows below:

|  |  |  |
| --- | --- | --- |
| Constant32 | FDh | IMM1 |
| Constant32 | FEh | IMM2 |
| Constant32 | EAh | NOP |

The IMM1, IMM2 prefixes append onto the constant field of the following instruction. IMM1 may be used without IMM2 if the constant does not require 64 bits. If both prefixes are used they should be used in the order IMM1, IMM2. IMM1 and IMM2 prefixes lock out interrupts until the following instruction completes.

There is also a NOP instruction that looks a lot like a prefix instruction. The IMM1 instruction adds 32 bits to the inherent constant field of an instruction. The IMM2 instruction adds up to an additional 32 bit where 64 bit constants are required. If both prefixes are required, they must be used in sequence (IMM1, IMM2).

### SETHI

No, this is not the search for extra-terrestrials. I like the moniker because it reminds me of the existence of other things. SETHI is often called LUI which stands for ‘load upper immediate’.

One solution is to load an immediate value into a register using a pair of “set” instructions, then perform a register-register operation rather than a register-immediate operation. It looks like this:

|  |  |  |
| --- | --- | --- |
| ALU op used only to set the low order bits of a register -> |  | OR Rb,R0,#Low ; load low order |
| SETHI Instruction -> |  | SETHI Rb,#High ; load high order |
| Instruction Needing Large Immediate- translated into register operand -> |  | ADD Rt, Ra, Rb |

Disadvantages of this approach:

1. It often requires more memory than other solutions would. Using a large immediate requires three instructions rather than the two that a prefix would require.
2. It uses up a register(s).

Advantages of this approach:

1. It’s simple.
2. It doesn’t require processor interlocks, or re-execution of the prefix when interrupts occur. Allows instructions to execute as independent units.

### IMMxx

Second solution: use an immediate prefix instruction. The constant prefix instruction simply contains the bits of the constant that wouldn’t fit in the following instruction. It looks like the following:

|  |  |  |
| --- | --- | --- |
| Immediate prefix Instruction -> |  | IMM16 #HighBits |
| Instruction Needing Large Immediate -> |  | ADD Rt,Ra,#Lowbits |

Advantages:

It requires less memory space as the prefix needs only to contain bits to specify an immediate. Often the prefix can be arranged to contain sufficient information so that only a single instruction is needed, rather than the two that would be required for other solutions.

Disadvantages:

It can be complicated. It may require processor interlocks or re-execution of instructions when an interrupt occurs.

### LW Table

Third solution: place the large constants in a table in memory, then use regular load and store operations to load the constant into a register.

|  |  |  |
| --- | --- | --- |
| Load Instruction – retrieves value from table -> |  | LW Rb, constantAddress |
| Instruction Needing Large Immediate – translated into a register operand -> |  | ADD Rt,Ra,Rb |

Advantage:

It’s simple. It doesn’t require a special means (instructions) to handle constants. Uses a means already present in the processor. This may be useful when the size and complexity of a processor is an issue

Disadvantages:

1. It’s often slow. Load / store operations generally occur through the data port of the processor rather than the instruction port. There may be delays for memory access.

It uses a register.

### Half-Operand Instructions

Fourth solution: provide instructions that can operate on either half of a register. This looks like the following:

|  |  |  |
| --- | --- | --- |
| Instruction Needing Large Immediate (operates on lower half of register) -> |  | ADD Rt,Ra,#Low |
| Instruction operating on upper half of registers -> |  | ADDHI Rt,Ra,#High |

Advantages:

1. Minimizes code size.
2. It often doesn’t require the use of extra registers.

Disadvantages:

1) The number of instructions in the instruction set is increased. This may cause problems with the representation of instructions.

2) Increases the complexity of the processor.

## The Branch Set

One of the first things I look at when evaluating an ISA is the branch set. Is it semi-sensible or non-sense ? Branches may represent up to one quarter of instruction executed. Branches are one item that have to be well done in an architecture. What conditions will the processor branch on ? Is it a simple branch on zero / non-zero test or are there more complex conditions available ? What the branch set supports impacts what other instructions need to be available in the architecture. If branching only supports a zero / non-zero test, then other instructions must be present to setup the branch test. In the DLX architecture for instance, there are a set of ‘set’ instructions that set a register to a one or zero based on a condition. After a set instruction is done, then a conditional branch may occur. Many architectures include a compare instruction(s). For instance the MMIX architecture includes both signed (CMP) and unsigned compare (CMPU) instructions that set the value of a register to -1, 0, or 1 for less than, equal, or greater than another register. I used the same paradigm for the Raptor64 processor. For the Table888 processor there is a fairly standard set of branches that act like they are branching on a flag register value. If you’re used to the 6800 / 68x00 / 6502 series processor, these branches will look familiar.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | |  |  | |  |
| 40h | BEQ | branch if equal | 48h | BGT | branch if greater than |
| 41h | BNE | branch if not equal | 49h | BLE | branch if less or equal |
| 42h | BVS | branch if overflow set | 4Ah | BGE | branch if greater or equal |
| 43h | BVC | branch if overflow clear | 4Bh | BLT | branch if less than |
| 44h | BMI | branch if negative | 4Ch | BHI | branch if higher |
| 45h | BPL | branch if positive or zero | 4Dh | BLS | branch if lower or same |
| 46h | BRA | branch all the time | 4Eh | BHS | branch if higher or same |
| 47h | BNV | never branch | 4Fh | BLO | branch if lower |

### Branch Targets

Branches which change program flow conditionally are usually implemented as relative branches. One reason to implement using relative addresses is that it takes fewer bits to represent the target address of the branch. In many designs, typically 16 bits are allowed for, for a branch displacement even though only 12 bits are really necessary. It has to do with keeping the format of instructions simple and there is usually room in a branch instruction for sixteen bits. Even in byte-code architectures that use eight bit branch displacements by default, there is often a longer form for branches supported (for example the 6809). A second reason to use relative branching is that it allows code to be relocated in memory. Changing the location of the code in memory often does not require updating relative addresses associated with branch instructions. Note that if some form of memory management is present, it is possible to move a program in memory without having to worry about fixing up non-relative addresses, so the value of relative branches for this reason is limited.

A relative branch branches relative to the address of the branch instruction or the address of the next instruction (do not make it otherwise). I would strongly recommend using the address of the next instruction as the reference point for branches. It just makes it a bit more readable in machine code. A branch with a zero displacement arrives at the next instruction. As a ground rule, the displacement field should be at least 12 bits.

In the Table888 design 21 bits are allowed for because there are 24 bits available. This may seem like overkill, but it’s trying to look into the future of branches. When people write structured subroutines, they typically don’t create a routine more than a few pages long. This results in branching that branches within a few kilobytes of the branch location because branches are located within a subroutine. Hence the reason 12 bits is adequate. However if one is using an automated code generator, the code generator may generate larger subroutines. Note that this design differs from many in that it has page relative branching rather than full relative branching. The issue to overcome here is the 40 bit instruction size packed within a 128 bit bundle. Some portion of the target address needs to be encoded as an absolute address. At least the lower four bits would be absolute in this design. There isn’t an easy way to calculate the sum of a displacement and the current PC. Rather than jump through hoops with hardware, branches are simply made absolute within a 64kB page. This helps for readability in machine code too. Note also that there is a three bit field left over which is unassigned and should be zero. This field is reserved for future use to implement further branch instructions, perhaps for branch predictions, or perhaps for branch-to-register instructions which are not currently present in the architecture.

### Branch Instruction Format:

Shown below is the format of a conditional branch instruction.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 39 37 | 36 32 | 31 16 | 15 8 | 7 0 |  |
| ~3 | Disp5 | Addr16 | Ra8 | 4x8h | Bcc address |

### Branch Prediction

Branch prediction enhances performance by predicting which direction a conditional branch instruction will take. It is often used in overlapped or superscalar pipeline designs. Branch prediction can turn branches into a single cycle operation rather than a multi-cycle one which is what happens when a branch is taken in an overlapped pipeline design. Branch prediction has little value for the Table888 processor as it’s a non-overlapped pipeline. It takes multiple cycles to execute a branch whether or not prediction is present. Branch prediction adds additional complexity to the processor. The Raptor64 includes a (2,2) correlating branch predictor, for an example of a branch predictor.

### Looping Constructs

Sometimes processors support looping constructs directly. 680x0 has a decrement and branch instruction. 80x88 has loop instructions which decrement the CX register and branch. Decrementing a register then branching if it is non-zero is a common operation, so a number of processors implement these two operations together with a single instruction. It’s really like executing two instructions at once. Table888 supports a decrement and branch instruction for loop constructs.

## Other Control Flow Instructions

### Subroutine Calls

Subroutine calls represent about 1% of instructions executed, but it’s an important 1%. Some architectures store the return address for a subroutine call in a processor register, typically a general purpose register. These architectures may make use of a jump-and-link (JAL) instruction to both call a subroutine and return from it (for example xr16 – Grey Research).The PowerPC architecture makes use of a dedicated link register (LR). This works only for a single level of subroutine call, and the register must be saved onto the stack before calling a nested subroutine. Table888 automatically stores the return address on the stack for a subroutine call. Using a JAL instruction to return from a subroutine allows a return to a point past the original calling address. This is occasionally useful to skip over inline parameters passed to a subroutine. What’s more useful is removing parameters from the stack during a return operation. This is useful enough that a number of architectures incorporate it as part of a return instruction (680x0, 80x88). While Table888 doesn’t directly support returning past the calling point, it does support adding onto the stack pointer to remove parameters.

JSR: The jump-to-subroutine instruction first places the return address on the stack (which is the address of the next instruction) and then jumps to an absolute address. The JSR instruction loads the low order 32 bits of the program counter with the target address and leaves the upper 32 bits of the program counter unchanged. The range of this instruction may be extended to 64 bits via a constant prefix (IMM) instruction.

|  |  |  |
| --- | --- | --- |
| 39 8 | 7 0 |  |
| Address32 | 518h | JSR address |

JSR (address,Rn): This is an indexed indirect jump-to-subroutine instruction. First it saves the return address on the stack. Next it works by taking a table address and a register value as operands, calculates the index into the table, and loads the program counter with the value from the table. This is useful when one wants to setup a table of addresses of functions to call. Typically a call number is passed as a parameter to a routine, then the function address is looked up from a table using this instruction.

|  |  |  |  |
| --- | --- | --- | --- |
| 39 16 | 15 8 | 7 0 |  |
| Address24 | Ra8 | 538h | JSR (address,Rn) |

### Returning From Subroutines

Returning from a subroutine is the reverse operation to calling one. In a machine that uses registers this can be as simple as loading the PC with the register value. Some RISC architectures store the return address in a register. Table888, like many architectures loads the return address off the stack.

RTS: The RTS instruction returns from a subroutine by popping the return address from the stack. The immediate constant field is added to the stack pointer in order to remove pushed registers from the stack frame.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 39 32 | 31 16 | 15 8 | 7 0 |  |
| ~8 | SPOffset16 | ~8 | 60h | RTS |

### Returning from Interrupt Routines

Similar to a subroutine, interrupt routines also require a method of return. Typically returning from an interrupt routine requires loading some of the machine state from the stack in addition to the return address. Hardware interrupts are not normally invoked with parameters, so there are no parameters to pop off the stack at the end of an interrupt routine. Shown below is the instruction format for the RTI instruction, Table888’s way of returning from an interrupt. This instruction loads both the program counter and status register from the stack.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 40h | ~8 | ~8 | ~8 | 01h | RTI |

### Jumps

Strange as it may seem, unconditional jumps are actually very rarely used. Usually one wants the program to branch conditionally, or call a subroutine. An unconditional relative branch is usually used for jumping within a program. Jumps are sometimes used to handle exceptional conditions, where the normal subroutine return is circumvented. For instance a jump may be used to implement a program abort. Another place where jumps are used sometimes is with jump tables. Addresses of subroutines are stored in a table in memory. Functions in the table are called by loading a register with an index number, loading the address from the table using the index into the table and jumping to it. This operation can be done with registers and a jump-to-register value instruction. Table888 implements this complex operation directly as an indexed memory indirect jump.

JMP: The jump instruction takes care of jumping to an absolute address as opposed to a relative one. The jump instruction loads the low order 32 bits of the program counter with the target address and leaves the upper 32 bits of the program counter unchanged. The range of this instruction may be extended to 64 bits via a constant prefix (IMM) instruction.

|  |  |  |
| --- | --- | --- |
| 39 8 | 7 0 |  |
| Address32 | 508h | JMP address |

JMP (address,Rn): This is an indexed indirect jump instruction. It works by taking a table address and a register value as operands, calculates the index into the table, and loads the program counter with the value from the table. This is useful when one wants to setup a table of addresses of functions to call. Typically a call number is passed as a parameter to a routine, then the function address is looked up from a table using this instruction.

|  |  |  |  |
| --- | --- | --- | --- |
| 39 16 | 15 8 | 7 0 |  |
| Address24 | Ra8 | 528h | JMP (address,Rn) |

### Conditional Moves

Conditional moves are available in a number of architectures. The idea behind conditional moves is to avoid branches which are usually timely to execute. So a conditional move is a performance enhancing instruction. A conditional move ‘conditionally’ moves a value into a register based upon whether or not the condition is true. It’s like having a branch instruction combined with a load instruction. Table888 does not currently have any conditional move instructions.

## Predicated Instruction Execution

Some processors include the ability to execute virtually any instruction conditionally, for example the ARM processor or INTEL Itanium IA64. It’s a powerful means of removing branches from the instruction stream. Sequences of instructions executed with predicates rather than branching around the instructions should be kept short. The issue is the amount of time spent fetching the instructions and treating them as NOP’s versus the time it would take to branch around the instructions. A compiler can optimize this and choose the best means. One of the problems of predicates is that they use up bits in the instruction regardless of whether or not they’re actually useful. For instance the Itanium has a six bit field in virtually every instruction. The result is that a wider instruction format of 41 bits is used. A second problem with predicates is that they act like a second instruction being executed at the same time as the instruction they are associated with. The predicate operation requires a predicate register read, and a predicate evaluation operation. This adds complexity to the processor. Predicate registers are another form of register that has to be present and bypassed in an overlapped or superscalar design.

## Comparison Results:

Another issue to resolve is whether to use a flag register(s) or a result stored in a general purpose register to determine when to branch conditionally. Avoiding the use of a flags register makes it easier to implement an overlapped pipelined or superscalar design. However, most processors in large scale use use an explicit flags registers (80x88, SPARC, ARM, PowerPC uses eight flag registers). It is somewhat simpler architecturally just to use a general purpose register and branch based on the value in the register. The most common form of branching is branching on whether or not a register is zero, so a simpler architecture just uses the register directly (for example the DLX). The architecture presented here stores the flag result from a compare operation in a general purpose register. That register can then be tested using a branch instruction. Part of the benefit of having so many general purpose registers in the design is that they can act as a substitute for other forms of registers, in this case a flags register. Several of the general purpose registers in Table888 are designated as ‘flags registers’ by convention.

## Arithmetic Operations

In the simplest RISC machines one can by with just and ADD instruction. It’s possible to synthesize other operations like multiply from an ADD instruction. So instructions beyond the ADD instruction are provided for performance enhancement and programmer convenience. In some instruction sets multiply and divide operations are not supported as they consume hardware resources. Multiply and divide require multiple clock cycles to complete and have several states of their own.

Arithmetic operations include addition, subtraction, multiplication and division. These are available with the ADD, SUB, MUL, and DIV instructions. The format of the typical immediate mode instruction is shown below:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Immediate16 | Rt8 | Ra8 | 048 | ADD Rt,Ra,#imm |

There are both signed and unsigned versions of the arithmetic operations. However note there is no signed or unsigned compare operation as a single compare instruction produces results for both signed and unsigned comparisons. Signed and unsigned ADD and SUB currently work the same way. Two separate versions have been reserved in order to support the overflow exception in the future.

### Immediate Operate Functions

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| opcode / func |  |  | opcode / func |  |  |
| 04h | ADD # | addition | 14h | ADDU # | addition |
| 05h | SUB # | subtraction | 15h | SUBU # | subtraction |
| 06h | CMP # | comparison | 16h | LDI # | load immediate |
| 07h | MUL # | signed multiply | 17h | MULU # | unsigned multiply |
| 08h | DIV# | signed divide | 18h | DIVU # | unsigned divide |
| 09h | MOD # | signed modulus (remainder) | 19h | MODU # | unsigned modulus |

I’ve used the same codes for the function code in register to register instructions as the opcode code for arithmetic operations. Following is a sample register-to-register operate instruction format.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 068 | Rt8 | Rb8 | Ra8 | 028 | CMP Rt,Ra,Rb |

### Single Register Functions (The R table)

|  |  |  |
| --- | --- | --- |
| func |  |  |
| 05h | NEG |  |
| 08h | SXB | sign extend byte |
| 09h | SXC | sign extend character |
| 0Ah | SXH | sign extend half-word |

## Logical Operations

In the simplest of RISC machines one can get away with just a single inverting logical operation like NAND, or NOR. Other logical operations can be synthesized from the aforementioned ones. Once again additional instructions are supported for performance and programmer convenience.

Logic operations include logical ‘and’, logic ‘or’ and logical exclusive ‘or’ and others. The mnemonics are as follows: AND, OR, EOR, ANDN, NAND, NOR, ENOR, and ORN. Note there are no immediate forms for the following: NAND, NOR, ENOR, and ORN. The instructions formats for logical operations are the same as those for arithmetic ones.

### Immediate Operate Functions

|  |  |  |
| --- | --- | --- |
| opcode |  |  |
| 0Ch | AND # |  |
| 0Dh | OR # |  |
| 0Eh | EOR # |  |

### Dual Register Functions (the RR table)

|  |  |  |
| --- | --- | --- |
| func |  |  |
| 20h | AND |  |
| 21h | OR |  |
| 22h | EOR | exclusive or |
| 23h | ANDN | and with complement |
| 24h | NAND | complement and |
| 25h | NOR | complement or |
| 26h | ENOR | complement exclusive or |
| 27h | ORN | or with complement |

### Single register functions (The R table)

|  |  |  |
| --- | --- | --- |
| func |  |  |
| 06h | COM | one’s complement |
| 07h | NOT | logical ‘not’ |

## Shift Instructions

Shift instructions can take the place of some multiplication and division instructions. Some architectures provide shifts that shift only by a single bit. Others use counted shifts, the original 80x88 used multiple clock cycles to shift by an amount stored in the CX register. Table888 uses a barrel shifter to allow shifting by an arbitrary amount in a single clock cycle. Shifts are infrequently used and a barrel (or funnel) shifter is relatively expensive in terms of hardware resources.

The shift immediate instructions are implemented as a subset of the RR instruction group because the immediate value only needs to be six bits. This small value fits nicely into what is normally the register field for the instruction. It would be wasteful to implement these immediate mode instructions in the major opcode grouping. Shift instruction formats are shown below:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Func8 | Rt8 | Rb8 | | Ra8 | 28 | {Shifts} |
| Func8 | Rt8 | ~ | Imm6 | Ra8 | 28 | {Shifts} |

The func field is encoded as follows:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | -0 | -1 | -2 | -3 | -4 |
| 4- | SHL | ROL | SHR | ROR | ASR |
| 5- | SHL # | ROL # | SHR # | ROR # | ASR # |

## Other Instructions Reserved for Future Implementations

Branching to registers. Some higher performance designs include the capacity to conditionally branch to a location contained in a register. Supporting this functionality significantly increases the number of branch instructions. The benefit to being able to branch to a register is that the register value doesn’t have to be calculated like a branch displacement does. Therefore the target address of the branch can be known sooner.

Bit-field instructions. Bit-field instructions are nice-to-have but one can get by without them. Compilers can easily synthesize extract and insert of bit-fields using shift and ‘and’ or ‘or’ masking operations, at some performance cost.

Bitmap instructions. Bitmap instructions used to manipulate bitmaps are nice-to-have but once again they are instructions that can be synthesized by a compiler at some performance cost.

SIMD instructions. SIMD instructions are fairly straightforward to implement, however they take up a lot of room. They also may require additional registers to implement. SIMD instructions are often done with wide registers (for example 128 bits or more). SIMD instructions can considerably enhance performance for some applications because they operate on multiple data items at the same time using a single instruction.

String instructions. String type operations include block moving, block set, and block compare operations. The 80x88 has a number of string operations. Once again these operations can be performed using existing instructions at some performance cost. String operations can considerably enhance performance for some applications.

## Exception Handling

Software exceptions are just a special form of branching. When an exception occurs during an instruction, there is an automatic call to an exception handler which is located at an implied address. Almost the same thing can be done without software exceptions by using existing instructions to test for exceptional conditions, then branching if an exceptional condition is found. The reason to do things automatically is to improve performance and reduce code size. When exception handling is present, there’s no need to explicitly test for exceptional conditions in program code, the processor does it internally. There are fewer instructions fetched and executed and hence code runs faster.

|  |  |  |
| --- | --- | --- |
| Code with explicit testing |  | Code with exception handlers |
| CMP Rt,Rb,#0 |  | ; note that the opposite two lines testing |
| BEQ Rt,ExceptionHandler |  | ; for zero are unnecessary |
| DIV Rt,Ra,Rb |  | DIV Rt,Ra,Rb |
|  |  |  |

## Hardware Interrupts

Hardware interrupts are in some ways similar to software exceptions and a number of processors use the same hardware resources to implement both. The difference between a software exception and a hardware interrupt is that a software exception occurs as the result of executing an instruction and a hardware interrupt may occur at any time being triggered by an external event. Hardware interrupts are such a powerful mechanism and so useful that virtually all processors have support of some kind for them. A hardware interrupt allows the processor to respond to external events. The external event directly triggers a jump to hardware interrupt handling routine, rather than having the processor poll for the external event. The hardware interrupt ‘interrupts’ whatever the processor happens to be doing. Table888 supports hardware interrupts and uses the break (BRK) instruction in the implementation of hardware interrupts.

## Getting and Putting Data

In order to have data to work on some means must be present to transfer it to or from memory or an I/O device. Are there going to be explicit I/O instructions or is I/O memory mapped ? There is some appeal to having explicit I/O instructions. I/O typically does not require the same range of addressing that general memory does. I/O devices may be limited to a 64k page of memory as on for example the 80x88. In the test system I’ve built all the I/O is within a single megabyte address range even though there are gigabytes available. This would allow the use of shorter instructions to access the I/O. Another appealing aspect of explicit I/O instructions is that it makes it easy to indicate when data caching should not be used. One way to think of I/O instructions is as if they were uncached memory load / store instructions. Some designs have explicit uncached memory load / store operations, this is almost another way of saying I/O.

Transferring data to / from memory is what the load and store instructions are for.

Data doesn’t all come in the same size. Data size for different structures varies widely. Examples of large data structures are video frame buffers or a movie clip. A smaller structure may be a name such as a person’s name or place. About the best we can do here is load or store a portion of a data structure at a time. The processor handles the most primitive data types directly, these include bytes (8 bit), characters (16 bit), half-words (32 bit) and words (64 bit). Note that as a convention I call a 16 bit quantity a character. To me, a word is the word size of the machine, a half-word is half that size, and a byte is always eight bits. These quantities are called a byte (8 bits) a wyde (16 bits), a quadret (32 bits) and an octet (64 bits) by Knuth. The RISC paradigm is that the only instructions accessing memory are load or store instructions. This design doesn’t quite follow the paradigm. It also supports explicit stack push and stack pop operations in addition to load and store instructions. Pushing values onto the stack is a common way parameter passing is implemented in high-level languages. RISC machines synthesize this quite nicely using load and store instructions. I find push / pop instructions easier to read and understand while reading code. Is that store for a subroutine push ? or a general memory op ?

## Load / Store Instructions

The following are the load / store instructions currently supported by the architecture.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Displacement16 | | | Rt8 | Ra8 | 8x8h | Lx Rt,d16(Rn) |
| Offs6 | Sc2 | Rt8 | Rb8 | Ra8 | 8x8h | Lx Rt,o6(Ra+Rb\*sc) |

|  |  |  |  |
| --- | --- | --- | --- |
| Register Indirect with Displacement | | Indexed | |
| 80h | LB | 88h | LB |
| 81h | LBU | 89h | LBU |
| 82h | LC | 8Ah | LC |
| 83h | LCU | 8Bh | LCU |
| 84h | LH | 8Ch | LH |
| 85h | LHU | 8Dh | LHU |
| 86h | LW | 8Eh | LW |
| 87h | ~ | 8Fh | ~ |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Displacement16 | | | Rt8 | Ra8 | Ax8h | Sx Rt,d16(Rn) |
| Offs6 | Sc2 | Rt8 | Rb8 | Ra8 | Ax8h | Sx Rt,o6(Ra+Rb\*sc) |

|  |  |  |  |
| --- | --- | --- | --- |
| Register Indirect with Displacement | | Indexed | |
| A0h | SB | A8h | SB |
| A1h | SC | A9h | SC |
| A2h | SH | AAh | SH |
| A3h | SW | ABh | SW |
| A4h | ~ | ACh | ~ |
| A5h | ~ | ADh | ~ |
| A6h | PSH | AEh | ~ |
| A7h | POP | AFh | ~ |

## The Stack

This architecture has an explicitly defined stack. Oftentimes with RISC machines there is no explicit stack pointer. Instead one chooses a general register to use and uses regular load and store instructions. It’s a little bit less intuitive a way of doing things.

In this architecture register R255 is used as the stack pointer. The stack is used to store return addresses during subroutine calls. The stack may also be used to pass parameters to functions. There are instructions supporting stack operations which include JSR, RTS, PSH and POP. The format of stack push and pop instructions is shown below.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Rd8 | Rc8 | Rb8 | Ra8 | A6h | PUSH {reglist} |
| Rd8 | Rc8 | Rb8 | Ra8 | A7h | POP {reglist} |

Note that it is not possible to push R0 onto the stack. R0 is used as a placeholder for an empty slot in the push / pop instructions. Note that while some machines allow pushing or popping the entire register set with a single instruction, that is deemed to be not a good idea for a machine with 256 registers. It would create too much latency when other processing like interrupts is going on. The only other option is to be able to push or pop a subset of registers, which is allowed. The push / pop instructions push or pop any four of 256 registers. Note that the same register may be pushed or popped multiple times with the instruction.

## Data Caching

This design does not use a data cache. While a data cache can improve performance it adds complexity and can be tricky to debug. Store operations which typically write to memory are effectively un-cached anyway. Also I/O operations should not be cached. For some applications data isn’t even allowed to be cached.

## Address Modes:

A point of sale from a marketing perspective in the past has been the number and type of address modes available in the processor. “Use any address mode with any instruction.” was a statement about the simplicity of the processor when coding in assembly language. Symmetry of address modes for instructions was a selling point. These days load / store architectures are popular and in these architectures address modes really apply to only the load and store instructions. I follow this paradigm. While it is possible to have quite a general set of address modes including things like memory indirect addressing and automatic incrementing or decrementing of registers (see the 680x0 architecture for example), complex address mode can be synthesized from simpler ones and the synthesized address modes execute just as fast as built in ones. Complex addressing modes were really just an attempt a programmer convenience while programming in assembly language. Unless the language compiler is really sophisticated it’s unlikely to even be able to use some of the more complex address modes. Many RISC designs include only a single addressing mode – register indirect with displacement or sometimes only register indirect. They then rely on a compiler to synthesize other address modes are required. For this design I’ve chosen to implement two address modes for load and store instruction. The modes are register indirect with displacement, and indexed addressing with a scaled index register. I happen to like the scaled indexed address mode. It’s sometimes convenient to use the scaling.

Indexed addressing with a scaled index register works by adding two registers together with an offset in order to form the address of the data. The second index register may be optionally multiplied by 2, 4, or 8, this is called scaling. The idea behind scaling is that data may be accessed by an ordinal number, incrementing a register by one unit at a time in order to access the next data item. The scale factor accounts for the size of the data which may be one, two, four, or eight bytes in size. Without scaling it is necessary to use another register and perform a multiplication or shift operation prior to the load / store. Note that scaled indexed addressing mode uses an offset and not a displacement. The difference between an offset and a displacement is that an offset is always positive and a displacement may be either positive or negative. The offset is limited to six bits. If a larger offset or displacement is required it will have to be managed using registers. Shown below is the instruction format for scaled indexed addressing.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 | 31 24 | 23 16 | 15 8 | 7 0 |  |
| Offs6 | Sc2 | Rt8 | Rb8 | Ra8 | Ax8h | Sx Rt,o6(Ra+Rb\*sc) |

The other addressing mode that is highly useful is register indirect with displacement. In this address mode a register is added to a displacement In order to form the data address. Several other address modes may be emulated using this one. Setting the register to zero results in a displacement only mode, and setting the displacement to zero results in a register indirect mode. The displacement field is sixteen bits in size. It may be extended up to sixty-four bits using the constant extension instructions. Shown below is the instruction format for the register indirect with displacement address mode.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 39 24 | 23 16 | 15 8 | 7 0 |  |
| Displacement16 | Rt8 | Ra8 | 8x8h | Lx Rt,d16(Rn) |

## Support for Semaphores

While semaphores can be implemented using software only, it is an extremely expensive operation and slow to perform only with software. Ideally there is some support for semaphore operations supported by the processor itself. Instructions that support semaphores include instructions that atomically read-modify-write memory. A compare and swap instruction has been implemented on a number of processors to support semaphore operations. Other instructions include test-and-set bit, or increment, decrement or rotate memory.

An alternative to atomic memory instructions are instructions that perform a load and then a conditional store. These are called a locked or linked load and store. The load operation sets a flag in the processor that a semaphore access is desired. A following store operation checks this flag and aborts the store if the flag isn’t set. The flag may be reset when another processor accesses the memory region identified by the load.

This processor does not currently support a compare-and-swap or other atomic memory operations. Instead semaphore will have to be implemented with software or external hardware. The test system has a set of 1024 hardware semaphore registers available to use which can be accessed like a memory device.

# Pipeline Design

This is a non-overlapped pipelined design. A pipelined design implements the processor with a number of pipeline stages that data and instructions pass through. In an overlapped pipeline design there can be multiple instructions and multiple data items in the pipeline at the same time. Each instruction and data item can be present in each stage of the pipeline. Data and instruction dependencies between pipeline stages are resolved by hardware. An overlapped pipeline design is like a bucket-brigade where every person in the line has a bucket of water. A non-overlapped design is like a bucket brigade where there is only a single bucket of water available to be handled. This design does not use an overlapped pipeline, an overlapped pipeline is (a) more complex to implement, trickier to debug, harder to understand and (b) results in a slightly lower clock frequency for the design. However, the overall performance of an overlapped pipelined design is much greater than that of a non-overlapped design (for example by about a factor of two or more). The Raptor64 is an example of an overlapped-pipelined design. It has a CPI of around 1.5. The RTF65003 is a non-overlapped design, it has a CPI of about 3.0. The clock frequencies of the designs are comparable, although the RTF65003 has a slightly higher clock frequency achievable.

## Processor Stages / States

This section gives a general overview of what is done during each pipeline stage. The description of these stages is particular to this design. These stages are commonly found in many designs. I seem to intermix the term ‘stage’ with ‘state’. The two are similar. However a stage may contain multiple states. For instance an often identified stage is the memory stage. This stage often contains multiple states for interfacing to memory. A stage is a higher level of looking at the design.

RESET: Long running reset operations, like invalidating the cache, are done by this state. This stage transitions to the IFETCH stage.

IFETCH: Instruction fetch – This is often called a stage because sometimes multiple states are present. At this stage instructions are fetched from memory or a cache and made ready to be decoded. Register file access may also begin at this stage depending on the instruction. This stage transitions to the DECODE stage (or the ICACHE stage if there is a cache miss).

DECODE: Decode / Register access – at this stage the instruction is decoded, in parallel registers may be accessed from the register file. Constant values are also setup at this stage.

EXECUTE: - at this stage instructions are “executed”. Results are calculated based on the decoding of the previous stage.

### **Memory Stage:**

During this stage data is loaded from or stored to memory. This stage contains multiple load and store states.

LOAD1: - at this stage a memory transaction is begun. The appropriate control signals are output to the control bus and address placed on the address bus. This stage transitions to the LOAD2 stage.

LOAD2: - at this stage the memory transaction is completed (for half word or smaller data sizes). The data from memory is received into a temporary holding register. This stage transitions back to the IFETCH stage.

LOAD3:-at this state a second memory transaction is initiated in order to load the second half of a word from memory. This state is reached only for word sized operands. It transitions to the LOAD4 state.

LOAD4: - at this stage the memory transaction is completed (for word data sizes). The data from memory is received into a temporary holding register. This stage usually transitions back to the IFETCH stage.

STORE1: - at this stage memory transaction is begun. . The appropriate control signals are output to the control bus, address placed on the address bus, and data output to the data bus. This stage transitions to the STORE2 stage.

STORE2: - at this stage the memory transaction is completed. This stage may transition back to the IFETCH stage.

STORE3: - at this stage memory transaction is begun for the second half of a word operand. The appropriate control signals are output to the control bus, address placed on the address bus, and data output to the data bus. This stage transitions to the STORE4 stage.

STORE4: - at this stage the memory transaction is completed. This stage may transition back to the IFETCH stage

### Instruction Fetch:

This stage is where you find out I lied about the non-overlapped pipeline design. At this stage there is a single stage of overlap for efficiency purposes. Because it is easy and straightforward to implement, the register file update takes place at this stage in an overlapped fashion. The results from the previous instruction are written back to the register file while the next instruction is being fetched. It is safe to do for this one stage because there are no dependencies to resolve. Doing this improves the processor performance by about 20% !. Many instructions can execute in one fewer clock cycles, and the estimated CPI is only four rather than five cycles as a result of the overlap. Often there is an explicit pipeline stage called ‘WRITEBACK’ for writing results back to the register file.

An important item that is done in the IFETCH stage is checking for interrupts.

### Instruction Cache

It’s almost pointless to try to execute instructions at a high clock frequency without an instruction cache present. An instruction cache adds much to the performance of a machine. As much as 75% of memory accesses can be for instruction fetches. Loading of the instruction cache can make use of burst memory transactions, which further increases performance. Without an instruction cache, performance is limited by the speed of external memory. External memory tends to be quite slow compared to processor speeds. Without a cache there can be no overlapping of instruction fetches when another device is accessing memory, and the cpu must wait while the device does it’s memory access. If one anticipates operating without an instruction cache, and with long memory cycle times, one can develop a processor that uses lots of clock cycles to execute instructions.

If one wants instructions to fetch from an instruction cache that has to be accounted for during the instruction fetch stage. It is sometimes desirable to bypass an instruction cache during instruction fetches. That means there must be a multiplexer somewhere to switch between cached and un-cached instructions.

### Decode

Decoding instructions is done with a big case statement. All instructions are processed by the instruction decoder. Some of the simpler instructions are also executed at this stage. Instructions that don’t require register values right away may begin execution. This stage transitions into the EXECUTE stage or back to the IFETCH stage for some instructions. This stage also transitions in the memory load and store stages (LOAD1 and STORE1).

### Register File Access

During the decode stage, register file access usually takes place as well.

In the ISA the target register field “floats around” while the Ra, Rb, and Rc register read ports are always located at the same positions in the instruction set. This allows the incoming instruction to feed the register port number directly to the register to begin reading registers right away. The target register field can “float” because it isn’t needed until the register file is updated during the next IFETCH cycle. This means that the target register can be set in the decode stage. Shown in the code below, the register specs are taken directly from the IR (instruction register) while the Rt field is another register waiting to be loaded in the DECODE stage.

|  |
| --- |
| wire [7:0] Ra = ir[15:8];  wire [7:0] Rb = ir[23:16];  wire [7:0] Rc = ir[31:24];  reg [7:0] Rt; |

### Execute

This is the last stage for many instructions. Branches and other control flow instructions are executed during this stage. Memory loads and stores are also begun. It is possible to execute any instructions now because the register values from the register fetch or decode stage are stable.

By the time the EXECUTE stage is reached, all instructions will have been setup for execution, or already executed in the DECODE stage. Once again, like the DECODE stage, the EXECUTE stage uses a big case statement. At the end of the case list there is a default case. This is the place that unimplemented instructions would be handled. The normal procedure would be to invoke an unimplemented instruction exception. However for simplicity this processor just treats the unimplemented instruction like a NOP operation.

Table888 approaches ALU operations by using an inline ALU to keep things simple. The ALU is incorporated directly into the EXECUTE state. Usually the ALU is a separate distinct unit. Having a distinct ALU unit would probably allow for better optimization.

## Nice-to-Have Hardware Features

Clock stopping. Ideally the processor should be able to stop the clock under certain conditions. this is often done with a stop (STP) instruction. The Stop instruction often puts the processor in a lower power mode to conserve energy.

As it is now, the processor only implements 32 address bits for external addressing. 32 bits is enough to support 4GB of memory. The board has only a 128MB of memory, so it would be wasteful to implement a 64 bit addressing scheme.

Checking for unaligned memory access. Currently the processor does not validate that the address for data is properly aligned. It’ll go ahead and try to load data from unaligned addresses if they are specified that way; but it won’t work properly.

Check for unimplemented instructions. Unimplemented instructions should exception to a handler routine. This isn’t present in the processor, and it just treats an unimplemented instruction like a NOP.

Additional arithmetic operations such as square root, minimum and maximum functions.

Compare-and-swap and other instructions supporting semaphore operations.

Protection mechanism.

# Implementing the Processor

This section describes the details of implementing the processor.

## Convenience Tasks

A number of tasks are used for implementing parts of the processor.

### next\_state();

Throughout the code you will see the next\_state(<state name>); task called. All this task does is assign what state is next ( state <= nxt; ). It’s written as a task to allow debugging code to be placed at the time the state transitions.

### wb\_xxxx();

These tasks are for interfacing to the WISHBONE bus. It’s fairly common practice to implement the bus interfacing with tasks.

A number of Verilog tasks are used to implement the bus interfacing. A Verilog task is a bit like calling a subroutine in the high-level language; however it generates hardware every time it is called, so one has to be careful.

I set the bus controls to inactive during the wb\_nack() task, including setting the address and data lines to zero. Setting these signals to zero allows another device to take over the bus by having it wire-or’d to the same signal set. Wire’oring signals saves logic resources over having bus multiplexors.

## Implementing Processor Reset

During processor reset it’s desirable to minimize amount of logic reset. Only the registers necessary to guarantee proper operation of the processor are reset. It consumes hardware to reset registers, so only those registers that really need to be reset are reset.

At reset the instruction cache is disabled because there could be random data in it. The cache has to be invalidated before it can be used. We don’t want the processor to go off into never-never land because of bad instructions in the cache.

Interrupts can’t be allowed to occur before some software initialization has taken place. In particular the stack must be set up. There may be other devices requiring initialization before interrupts occur as well. At processor reset a global interrupt enable(gie) bit is reset to disable interrupts. This global bit is set to true on the first load of the stack pointer register.

The processor has to start executing instructions somewhere, so the PC needs to be set at reset. It is set to 32’hFFFFFFF0.

## Implementing the IFETCH stage

### Implementing the Program Counter

The program counter (also called an instruction pointer) is used to address instructions and has it’s own dedicated register. Because instructions are 40 bits wide, the program counter increments by five bytes during the decode stage, after an instruction is fetched. As mentioned earlier instructions are handled in bundles of 128 bits. So the program counter must also skip over a byte every third instruction. This is easily handled with a small table wrapped up into a function to increment the PC shown below:

|  |
| --- |
| function [31:0] pc\_inc;  input [31:0] pc;  begin  case(pc[3:0])  4'd0: pc\_inc = {pc[31:4],4'd5};  4'd5: pc\_inc = {pc[31:4],4'd10};  4'd10: pc\_inc = {pc[31:4],4'd0} + 32'd16;  default: pc\_inc = 32'hFFFFFFB0;  endcase  end  endfunction |

Note that if the program counter becomes unaligned it is automatically set to the alignment fault vector (32’hFFFFFFB0).The least significant four bits of the program counter should always be one of 0h, 5h or Ah. There isn’t much that can be done for a program where the program counter is out of alignment. There’s no telling what happened to the program. Jumping directly to the alignment fault vector doesn’t allow the processor to stack information as it would for other exceptions.

At processor reset the program counter is forced to 32’hFFFFFFF0. There should be a jump or branch instruction to boot code located there.

The default in the DECODE stage is to always increment the program counter. This default is overridden later by various instructions.

The program counter is accessible for read access as register number FEh. It is available as it is sometimes convenient for program counter relative addressing. With 256 registers available is doesn’t make sense not to include this one. Note that data must be appropriately aligned in memory and the program counter counts mod 5, so using it for program counter relative addressing could be a challenge, but it’s available.

There are a number of instructions dedicated to modifying the program counter. These include jumps which set the program counter directly, jump-to-subroutine which also sets the program counter directly, branches which modify the program counter by adding or subtracting from it, the RTS instruction which loads the program counter from the stack and optionally adds to it, and the BRK instruction.

### Implementing the Instruction Cache

The instruction cache is an 8KB direct-mapped cache. Direct mapped caches are about the simplest to implement. There are other cache types in existence which can offer better performance.

The instruction cache would be complicated by the fact that the processor fetches instructions in groups of five bytes except that the processor was designed with 128 bit instruction bundles. As far as the cache is concerned it only has to fetch cache lines that are aligned on sixteen byte addresses. It doesn’t need to worry about the processor quirks.

|  |  |
| --- | --- |
| 0x00…0000 | Line #0 |
| 0x00…0010 | Line #1 |
| 0x00…0020 | Line #2 |
| 0x00...0030 | Line #3 |
| 0x00...0040 | Line #4 |
| … |  |

There are two pieces to an instruction cache, the cache memory and cache tag ram.

#### The Cache Ram

The instruction cache memory is implemented using a small synchronous ram memory which is embedded within a larger cache module:

|  |
| --- |
| module syncram\_512x32\_1rw1r(wclk, wr, wa, i, rclk, ra, o);  input wclk;  input wr;  input [8:0] wa;  input [31:0] i;  input rclk;  input [8:0] ra;  output [31:0] o;  reg [31:0] mem [511:0];  reg [8:0] rra;  always @(posedge wclk)  if (wr)  mem[wa] <= i;  always @(posedge rclk)  rra <= ra;  assign o = mem[rra];  endmodule |

The larger cache ram module uses four of the syncram modules to create a 128 bit wide memory. The 128 bit width is required in order to obtain 40 bit slices at one time. The total cache ram is 8kB in size (or 1536 instruction words). 128 bit data from this memory is multiplexed onto an instruction bus according to the low order four bits of the program counter. If the program counter is unaligned, an alignment fault jump is placed into the instruction stream. Also pulled out of the bundle are debug bits for the instruction.

|  |
| --- |
| module icache\_ram(wclk, wr, wa, i, rclk, pc, insn, debug\_bits);  input wclk;  input wr;  input [12:0] wa;  input [31:0] i;  input rclk;  input [12:0] pc;  output reg [39:0] insn;  output reg [1:0] debug\_bits;  wire [31:0] o1,o2,o3,o4;  syncram\_512x32\_1rw1r u1 (wclk, wr && wa[3:2]==2'b00, wa[12:4], i, rclk, pc[12:4], o1);  syncram\_512x32\_1rw1r u2 (wclk, wr && wa[3:2]==2'b01, wa[12:4], i, rclk, pc[12:4], o2);  syncram\_512x32\_1rw1r u3 (wclk, wr && wa[3:2]==2'b10, wa[12:4], i, rclk, pc[12:4], o3);  syncram\_512x32\_1rw1r u4 (wclk, wr && wa[3:2]==2'b11, wa[12:4], i, rclk, pc[12:4], o4);  wire [127:0] bundle = {o4,o3,o2,o1};  always @(bundle or pc)  case(pc[3:0])  4'h0: insn <= bundle[ 39: 0];  4'h5: insn <= bundle[ 79:40];  4'hA: insn <= bundle[119:80];  default: insn <= 40'hFFFFFFEC\_50; // JMP Alignment fault  endcase  always @(bundle or pc)  case(pc[3:0])  4'h0: debug\_bits <= bundle[121:120];  4'h5: debug\_bits <= bundle[123:122];  4'hA: debug\_bits <= bundle[125:124];  default: debug\_bits <= 2'b00;  endcase  endmodule |

The tag ram and cache ram modules are instanced in Table888 as follows:

|  |
| --- |
| wire ihit;  icache\_tagram u1 (  .wclk(clk\_i),  .wr((ack\_i & isInsnCacheLoad)|isCacheReset),  .wa(adr\_o),  .v(!isCacheReset),  .rclk(~clk\_i),  .pc(pc),  .hit(ihit)  );  icache\_ram u2 (  .wclk(clk\_i),  .wr(ack\_i & isInsnCacheLoad),  .wa(adr\_o),  .i(dat\_i),  .rclk(~clk\_i),  .pc(pc),  .insn(insn),  .debug\_bits()  ); |

State machine states required to load the cache are shown below:

|  |
| --- |
| // ----------------------------------------------------------------------------  // Instruction cache load machine states.  // ----------------------------------------------------------------------------  ICACHE1:  begin  isInsnCacheLoad <= `TRUE;  wb\_burst(6'd3,{pc[31:4],4'h0});  next\_state(ICACHE2);  end  ICACHE2:  if (ack\_i) begin  if (adr\_o[3:2]==2'b10)  cti\_o <= 3'b111;  if (adr\_o[3:2]==2'b11) begin  isInsnCacheLoad <= `FALSE;  wb\_nack();  next\_state(IFETCH); // return to where we came from  end  adr\_o[3:2] <= adr\_o[3:2] + 2'd1;  end |

The ICACHE1 state starts a WISHBONE burst transfer then transitions to the ICACHE2 state. A burst transfer is four 32 bit words in length, 128 bits, which is the cache line length. The ICACHE2 state waits for ack’s back from memory and takes care of incrementing the address.The Tag Ram

The tag ram makes use of the syncram module to store the tags for the instruction cache. Each tag is 32 bits in size of which only 20 bits are used. A cache hit test is performed by comparing the address stored in the tag ram to the program counter. If they match, and the tag ram valid bit is set, then hit will be true.

|  |
| --- |
| module icache\_tagram(wclk, wr, wa, v, rclk, pc, hit);  input wclk;  input wr;  input [31:0] wa;  input v;  input rclk;  input [31:0] pc;  output hit;  wire [31:0] tag;  syncram\_512x32\_1rw1r u1 (wclk, wr && wa[3:2]==2'b11, wa[12:4], {wa[31:1],v}, rclk, pc[12:4], tag);  assign hit = tag[31:13]==pc[31:13] && tag[0];  endmodule |

If the hit test fails, it is detected in the IFETCH state of the processor, and a memory access to load the missing i-cache line is initiated by transitioning to the ICACHE1 state.

|  |
| --- |
| else if (!ihit & !uncachedArea & icacheOn)  next\_state(ICACHE1); |

#### Implementing Cache Invalidates

There are times when the cache needs to be invalidated. The cache needs to be invalidated when new code is loaded into a block of memory previously occupied by a different code. To simplify the hardware this design uses software to invalidate the cache. Cache lines can be invalidated by calling a short subroutine consisting only of an RTS located a fixed location in memory. The entire cache can be invalidated by calling a routine consisting of only NOP operations, where the routine is as large as the cache to invalidate. Performing software invalidation of the cache is a lot slower for performance than having hardware invalidation present.

### Implementing Uncached Instruction Access

There are times when uncached access to instructions is desirable. Code that is only executed one-time may be better to run with the cache off, so that it doesn’t boot code out of the cache that runs many times. Uncached instruction access is required at least to boot the processor. It’s also valuable for debugging purposes. It can be confusing during a debug simulation run not to be able to see the instructions (the address bus) executing in sequence. With a cache running there are many cases where the address bus appears idle because instruction fetches are coming from the cache.

Uncached access acts a little bit like a continuous cache miss. For every instruction to execute a transition is made to the IBUF1 state in order to fetch the instruction.

|  |
| --- |
| else if (ibufmiss & (uncachedArea | !icacheOn))  next\_state(IBUF1); |

ibufmiss is a signal that tests the address for the current instruction against the program counter. This signal will always mismatch except for instructions that branch back to themselves. The address associated with the current instruction is stored in the ibufadr signal.

|  |
| --- |
| reg [31:0] ibufadr;  wire ibufmiss = ibufadr != pc; |

The IBUF1 state initiates a two word burst access in order to load a buffer. The program counter is rounded down to align with a 32 bit word address.

|  |
| --- |
| IBUF1:  begin  wb\_burst(6'd1,{pc[31:2],2'h0});  next\_state(IBUF2);  end |

It takes two 32-bit word accesses to load the instruction buffer with a 40 bit instruction. The first access loads the bottom portion of the instruction buffer with data coming from memory. The data is appropriately aligned with a multiplexor. It may be necessary to read beginning with any byte of the first word fetched as the PC addressing works mod five.

|  |
| --- |
| IBUF2:  begin  if (ack\_i) begin  cti\_o <= 3'b111;  adr\_o <= adr\_o + 32'd4;  case(pc[1:0])  2'b00: ibuf[31:0] <= dat\_i;  2'b01: ibuf[23:0] <= dat\_i[31:8];  2'b10: ibuf[15:0] <= dat\_i[31:16];  2'b11: ibuf[7:0] <= dat\_i[31:24];  endcase  next\_state(IBUF3);  end  end |

The second word access fills the top portion of the instruction buffer with the remaining required bits. The ibufadr is set the address of instructions just fetched (the program counter address) so that a match will occur in the IFETCH state, and a transition back to the IFETCH state is made.

|  |
| --- |
| IBUF3:  begin  if (ack\_i) begin  wb\_nack();  ibufadr <= pc;  case(pc[1:0])  2'b00: ibuf[39:32] <= dat\_i[7:0];  2'b01: ibuf[39:24] <= dat\_i[15:0];  2'b10: ibuf[39:16] <= dat\_i[23:0];  2'b11: ibuf[39:8] <= dat\_i;  endcase  next\_state(IFETCH);  end  end |

Note that access is somewhat inefficient as typically the same word of memory is fetched for a subsequent instruction. The state machine has been designed to be simple, but could be optimized further.

### Implementing Hardware Interrupts

There are typically two places that hardware interrupts can be checked for, 1) at the start of an instruction or 2) at the end of the execution of an instruction. Hardware interrupts in Table888 are checked for at the start of the IFETCH state. The first thing that happens in the IFETCH stage is a check for a non-maskable interrupt. Placing this check first gives a non-maskable interrupt the highest priority of operation. Next maskable interrupts are checked for.

|  |
| --- |
| IFETCH:  begin  hwi <= `FALSE;  next\_state(DECODE);  if (nmi\_edge & gie & ~hasIMM) begin  ir[7:0] <= `BRK;  ir[39:8] <= `NMI\_VECT;  nmi\_edge <= 1'b0;  hwi <= `TRUE;  end  else if (irq\_i & gie & !im & ~hasIMM) begin  ir[7:0] <= `BRK;  ir[39:8] <= {vbr[31:13],vect\_i,4'd0};  hwi <= `TRUE;  end |

Note that interrupts are ignored before the stack pointer is loaded, by checking the global interrupt enable bit. Interrupts are also ignored if a constant prefix instruction was previously executed.

Non-maskable interrupts are edge triggered. If an interrupt edge occurs it is detected and recorded; even if the interrupt ‘goes away’ at a later point, the non-maskable interrupt routine is still called. Contrasted with maskable interrupts which are level sensitive, meaning the interrupt is taken only if it is still present on the interrupt line when the processor checks for it.

|  |
| --- |
| if (nmi\_i & !nmi1)  nmi\_edge <= `TRUE; |

Maskable interrupts use a vector table to determine the address of the interrupt subroutine. The vector number, used in the calculation of the address, is loaded by the processor from the vect\_i signal. The vect\_i signal is supplied externally by the interrupt controller when an interrupt occurs.

## Implementing the DECODE stage

### Implementing Immediates

Many instructions have constant or immediate operands associated with them. These values need to be appropriately sign or zero extended before use. Another thing that has to be included is support for large constants. Support for large constants is accomplished using immediate prefix instructions which load a constant buffer. The constant buffer is called ‘immbuf’ and declared as a register:

|  |
| --- |
| reg [63:0] immbuf; |

The IMM1, IMM2 prefixes append onto the constant field of the following instruction. IMM1 may be used without IMM2 if the constant does not require 64 bits. If both prefixes are used they should be used in the order IMM1, IMM2. IMM1 and IMM2 prefixes lock out interrupts until the following instruction completes. Lockout code shown below:

|  |
| --- |
| wire hasIMM = isIMM1|isIMM2; |
| if (nmi\_edge & gie & ~hasIMM) begin  ir[7:0] <= `BRK;  ir[39:8] <= `NMI\_VECT;  nmi\_edge <= 1'b0;  hwi <= `TRUE;  end  else if (irq\_i & gie & !im & ~hasIMM) begin  ir[7:0] <= `BRK;  ir[39:8] <= `IRQ\_VECT;  hwi <= `TRUE;  end |
|  |

The IMM1 prefix sign extends an immediate constant found in a 32 bit immediate constant field in the instruction, to 64 bits and places the result into an internal constant buffer. The constant buffer is a non-visible internal buffer used by the processor to build large immediate constants. Typically a sixteen bit constant can be extended to forty-eight bits using just the IMM1 prefix.

The IMM2 prefix loads a 32 bit immediate constant into the upper half of the constant buffer leaving the lower half unchanged, overriding the previous sign extension of an IMM1 instruction. Combining an IMM2 instruction with an IMM1 instruction allows a 64 bit constant to be built in the buffer. Both prefixes set a flag in the decoder stage so that interrupt lock-outs can occur.

|  |
| --- |
| // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -  // Prefixes follow  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -  `IMM1:  begin  isIMM1 <= `TRUE;  immbuf <= {{32{ir[39]}},ir[39:8]};  end  `IMM2:  begin  isIMM2 <= `TRUE;  immbuf[63:32] <= ir[39:8];  end |

Constants for a variety of instructions are multiplexed into a single immediate register during the decode stage. The default is to sign extend a sixteen bit constant field, which is the most common. The constant field is combined with the immediate prefix buffer if an immediate prefix was present.

|  |
| --- |
| // Immediate value multiplexer  case(opcode)  `LDI: imm <= hasIMM ? {immbuf[39:0],ir[39:16]} : {{40{ir[39]}},ir[39:16]};  `JSR: imm <= ir[39:8]; // PC has only 32 bits implemented  `JMP: imm <= ir[39:8];  `JSR\_IX: imm <= hasIMM ? {immbuf[39:0],ir[39:16]} : ir[39:16];  `JMP\_IX: imm <= hasIMM ? {immbuf[39:0],ir[39:16]} : ir[39:16];  `JMP\_DRN: imm <= hasIMM ? {immbuf[39:0],ir[39:16]} : ir[39:16];  `JSR\_DRN: imm <= hasIMM ? {immbuf[39:0],ir[39:16]} : ir[39:16];  `LBX,`LBUX,`LCX,`LCUX,`LHX,`LHUX,`LWX,`SBX,`SCX,`SHX,`SWX:  imm <= hasIMM ? {immbuf[57:0],ir[39:34]} : ir[39:34];  default:  if (hasIMM)  imm <= {immbuf[47:0],ir[39:24]};  else  imm <= {{48{ir[39]}},ir[39:24]};  endcase |

### Implementing Target Register Selection

As mentioned earlier, the target register field of the instruction “floats around” as it is not needed until after the DECODE stage. The target register number Rt is set from this field which is pulled out of the instruction register from the appropriate position according to the instruction.

|  |
| --- |
| // Set the target register  case(opcode)  `R: Rt <= ir[23:16];  `RR: Rt <= ir[31:24];  `LDI: Rt <= ir[15:8];  `ADDI,`ADDUI,`SUBI,`SUBUI,`CMPI,`MULI,`MULUI,`DIVI,`DIVUI,`MODI,`MODUI,  `ANDI,`ORI,`EORI:  Rt <= ir[23:16];  `LB,`LBU,`LC,`LCU,`LH,`LHU,`LW:  Rt <= ir[23:16];  `LBX,`LBUX,`LCX,`LCUX,`LHX,`LHUX,`LWX:  Rt <= ir[31:24];  default:  Rt <= 8'h00;  endcase |

The default is to set the target register number to zero, which means that no register update will take place (or rather register zero is updated, but it bypassed to zero anyway). Note that a number of instructions like (SEI, CLI, PLP, etc.) require that the target field of the instruction is set to zero, otherwise a register would be updated. The assembler takes care of this. Currently, there is no hardware protection against setting the field to a non-zero value.

## Implementing the EXECUTE Stage

### Implementing Branches

Branches evaluate the result of a comparison operation and branch accordingly. The comparison result is stored in a register, this register value is available from operand register ‘a’. The opcode is decoded and a flag signal (take\_branch) is set depending on the branch condition.

|  |
| --- |
| // Evaluate branches  //  reg take\_branch;  always @(a or opcode)  case(opcode)  `BEQ: take\_branch <= a[1];  `BNE: take\_branch <= !a[1];  `BVS: take\_branch <= a[62];  `BVC: take\_branch <= !a[62];  `BMI: take\_branch <= a[63];  `BPL: take\_branch <= !a[63];  `BRA: take\_branch <= `TRUE;  `BRN: take\_branch <= `FALSE;  `BHI: take\_branch <= a[0] & !a[1];  `BHS: take\_branch <= a[0];  `BLO: take\_branch <= !a[0];  `BLS: take\_branch <= !a[0] | a[1];  `BGT: take\_branch <= (a[63] & a[62] & !a[1]) | (!a[63] & !a[62] & !a[1]);  `BGE: take\_branch <= (a[63] & a[62])|(!a[63] & !a[62]);  `BLT: take\_branch <= (a[63] & !a[62])|(!a[63] & a[62]);  `BLE: take\_branch <= a[1] | (a[63] & !a[62])|(!a[63] & a[62]);  default: take\_branch <= `FALSE;  endcase |

In the EXECUTE stage the take\_branch signal is tested and the PC updated if the signal is true.

|  |
| --- |
| `BEQ,`BNE,`BVS,`BVC,`BMI,`BPL,`BRA,`BRN,`BGT,`BGE,`BLT,`BLE,`BHI,`BHS,`BLO,`BLS:  begin  next\_state(IFETCH);  if (take\_branch) begin  pc[15: 0] <= ir[31:16];  pc[31:16] <= pc[31:16] + {{11{ir[36]}},ir[36:32]};  end  end |

### Implementing the JMP Instruction

The JMP instruction is one of the easiest to implement; the PC is simply loaded with bits from the IR. Note that only the lowest 32 bits of the PC are loaded. In this implementation only 32 bits are supported.

|  |
| --- |
| `JMP: pc <= ir[39:8]; |

### Implementing the JSR Instruction

The JSR instruction must first save the current program counter on the stack before loading the program counter with the subroutine address. It uses the general purpose STORE1 state to store the program counter. The JSR decode also sets a flag indicating the JSR instruction is in progress. This flag is used in the STORE4 state.

|  |
| --- |
| `JSR:  begin  isJSR <= `TRUE;  wadr <= sp\_dec[31:0];  sp <= sp\_dec;  store\_what <= `STW\_PC;  next\_state(STORE1);  end |

The STORE4 state detects the JSR instruction and loads the program counter. Program counter loading has to be done after the program counter is saved on the stack. This state also detects a PUSH or POP operation and branches back to the DECODE state if a PUSH or POP is in progress.

|  |
| --- |
| STORE4:  begin  wb\_nack();  case(1'b1)  isBRK:  begin  if (store\_what==`STW\_PC) begin  pc <= ir[39:8];  next\_state(IFETCH);  end  else begin  store\_what <= `STW\_PC;  next\_state(STORE1);  end  end  isJSR:  begin  pc <= ir[39:8];  next\_state(IFETCH);  end  isJSRix:  next\_state(LOAD1);  isPOP,isPUSH:  next\_state(DECODE);  default:  next\_state(IFETCH);  endcase  end |

### Implementing the JSR (address,Rn) and JMP (address,Rn) Instructions

The JMP instruction is similar, but simpler than the JSR instruction so I’m only describing the JSR instruction which is more complex. The JMP instruction only needs to perform a load; the store of the program counter isn’t required.

The indexed indirect addressing mode for the JSR instruction requires both a store and a load operation. The store portion is similar as to the JSR instruction, except at the end of the store a transition is made to the LOAD1 state in order to load the subroutine address from a table. Both the load and store addresses are setup in the DECODE stage. The stack pointer is also decremented and a flag is set indicating that a JSR (addr,Rn) instruction is taking place.

|  |
| --- |
| `JSR\_IX:  begin  radr <= rfoa + imm;  wadr <= sp\_dec[31:0];  sp <= sp\_dec;  isJSRix <= `TRUE;  store\_what <= `STW\_PC;  next\_state(STORE1);  end |

In the STORE4 state shown previously, a transition is made to the LOAD1 state if a JSR (addr,Rn) instruction is detected.

### Implementing the CMP Instruction

The compare instruction needs to generate result flags as the result of a comparison. The result flags are actually generated from the result of a subtract operation. Hence there are two layers to a compare operation, unlike other arithmetic operations. One of the layers has to be moved out to combinational logic, which is shown below. The compare operation makes use of a function to calculate overflow.

|  |
| --- |
| // Overflow:  // Add: the signs of the inputs are the same, and the sign of the  // sum is different  // Sub: the signs of the inputs are different, and the sign of  // the sum is the same as B  function overflow;  input op;  input a;  input b;  input s;  begin  overflow = (op ^ s ^ b) & (~op ^ a ^ b);  end  endfunction  // Generate result flags for compare instructions  wire [64:0] cmp\_res = a - (isCMPI ? imm : b);  reg nf,vf,cf,zf;  always @(cmp\_res or a or b or imm or isCMPI)  begin  cf <= cmp\_res[64];  nf <= cmp\_res[63];  vf <= overflow(1,a[63],isCMPI ? imm[63] : b[63], cmp\_res[63]);  zf <= cmp\_res[63:0]==64'd0;  end |

After the flags are calculated they are loaded onto the result bus during the DECODE stage:

|  |
| --- |
| // This case statement decodes all instructions.  case(opcode)  `RR:  case(func)  …  `CMP: res <= {nf,vf,60'd0,zf,cf};  …  default: res <= 65'd0;  endcase  …  `CMPI: res <= {nf,vf,60'd0,zf,cf}; |

Note that the negative flag is loaded into the most significant bit (MSB) of the result. This allows branches to branch directly based on whether or not a register is minus or non-minus. It is also possible to branch directly based on whether a register is odd or even.

### Implementing Arithmetic and Logical Instructions

Most arithmetic and logical instructions work the same way. They can be quickly calculated on a single line of code. The exception is multiply and divide operations which require multiple cycles and have their own states.

|  |
| --- |
| // This case statement decodes all instructions.  case(opcode)  `RR:  case(func)  `ADD,`ADDU: res <= rfoa + rfob;  `SUB,`SUBU: res <= rfoa - rfob;  `CMP: res <= {nf,vf,60'd0,zf,cf};  `MUL,`MULU,`DIV,`DIVU,`MOD,`MODU: next\_state(MULDIV);  `AND: res <= rfoa & rfob;  `OR: res <= rfoa | rfob;  `EOR: res <= rfoa ^ rfob;  `ANDN: res <= rfoa & ~rfob;  `NAND: res <= ~(rfoa & rfob);  `NOR: res <= ~(rfoa | rfob);  `ENOR: res <= ~(rfoa ^ rfob);  `ORN: res <= rfoa | ~rfob;  …  default: res <= 65'd0;  endcase  `ADDI,`ADDUI: res <= rfoa + imm;  `SUBI,`SUBUI: res <= rfoa - imm;  `CMPI: res <= {nf,vf,60'd0,zf,cf};  `MULI,`MULUI,`DIVI,`DIVUI,`MOD,`MODU: next\_state(MULDIV);  `ANDI: res <= rfoa & imm;  `ORI: res <= rfoa | imm;  `EORI: res <= rfoa ^ imm; |

### Implementing Multiply and Divide

First, a note. Modulus and divide are the same operation. They are performed by the same hardware, the divide hardware produces both a quotient and a remainder. The quotient is the divide result and the remainder is the modulus result. So when I talk about divide I’m also referring to the modulus operation unless otherwise noted.

The first thing the multiply and divide instructions do is setup operands. For signed multiplies the result sign is calculated and the operands are made positive if they are negative. The result will be corrected to the right sign later.

|  |
| --- |
| MULDIV:  begin  cnt <= 7'd64;  case(opcode)  `MULUI:  begin  aa <= a;  bb <= b;  res\_sgn <= 1'b0;  next\_state(MULT1);  end  `MULI:  begin  aa <= a[63] ? -a : a;  bb <= b[63] ? -b : b;  res\_sgn <= a[63] ^ b[63];  next\_state(MULT1);  end  `DIVUI,`MODUI:  begin  aa <= a;  bb <= b;  q <= a[62:0];  r <= a[63];  res\_sgn <= 1'b0;  next\_state(DIV);  end  `DIVI,`MODI:  begin  aa <= a[63] ? -a : a;  bb <= b[63] ? -b : b;  q <= pa[62:0];  r <= pa[63];  res\_sgn <= a[63] ^ b[63];  next\_state(DIV);  end  `RR:  case(func)  `MULU:  begin  aa <= a;  bb <= b;  res\_sgn <= 1'b0;  next\_state(MULT1);  end  `MUL:  begin  aa <= a[31] ? -a : a;  bb <= b[31] ? -b : b;  res\_sgn <= a[63] ^ b[63];  next\_state(MULT1);  end  `DIVU,`MODU:  begin  aa <= a;  bb <= b;  q <= a[62:0];  r <= a[63];  res\_sgn <= 1'b0;  next\_state(DIV);  end  `DIV,`MOD:  begin  aa <= a[63] ? -a : a;  bb <= b[63] ? -b : b;  q <= pa[62:0];  r <= pa[63];  res\_sgn <= a[63] ^ b[63];  next\_state(DIV);  end  default:  state <= IDLE;  endcase  endcase  end |

The multiply instruction makes use of the built in Verilog multiply operator ‘\*’.

|  |
| --- |
| wire [127:0] p1 = aa \* bb; |

Since it is a 64 bit multiply operation the delay is greater than the clock cycle period for the remainder of instructions. So that this delay does not impact the processor’s clock cycle time, it is implemented over several clock cycles. It looks like the state machine isn’t doing anything during those cycles (MULT1, MULT2), but it is actually waiting for the multiply to complete.

|  |
| --- |
| // Three wait states for the multiply to take effect. These are needed at  // higher clock frequencies. The multiplier is a multi-cycle path that  // requires a timing constraint.  MULT1: state <= MULT2;  MULT2: state <= MULT3;  MULT3: begin  p <= p1;  next\_state(res\_sgn ? FIX\_SIGN : MD\_RES);  end |

Multiply and Divide both use a common state to fix up the result for signed multiplies and divides. If the results should be negative, they are made so during this state.

|  |
| --- |
| FIX\_SIGN:  begin  next\_state(MD\_RES);  if (res\_sgn) begin  p <= -p;  q <= -q;  r <= -r;  end  end |

A final state places the appropriate result on the result bus.

|  |
| --- |
| MD\_RES:  begin  if (opcode==MULI || opcode==MULUI || (opcode==`RR && (func==MUL || func==MULU))  res <= p[63:0];  else if (opcode==DIVI || opcode==DIVUI || (opcode==`RR && (func==DIV || func==DIVU))  res <= q[63:0];  else  res <= r[63:0];  next\_state(IFETCH);  end |

There are several different method of performing a divide operation (Booth, Newton, ). To improve divider performance, results may also be cached. The method used here is a basic non-restoring algorithm. The algorithm doesn’t use any shortcuts, and operates a single bit at a time (radix 2). It takes over 64 clock cycles to perform a divide. Multiply is much faster, typically done in under eight clock cycles.

|  |
| --- |
| wire [63:0] diff = r - bb; |
| DIV:  begin  q <= {q[62:0],~diff[63]};  if (cnt==7'd0) begin  next\_state(res\_sgn ? FIX\_SIGN : MD\_RES);  if (diff[63])  r <= r[62:0];  else  r <= diff[62:0];  end  else begin  if (diff[63])  r <= {r[62:0],q[63]};  else  r <= {diff[62:0],q[63]};  end  cnt <= cnt - 7'd1;  end |

### Implementing Shift Operations

Shift operations are implemented using an extra wide (128 bit) result bus so that all the bits shifted can be captured within the result register. Shifts use the built in shift operators (<<. >>) to generate a barrel shifter.

|  |
| --- |
| // Shifts  wire [5:0] shamt = isShifti ? Rb[5:0] : b[5:0];  wire [127:0] shlo = {64'd0,a} << shamt;  wire [127:0] shro = {a,64'd0} >> shamt;  wire signed [63:0] as = a;  wire signed [63:0] asro = as >> shamt; |

Rotate operations are implemented by merging together two halves of the 128 bit shift result register.

## Implementing the Memory Stage

### Implementing Loads

Loads make use of a read address signal called ‘radr’ which holds onto the read address.

The load address is calculated in the DECODE stage, the operation size is set according to the instruction in the ld\_size signal. Next a general purpose load state (LOAD1) is called. One element from the DECODE case statement is shown below. The other cases are similar.

|  |
| --- |
| `LB:  begin  radr <= a + imm;  ld\_size = byt;  next\_state(LOAD1);  end |

The general purpose load machine follows below. It takes care of regular load instructions, the POP instruction, RTS instruction and indexed indirect JMP’s and JSR’s. Loads always load the result bus and hence don’t need the qualifying signal analogous to ‘store\_what’. The first thing it does is initiate a read cycle at the ‘radr’ address. There are four load states, the last two states are reached only for word sized operations. It takes two bus cycles to perform a word load because the bus interface is only 32 bits wide. The second state takes care of sign and zero extending data that is less than a word in size.

|  |
| --- |
| // ----------------------------------------------------------------------------  // LOAD machine states.  // ----------------------------------------------------------------------------  LOAD1:  begin  wb\_read(radr);  next\_state(LOAD2);  end  LOAD2:  begin  if (ack\_i) begin  radr <= radr + 32'd4;  wb\_nack();  if (ld\_size==word) begin  res[31:0] <= dat32;  next\_state(LOAD3);  end  else begin  case(ld\_size)  uhalf: res <= dat32;  half: res <= {{32{dat32[31]}},dat32};  uchar: res <= dat16;  char: res <= {{48{dat16[15]}},dat16};  ubyte: res <= dat8;  byt: res <= {{56{dat8[7]}},dat8};  default: res[31:0] <= dat32;  endcase  next\_state(IFETCH);  end  end  end  LOAD3:  begin  wb\_read(radr);  next\_state(LOAD4);  end  LOAD4:  begin  if (ack\_i) begin  wb\_nack();  res[63:32] <= dat32;  case(1'b1)  isJSRix,isJMPix:  begin  pc <= res[31:0];  next\_state(IFETCH);  end  isRTS:  begin  pc <= res[31:0] + ir[15:8];  next\_state(IFETCH);  end  isPOP:  begin  wrrf <= `TRUE;  next\_state(DECODE);  end  default:  nextstate(IFETCH);  endcase  end  end |

Data for the loads comes from the bus and is multiplexed into the appropriate position for use by the load machine by the following code:

|  |
| --- |
| // Data input multiplexers  reg [7:0] dat8;  always @(dat\_i or radr)  case(radr[1:0])  2'b00: dat8 <= dat\_i[7:0];  2'b01: dat8 <= dat\_i[15:8];  2'b10: dat8 <= dat\_i[23:16];  2'b11: dat8 <= dat\_i[31:24];  endcase  reg [15:0] dat16;  always @(dat\_i or radr)  case(radr[1])  1'b0: dat16 <= dat\_i[15:0];  1'b1: dat16 <= dat\_i[31:16];  endcase  wire [31:0] dat32 = dat\_i; |

### Implementing Stores

Stores make use of a write address signal called ‘wadr’ which holds onto the write address.

The write address is calculated in the DECODE stage, the operation size is set according to the instruction in the st\_size signal. What to store is identified in the ‘store\_what’ signal. Depending on the instruction a handful of different items may be stored. Next a general purpose store state (STORE1) is called. One element from the DECODE case statement is shown below. The other cases are similar.

|  |
| --- |
| `SB:  begin  wadr <= a + imm;  st\_size = byt;  store\_what <= `STW\_B;  next\_state(STORE1);  end |

The general purpose store machine follows below. It takes care of regular store instructions, the PUSH instruction and the JSR instructions. The first thing it does is initiate a write cycle at the ‘wadr’ address with the appropriate data and size. There are four store states, the last two states are reached only for word sized operations. It takes two bus cycles to perform a word store because the bus interface is only 32 bits wide.

|  |
| --- |
| // ----------------------------------------------------------------------------  // STORE machine states.  // ----------------------------------------------------------------------------  STORE1:  begin  case (store\_what)  `STW\_A: wb\_write(st\_size,wadr,a[31:0]);  `STW\_B: wb\_write(st\_size,wadr,b[31:0]);  `STW\_C: wb\_write(st\_size,wadr,c[31:0]);  `STW\_PC: wb\_write(word,wadr,pc[31:0]);  `STW\_SR: wb\_write(word,wadr,sr[31:0]);  endcase  next\_state(STORE2);  end  STORE2:  begin  wb\_nack();  wadr <= wadr + 32'd4;  if (st\_size==word)  next\_state(STORE3);  else  next\_state(IFETCH);  end  STORE3:  begin  case (store\_what)  `STW\_A: wb\_write(word,wadr,a[63:32]);  `STW\_B: wb\_write(word,wadr,b[63:32]);  `STW\_C: wb\_write(word,wadr,c[63:32]);  `STW\_PC: wb\_write(word,wadr,32'h0);  `STW\_SR: wb\_write(word,wadr,32'd0);  endcase  next\_state(STORE4);  end  STORE4:  begin  wb\_nack();  case(1'b1)  isBRK:  begin  if (store\_what==`STW\_PC) begin  pc <= ir[39:8];  next\_state(IFETCH);  end  else begin  store\_what <= `STW\_PC;  next\_state(STORE1);  end  end  isJSR:  begin  pc <= ir[39:8];  next\_state(IFETCH);  end  isJSRix:  next\_state(LOAD1);  isPOP,isPUSH:  next\_state(DECODE);  default:  next\_state(IFETCH);  endcase  end |

### Implementing the Stack Pointer

The stack pointer needs to be updated at the same time that a register load from the stack is taking place. This would require two writes ports on the register file. In order to implement two writes at the same time, the stack pointer has it’s own register separate from the general register file. Reading this register is handled by the same multiplexor that puts the zero constant on the register output when R0 is read. Note that this multiplexor is repeated three times, once for each read port.

|  |
| --- |
| always @\*  case (Ra)  8’h00: rfoa <= 64’d0;  8’hFE: rfoa <= pc;  8’hFF: rfoa <= sp;  default: rfoa <= regfile[Ra];  endcase |

The stack pointer is always aligned at a word (8 byte) address. Loading the stack pointer clears the three least significant bits. The pointer increments and decrements by eight bytes during push and pop operations.

The first load of the stack pointer causes the global interrupt enable bit to be set.

|  |
| --- |
| // Update the register file  if (state==IFETCH || wrrf) begin  regfile[Rt] <= res;  if (Rt==8'hFF) begin  sp <= {res[63:3],3'b000};  gie <= `TRUE;  end  end |

### Implementing Stack PUSH / POP operations

First, POP is similar to PUSH so I’m only describing the PUSH code.

The push / pop operations would seem to need four ports on the register file, but they really only need a single port at a time. Some trickery is involved here as these instructions make use of the instruction register as a shift register. After the first register field of the instruction is used to access data, the instruction register is shifted over so that the next register field of the instruction is in the place of the first one. This causes a new register read from the register file.

The first thing done in the decode stage is to set a decode flag (isPUSH or isPOP) to true so that subsequent machine states know they are dealing with a PUSH or POP instruction. For a push operation, the write address is set to the decremented value of the stack pointer. The store opcode is set to store operand ‘a’. Note that the operand ‘a’ is loaded at the top of the DECODE stage, where other operands from the register file are loaded as well. What happens next depends on whether or not there is anything left to PUSH. If there is nothing left to push on the stack, the PUSH operation transitions back to the IFETCH stage. If an empty field is found in the IR, then it is skipped by transitioning back to the DECODE stage, otherwise control transfers to the STORE1 state. Ideally the assembler packs the registers so that all the empty fields are at the end of the instruction, in order to minimize the instruction execution time; but the processor doesn’t care if there is an empty field in the middle. If all the fields are empty, then the instruction acts like a NOP operation. Note that the PC increment is overridden unless the PUSH operation is transitioning back to the IFETCH state.

|  |
| --- |
| `PUSH:  begin  isPUSH <= `TRUE;  ir[39:8] <= {8'h00,ir[39:16]};  wadr <= sp\_dec[31:0];  store\_what <= `STW\_A;  if (ir[39:8]==32'h0)  next\_state(IFETCH);  else if (ir[15:8]==8'h00) begin  pc <= pc;  next\_state(DECODE);  end  else begin  pc <= pc;  sp <= sp\_dec;  next\_state(STORE1);  end  end |

## Implementing the Writeback Stage

### Implementing Register Updates

Registers are updated outside of the case statement where the remaining states are processed in order to allow register updates to occur during other stages besides the instruction fetch stage. If either the state is IFETCH or the write to registers flag (wrrf) is set, registers are written.

|  |
| --- |
| // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -  // Update the register file  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -  if (state==IFETCH || wrrf) begin  regfile[Rt] <= res;  if (Rt==8'hFF) begin  sp <= {res[63:3],3'b000};  gie <= `TRUE;  end  end |

# Instruction Set Description

A description of the instruction set follows.

## ADD - addition

ADD Rt, Ra, #i16

ADD Rt, Ra, Rb

ADDU Rt, Ra, #i16

ADDU Rt, Ra, Rb

Instruction Formats:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 048h | Rt8 | Rb8 | Ra8 | 028 | ADD Rt,Ra,Rb |
| Immediate16 | | Rt8 | Ra8 | 048 | ADD Rt,Ra,#imm |
| 148h | Rt8 | Rb8 | Ra8 | 028 | ADDU Rt,Ra,Rb |
| Immediate16 | | Rt8 | Ra8 | 148 | ADDU Rt,Ra,#imm |

Operation:

#### Register Immediate Form

Rt = Ra + immediate16

#### Register-Register Form

Rt = Ra + Rb

Notes:

The immediate constant may be extended up to 64 bits with immediate prefix instructions.

Currently the ADD and ADDU instruction both operate the same way.

## AND – bitwise logical ‘and’

AND Rt, Ra, #i16

AND Rt, Ra, Rb

Instruction Formats:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 208 | Rt8 | Rb8 | Ra8 | 028 | AND Rt,Ra,Rb |
| Immediate16 | | Rt8 | Ra8 | 0C8 | AND Rt,Ra,#imm |

Operation:

#### Register Immediate Form

Rt = Ra & immediate16

#### Register-Register Form

Rt = Ra & Rb

Notes:

The immediate constant may be extended up to 64 bits with immediate prefix instructions.

## ANDN – bitwise logical ‘and’ with complement

ANDN Rt, Ra, Rb

Instruction Formats:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 238 | Rt8 | Rb8 | Ra8 | 028 | AND Rt,Ra,Rb |

Operation:

#### Register-Register Form

Rt = Ra & ~Rb

Notes:

## ASR – Arithmetic Shift Right

ASR Rt, Ra, #i6

ASR Rt, Ra, Rb

Instruction Formats:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 448 | Rt8 | Rb8 | | Ra8 | 28 | ASR Rt, Ra, Rb |
| 548 | Rt8 | ~ | Imm6 | Ra8 | 28 | ASR Rt, Ra, #i6 |

Operation:

#### Register Immediate Form

Rt = Ra >> immediate6

#### Register-Register Form

Rt = Ra >> Rb

Notes:

Performs an arithmetic shift right, preserving the sign bit of the value.

## Bcc – Branches

Bcc Ra,target\_address

Instruction Formats:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 39 37 | 36 32 | 31 16 | 15 8 | 7 0 |  |
| ~3 | Disp5 | Addr16 | Ra8 | 4x8h | Bcc address |

Operation:

PC[15:0] = Addr16

PC[63:16] = PC[63:16] + Disp5

Notes:

Branches are page relative and absolute within a page. A branch is taken to the target address if the condition is true. Branches may branch forwards or backwards up to 1MB in range. The unused bits in the instruction should be set to zero.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | |  |  | |  |
| 40h | BEQ | branch if equal | 48h | BGT | branch if greater than |
| 41h | BNE | branch if not equal | 49h | BLE | branch if less or equal |
| 42h | BVS | branch if overflow set | 4Ah | BGE | branch if greater or equal |
| 43h | BVC | branch if overflow clear | 4Bh | BLT | branch if less than |
| 44h | BMI | branch if negative | 4Ch | BHI | branch if higher |
| 45h | BPL | branch if positive or zero | 4Dh | BLS | branch if lower or same |
| 46h | BRA | branch all the time | 4Eh | BHS | branch if higher or same |
| 47h | BNV | never branch | 4Fh | BLO | branch if lower |

## BRK – Breakpoint

BRK address

Instruction Formats:

|  |  |  |
| --- | --- | --- |
| Address32 | 008 | BRK |

Operation:

push (status register)

push (program counter)

PC = Address

Notes:

Perform an interrupt or exception handler. This instruction acts like the JSR instruction except that it also pushes the status register onto the stack. The BRK instruction is used by hardware interrupts to call a hardware interrupt processing routine. The RTI instruction should be used to return from the BRK handler.

## BSR – Branch to Subroutine

BSR target

Instruction Formats:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ~3 | Disp5 | Offset16 | ~8 | 568h | BSR address |

Operation:

#### Page Relative Address Form

SP = SP - 8

memory[SP] = PC

PC[15:0] = Offset

PC[63:16] = PC[63:16]+sign extend(Displacement)

Notes:

## CLI – Clear Interrupt Mask

CLI

Instruction Formats:

|  |  |  |  |
| --- | --- | --- | --- |
| 318 | ~24 | 018 | CLI |

Operation:

im = 0

Notes:

This instruction clears the interrupt mask, enabling interrupts.

## CMP - Comparison

CMP Rt, Ra, #i16

CMP Rt, Ra, Rb

Instruction Formats:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 068 | Rt8 | Rb8 | Ra8 | 028 | CMP Rt,Ra,Rb |
| Immediate16 | | Rt8 | Ra8 | 068 | CMP Rt,Ra,#imm |

Operation:

#### Register Immediate Form

Rt = flags of (Ra - immediate16)

#### Register-Register Form

Rt = flags of (Ra – Rb)

Notes:

The immediate constant may be extended up to 64 bits with immediate prefix instructions.

Compare performs both signed and unsigned comparison at the same time.

The most significant bit of the target register is set to the sign bit of the result. Branch instruction may branch on whether a register is minus or non-minus without performing a compare beforehand.

## COM – bitwise ones complement

COM Rt, Ra

Instruction Formats:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 068 | ~8 | Rt8 | Ra8 | 018 | COM Rt, Ra |

Operation:

#### Register-Register Form

Rt = ~Ra

Notes:

All the bits in Ra are inverted and placed into the target register Rt.

## DIV - Division

DIV Rt, Ra, #i16

DIV Rt, Ra, Rb

DIVU Rt, Ra, #i16

DIVU Rt, Ra, Rb

Instruction Formats:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 088h | Rt8 | Rb8 | Ra8 | 028 | DIV Rt,Ra,Rb |
| Immediate16 | | Rt8 | Ra8 | 088 | DIV Rt,Ra,#imm |
| 188h | Rt8 | Rb8 | Ra8 | 028 | DIVU Rt,Ra,Rb |
| Immediate16 | | Rt8 | Ra8 | 188 | DIVU Rt,Ra,#imm |

Operation:

#### Register Immediate Form

Rt = Ra / immediate16

#### Register-Register Form

Rt = Ra / Rb

Notes:

The immediate constant may be extended up to 64 bits with immediate prefix instructions.

The signed division operation takes 69 cycles to complete. The unsigned operation takes 68 cycles to complete.

## EOR – bitwise logical exclusive ‘or’

EOR Rt, Ra, #i16

EOR Rt, Ra, Rb

Instruction Formats:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 228 | Rt8 | Rb8 | Ra8 | 028 | EOR Rt,Ra,Rb |
| Immediate16 | | Rt8 | Ra8 | 0E8 | EOR Rt,Ra,#imm |

Operation:

#### Register Immediate Form

Rt = Ra ^ immediate16

#### Register-Register Form

Rt = Ra ^ Rb

Notes:

The immediate constant may be extended up to 64 bits with immediate prefix instructions.

## ENOR – complement bitwise logical exclusive ‘or’

ENOR Rt, Ra, Rb

Instruction Formats:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 268 | Rt8 | Rb8 | Ra8 | 028 | ENOR Rt,Ra,Rb |

Operation:

#### Register-Register Form

Rt = Ra ^ Rb

Notes:

There is no immediate form to this instruction.

## IMMx – Immediate Prefix

IMM1 #i32

IMM2 #i32

Instruction Formats:

|  |  |  |
| --- | --- | --- |
| Constant32 | FDh | IMM1 |
| Constant32 | FEh | IMM2 |

Operation:

IMM1: constant buffer = sign extend (immediate32)

IMM2: constant buffer[63:32] = immediate32

Notes:

The IMM1, IMM2 prefixes append onto the constant field of the following instruction. IMM1 may be used without IMM2 if the constant does not require 64 bits. If both prefixes are used they should be used in the order IMM1, IMM2. IMM1 and IMM2 prefixes lock out interrupts until the following instruction completes.

The IMM1 prefix sign extends an immediate constant found in a 32 bit immediate constant field in the instruction, to 64 bits and places the result into an internal constant buffer. The constant buffer is a non-visible internal buffer used by the processor to build large immediate constants. Typically a sixteen bit constant can be extended to forty-eight bits using just the IMM1 prefix.

The IMM2 prefix loads a 32 bit immediate constant into the upper half of the constant buffer leaving the lower half unchanged, overriding the previous sign extension of an IMM1 instruction. Combining an IMM2 instruction with an IMM1 instruction allows a 64 bit constant to be built in the buffer.

## JMP – Jump

JMP abs

JMP (abs,Rn)

JMP d(Rn)

Instruction Formats:

|  |  |  |  |
| --- | --- | --- | --- |
| 39 8 | | 7 0 |  |
| Address32 | | 508h | JMP address |
| 39 16 | 15 8 | 7 0 |  |
| Address24 | Ra8 | 528h | JMP (address,Rn) |
| Displacement24 | Ra8 | 548h | JMP d24(Rn) |

Operation:

#### Absolute Address Form

PC = Address32

#### Memory Indexed Indirect Form

PC = memory[address + Rn]

#### Register Indirect with Displacement Form

PC = displacement + Rn

Notes:

The address constant may be extended up to 64 bits with immediate prefix instructions.

## JSR – Jump to Subroutine

JSR abs

JSR (abs,Rn)

JSR d(Rn)

Instruction Formats:

|  |  |  |  |
| --- | --- | --- | --- |
| 39 8 | | 7 0 |  |
| Address32 | | 518h | JSR address |
| 39 16 | 15 8 | 7 0 |  |
| Address24 | Ra8 | 538h | JSR (address,Rn) |
| Displacement24 | Ra8 | 558h | JSR d24(Rn) |

Operation:

#### Absolute Address Form

SP = SP - 8

memory[SP] = PC

PC = Address32

#### Memory Indexed Indirect Form

SP = SP - 8

memory[SP] = PC

PC = memory[address + Rn]

#### Register Indirect with Displacement Form

SP = SP - 8

memory[SP] = PC

PC = displacement + Rn

Notes:

The address constant may be extended up to 64 bits with immediate prefix instructions.

## LB – Load Byte with Sign Extend

LB Rt, d(Rn)

LB Rt, o(Ra + Rb \* scale)

Instruction Formats:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Displacement16 | | | Rt8 | Ra8 | 808h | LB Rt,d16(Rn) |
| Offs6 | Sc2 | Rt8 | Rb8 | Ra8 | 888h | LB Rt,o6(Ra+Rb\*sc) |

Operation:

#### Register Indirect with Displacement Form

Rt = sign extend(memory[displacement + Ra])

#### Register-Register Form

Rt = sign extend(memory[offset + Ra + Rb \* scale])

Notes:

The displacement constant may be extended up to 64 bits with immediate prefix instructions.

|  |  |
| --- | --- |
| Sc2 Code | Multiply By |
| 0 | 1 |
| 1 | 2 |
| 2 | 4 |
| 3 | 8 |

## LBU – Load Byte with Zero Extend

LBU Rt, d(Rn)

LBU Rt, o(Ra + Rb \* scale)

Instruction Formats:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Displacement16 | | | Rt8 | Ra8 | 818h | LBU Rt,d16(Rn) |
| Offs6 | Sc2 | Rt8 | Rb8 | Ra8 | 898h | LBU Rt,o6(Ra+Rb\*sc) |

Operation:

#### Register Indirect with Displacement Form

Rt = zero extend(memory[displacement + Ra])

#### Register-Register Form

Rt = zero extend(memory[offset + Ra + Rb \* scale])

Notes:

The displacement constant may be extended up to 64 bits with immediate prefix instructions.

|  |  |
| --- | --- |
| Sc2 Code | Multiply By |
| 0 | 1 |
| 1 | 2 |
| 2 | 4 |
| 3 | 8 |

## LC – Load Character with Sign Extend

LC Rt, d(Rn)

LC Rt, o(Ra + Rb \* scale)

Instruction Formats:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Displacement16 | | | Rt8 | Ra8 | 828h | LC Rt,d16(Rn) |
| Offs6 | Sc2 | Rt8 | Rb8 | Ra8 | 8A8h | LC Rt,o6(Ra+Rb\*sc) |

Operation:

#### Register Indirect with Displacement Form

Rt = sign extend(memory[displacement + Ra])

#### Register-Register Form

Rt = sign extend(memory[offset + Ra + Rb \* scale])

Notes:

The displacement constant may be extended up to 64 bits with immediate prefix instructions.

|  |  |
| --- | --- |
| Sc2 Code | Multiply By |
| 0 | 1 |
| 1 | 2 |
| 2 | 4 |
| 3 | 8 |

## LCU – Load Character with Zero Extend

LCU Rt, d(Rn)

LCU Rt, o(Ra + Rb \* scale)

Instruction Formats:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Displacement16 | | | Rt8 | Ra8 | 838h | LCU Rt,d16(Rn) |
| Offs6 | Sc2 | Rt8 | Rb8 | Ra8 | 8B8h | LCU Rt,o6(Ra+Rb\*sc) |

Operation:

#### Register Indirect with Displacement Form

Rt = zero extend(memory[displacement + Ra])

#### Register-Register Form

Rt = zero extend(memory[offset + Ra + Rb \* scale])

Notes:

The displacement constant may be extended up to 64 bits with immediate prefix instructions.

|  |  |
| --- | --- |
| Sc2 Code | Multiply By |
| 0 | 1 |
| 1 | 2 |
| 2 | 4 |
| 3 | 8 |

## LDI – Load Immediate

LDI Rt, #i24

Instruction Formats:

|  |  |  |  |
| --- | --- | --- | --- |
| Immediate24 | Rt8 | 168h | LDI Rt,#imm |

Operation:

#### Register Immediate Form

Rt = immediate24

Notes:

The immediate constant is sign extended to 64 bits and loaded into the target register. The constant may be extended up to 64 bits with immediate prefix instructions.

## LH – Load Half-Word with Sign Extend

LH Rt, d(Rn)

LH Rt, o(Ra + Rb \* scale)

Instruction Formats:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Displacement16 | | | Rt8 | Ra8 | 848h | LH Rt,d16(Rn) |
| Offs6 | Sc2 | Rt8 | Rb8 | Ra8 | 8C8h | LH Rt,o6(Ra+Rb\*sc) |

Operation:

#### Register Indirect with Displacement Form

Rt = sign extend(memory32[displacement + Ra])

#### Register-Register Form

Rt = sign extend(memory32[offset + Ra + Rb \* scale])

Notes:

The displacement constant may be extended up to 64 bits with immediate prefix instructions. The memory address must be 32 bit (four byte) aligned.

|  |  |
| --- | --- |
| Sc2 Code | Multiply By |
| 0 | 1 |
| 1 | 2 |
| 2 | 4 |
| 3 | 8 |

## LHU – Load Half-Word with Zero Extend

LHU Rt, d(Rn)

LHU Rt, o(Ra + Rb \* scale)

Instruction Formats:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Displacement16 | | | Rt8 | Ra8 | 858h | LHU Rt,d16(Rn) |
| Offs6 | Sc2 | Rt8 | Rb8 | Ra8 | 8D8h | LHU Rt,o6(Ra+Rb\*sc) |

Operation:

#### Register Indirect with Displacement Form

Rt = zero extend(memory32[displacement + Ra])

#### Register-Register Form

Rt = zero extend(memory32[offset + Ra + Rb \* scale])

Notes:

The displacement constant may be extended up to 64 bits with immediate prefix instructions. The memory address must be 32 bit (four byte) aligned.

|  |  |
| --- | --- |
| Sc2 Code | Multiply By |
| 0 | 1 |
| 1 | 2 |
| 2 | 4 |
| 3 | 8 |

## LW – Load Word

LW Rt, d(Rn)

LW Rt, o(Ra + Rb \* scale)

Instruction Formats:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Displacement16 | | | Rt8 | Ra8 | 868h | LW Rt,d16(Rn) |
| Offs6 | Sc2 | Rt8 | Rb8 | Ra8 | 8E8h | LW Rt,o6(Ra+Rb\*sc) |

Operation:

#### Register Indirect with Displacement Form

Rt = sign extend(memory64[displacement + Ra])

#### Register-Register Form

Rt = sign extend(memory64[offset + Ra + Rb \* scale])

Notes:

The displacement constant may be extended up to 64 bits with immediate prefix instructions. The memory address must be 64 bit (eight byte) aligned.

|  |  |
| --- | --- |
| Sc2 Code | Multiply By |
| 0 | 1 |
| 1 | 2 |
| 2 | 4 |
| 3 | 8 |

## MOD – Signed Modulus

MOD Rt, Ra, #i16

MOD Rt, Ra, Rb

Instruction Formats:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 098h | Rt8 | Rb8 | Ra8 | 028 | MOD Rt,Ra,Rb |
| Immediate16 | | Rt8 | Ra8 | 098 | MOD Rt,Ra,#imm |

Operation:

#### Register Immediate Form

Rt = Ra mod immediate16

#### Register-Register Form

Rt = Ra mod Rb

Notes:

The immediate constant may be extended up to 64 bits with immediate prefix instructions.

## MODU – Unsigned Modulus

MODU Rt, Ra, #i16

MODU Rt, Ra, Rb

Instruction Formats:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 198h | Rt8 | Rb8 | Ra8 | 028 | MODU Rt,Ra,Rb |
| Immediate16 | | Rt8 | Ra8 | 198 | MODU Rt,Ra,#imm |

Operation:

#### Register Immediate Form

Rt = Ra mod immediate16

#### Register-Register Form

Rt = Ra mod Rb

Notes:

The immediate constant may be extended up to 64 bits with immediate prefix instructions.

## MUL – Signed Multiply

MUL Rt, Ra, #i16

MUL Rt, Ra, Rb

Instruction Formats:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0A8h | Rt8 | Rb8 | Ra8 | 028 | MUL Rt,Ra,Rb |
| Immediate16 | | Rt8 | Ra8 | 0A8 | MUL Rt,Ra,#imm |

Operation:

#### Register Immediate Form

Rt = Ra \* immediate16

#### Register-Register Form

Rt = Ra \* Rb

Notes:

The immediate constant may be extended up to 64 bits with immediate prefix instructions.

## MULU – Unsigned Multiply

MULU Rt, Ra, #i16

MULU Rt, Ra, Rb

Instruction Formats:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1A8h | Rt8 | Rb8 | Ra8 | 028 | MULU Rt,Ra,Rb |
| Immediate16 | | Rt8 | Ra8 | 1A8 | MULU Rt,Ra,#imm |

Operation:

#### Register Immediate Form

Rt = Ra \* immediate16

#### Register-Register Form

Rt = Ra \* Rb

Notes:

The immediate constant may be extended up to 64 bits with immediate prefix instructions.

## NAND – Complement Bitwise Logical ‘And’

NAND Rt, Ra, Rb

Instruction Formats:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 248h | Rt8 | Rb8 | Ra8 | 028h | NAND Rt,Ra,Rb |

Operation:

#### Register-Register Form

Rt = ~(Ra & Rb)

Notes:

There is no immediate form to this instruction.

## NEG – Negate

NEG Rt, Ra

Instruction Formats:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 058h | ~8 | Rt8 | Ra8 | 018 | NEG Rt,Ra |

Operation:

#### Register-Register Form

Rt = -Ra

Notes:

## NOP – No Operation

NOP

Instruction Formats:

|  |  |  |
| --- | --- | --- |
| Immediate32 | EA8 | NOP |

Operation:

none

Notes:

## NOR – Complement Bitwise Logical ‘Or’

NOR Rt, Ra, Rb

Instruction Formats:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 258h | Rt8 | Rb8 | Ra8 | 028h | NOR Rt,Ra,Rb |

Operation:

#### Register-Register Form

Rt = ~(Ra | Rb)

Notes:

There is no immediate form to this instruction.

## NOT – Not

NOT Rt, Ra

Instruction Formats:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 078h | ~8 | Rt8 | Ra8 | 018 | NOT Rt,Ra |

Operation:

#### Register-Register Form

Rt = !Ra

Notes:

If any bit in Ra is set the result in Rt is set to zero, if Ra is zero then Rt is set to one. The result in the register is thus either zero or one.

## OR – bitwise logical or

OR Rt, Ra, #i16

OR Rt, Ra, Rb

Instruction Formats:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 218h | Rt8 | Rb8 | Ra8 | 028h | OR Rt,Ra,Rb |
| Immediate16 | | Rt8 | Ra8 | 0D8h | OR Rt,Ra,#imm |

Operation:

#### Register Immediate Form

Rt = Ra | immediate16

#### Register-Register Form

Rt = Ra | Rb

Notes:

The immediate constant may be extended up to 64 bits with immediate prefix instructions.

## ORN – Bitwise Logical Or with Complement

ORN Rt, Ra, Rb

Instruction Formats:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 278h | Rt8 | Rb8 | Ra8 | 028h | ORN Rt,Ra,Rb |

Operation:

#### Register-Register Form

Rt = Ra | ~Rb

Notes:

There is no immediate form to this instruction.

## PHP – Push Processor Status

PHP

Instruction Formats:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 328h | ~8 | ~8 | ~8 | 018 | PHP |

Operation:

SP = SP - 8

memory[SP] = SR

Notes:

## PLP – Pull Processor Status

PLP

Instruction Formats:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 338h | ~8 | ~8 | ~8 | 018 | PLP |

Operation:

SR = memory[SP]

SP = SP + 8

Notes:

## POP – Pop Register

POP Ra,Rb,Rc,Rd

Instruction Formats:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Rd8 | Rc8 | Rb8 | Ra8 | A7h | POP {reglist} |

Operation:

If Ra <> 0

Ra = memory[SP]

SP = SP + 8

If Rb <> 0

Rb = memory[SP]

SP = SP + 8

If Rc <> 0

Rc = memory[SP]

SP = SP + 8

If Rd <> 0

Rd = memory[SP]

SP = SP + 8

Notes:

This instruction may be used to pop up to four registers from the stack.

## PUSH – Push Register

PUSH Ra,Rb,Rc,Rd

Instruction Formats:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Rd8 | Rc8 | Rb8 | Ra8 | A6h | PUSH {reglist} |

Operation:

If Ra <> 0

SP = SP - 8

memory[SP] = Ra

If Rb <> 0

SP = SP - 8

memory[SP] = Rb

If Rc <> 0

SP = SP - 8

memory[SP] = Rc

If Rd <> 0

SP = SP - 8

memory[SP] = Rd

Notes:

This instruction may be used to push up to four registers onto the stack.

## ROL – Rotate Left

ROL Rt, Ra, #i6

ROL Rt, Ra, Rb

Instruction Formats:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 41h | Rt8 | Rb8 | | Ra8 | 02h | ROL Rt,Ra,Rb |
| 51h | Rt8 | ~ | Imm6 | Ra8 | 02h | ROL Rt,Ra,#i6 |

Operation:

#### Register Immediate Form

Rt = Ra << immediate6

#### Register-Register Form

Rt = Ra << Rb

Notes:

Most significant bits are rotated into the least significant bits.

## ROR – Rotate Right

ROR Rt, Ra, #i6

ROR Rt, Ra, Rb

Instruction Formats:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 43h | Rt8 | Rb8 | | Ra8 | 02h | ROR Rt,Ra,Rb |
| 53h | Rt8 | ~ | Imm6 | Ra8 | 02h | ROR Rt,Ra,#i6 |

Operation:

#### Register Immediate Form

Rt = Ra >> immediate6

#### Register-Register Form

Rt = Ra >> Rb

Notes:

The least significant bits are rotated to the most significant bits.

## RTI – Return From Interrupt

RTI

Instruction Formats:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 40h | ~8 | ~8 | ~8 | 01h | RTI |

Operation:

PC = memory[SP]

SP = SP + 8

SR = memory[SP]

SP = SP + 8

Notes:

This instruction is used to return from an interrupt routine.

## RTS – Return From Subroutine

RTS #i16

Instruction Formats:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 39 32 | 31 16 | 15 8 | 7 0 |  |
| ~8 | SPOffset16 | ~8 | 60h | RTS |

Operation:

PC = memory[SP]

SP = SP + 8 + SPOffset

Notes:

This instruction is used to return from a subroutine. The stack pointer may be adjusted in order to remove parameters from the stack.

## SB – Store Byte

SB Rt, d(Rn)

SB Rt, o(Ra + Rb \* scale)

Instruction Formats:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Displacement16 | | | Rt8 | Ra8 | A08h | SB Rt,d16(Rn) |
| Offs6 | Sc2 | Rc8 | Rb8 | Ra8 | A88h | SB Rt,o6(Ra+Rb\*sc) |

Operation:

#### Register Indirect with Displacement Form

memory[displacement + Ra] = Rb

#### Register-Register Form

memory[offset + Ra + Rb \* scale] = Rc

Notes:

The displacement constant may be extended up to 64 bits with immediate prefix instructions.

|  |  |
| --- | --- |
| Sc2 Code | Multiply By |
| 0 | 1 |
| 1 | 2 |
| 2 | 4 |
| 3 | 8 |

## SC – Store Character

SC Rt, d(Rn)

SC Rt, o(Ra + Rb \* scale)

Instruction Formats:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Displacement16 | | | Rt8 | Ra8 | A18h | SC Rt,d16(Rn) |
| Offs6 | Sc2 | Rc8 | Rb8 | Ra8 | A98h | SC Rt,o6(Ra+Rb\*sc) |

Operation:

#### Register Indirect with Displacement Form

memory[displacement + Ra] = Rb

#### Register-Register Form

memory[offset + Ra + Rb \* scale] = Rc

Notes:

The displacement constant may be extended up to 64 bits with immediate prefix instructions.

|  |  |
| --- | --- |
| Sc2 Code | Multiply By |
| 0 | 1 |
| 1 | 2 |
| 2 | 4 |
| 3 | 8 |

## SH – Store Half-Word

SH Rt, d(Rn)

SH Rt, o(Ra + Rb \* scale)

Instruction Formats:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Displacement16 | | | Rt8 | Ra8 | A28h | SH Rt,d16(Rn) |
| Offs6 | Sc2 | Rc8 | Rb8 | Ra8 | AA8h | SH Rt,o6(Ra+Rb\*sc) |

Operation:

#### Register Indirect with Displacement Form

memory[displacement + Ra] = Rb

#### Register-Register Form

memory[offset + Ra + Rb \* scale] = Rc

Notes:

The displacement constant may be extended up to 64 bits with immediate prefix instructions.

|  |  |
| --- | --- |
| Sc2 Code | Multiply By |
| 0 | 1 |
| 1 | 2 |
| 2 | 4 |
| 3 | 8 |

## SHL – Shift Left

SHL Rt, Ra, #i6

SHL Rt, Ra, Rb

Instruction Formats:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 40h | Rt8 | Rb8 | | Ra8 | 02h | SHL Rt,Ra,Rb |
| 50h | Rt8 | ~ | Imm6 | Ra8 | 02h | SHL Rt,Ra,#i6 |

Operation:

#### Register Immediate Form

Rt = Ra << immediate6

#### Register-Register Form

Rt = Ra << Rb

Notes:

The least significant bits are loaded with zeros.

## SHR – Shift Right

SHR Rt, Ra, #i6

SHR Rt, Ra, Rb

Instruction Formats:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 42h | Rt8 | Rb8 | | Ra8 | 02h | SHR Rt,Ra,Rb |
| 52h | Rt8 | ~ | Imm6 | Ra8 | 02h | SHR Rt,Ra,#i6 |

Operation:

#### Register Immediate Form

Rt = Ra >> immediate6

#### Register-Register Form

Rt = Ra >> Rb

Notes:

The most significant bits are loaded with zeros.

## SW – Store Word

SW Rt, d(Rn)

SW Rt, o(Ra + Rb \* scale)

Instruction Formats:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Displacement16 | | | Rt8 | Ra8 | A38h | SW Rt,d16(Rn) |
| Offs6 | Sc2 | Rc8 | Rb8 | Ra8 | AC8h | SW Rt,o6(Ra+Rb\*sc) |

Operation:

#### Register Indirect with Displacement Form

memory[displacement + Ra] = Rb

#### Register-Register Form

memory[offset + Ra + Rb \* scale] = Rc

Notes:

The displacement constant may be extended up to 64 bits with immediate prefix instructions.

|  |  |
| --- | --- |
| Sc2 Code | Multiply By |
| 0 | 1 |
| 1 | 2 |
| 2 | 4 |
| 3 | 8 |

## SXB – Sign Extend Byte

SXB Rt, Ra

Instruction Formats:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 08h |  | Rt8 | Ra8 | 01h | SXB |

Operation:

#### Register Form

Rt = sign extend (Ra)

Notes:

The most significant bits (8 to 63) are loaded with the sign extension of bit 7.

## SXC – Sign Extend Character

SXC Rt, Ra

Instruction Formats:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 09h |  | Rt8 | Ra8 | 01h | SXC |

Operation:

#### Register Form

Rt = sign extend (Ra)

Notes:

The most significant bits (16 to 63) are loaded with the sign extension of bit 15.

## SXH – Sign Extend Half-Word

SXH Rt, Ra

Instruction Formats:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0Ah |  | Rt8 | Ra8 | 01h | SXH |

Operation:

#### Register Form

Rt = sign extend (Ra)

Notes:

The most significant bits (32 to 63) are loaded with the sign extension of bit 31.

# Glossary

## FPGA:

An acronym for Field Programmable Gate Array. FPGA’s consist of a large number of small RAM tables, flip-flops and other logic. These are all connected together with a programmable connection network. FPGA’s are ‘in the field’ programmable, and usually re-programmable. An FPGA’s re-programmability is typically RAM based. They are often used with configuration PROM’s so they may be loaded to perform specific functions.

HDL:

An acronym that stands for ‘Hardware Description Language’. A hardware description language is used to describe hardware constructs at a high level.

## Instruction Bundle:

A group of instructions. It is sometimes required to group instructions together into bundle. For instance all instructions in a bundle may be executed simultaneously on a processor as a unit. Instructions may also need to be grouped if they are oddball in size for example 41 bits, so that they can be fit evenly into memory. Typically a bundle has some bits that are global to the bundle, such as template bits, in addition to the encoded instructions.

## ISA:

An acronym for Instruction Set Architecture. The group of instructions that an architecture supports. ISA’s are sometimes categorized at extreme edges as RISC or CISC. Table888 falls somewhere in between with features of both RISC and CISC architectures.

## Program Counter:

A processor register dedicated to addressing instructions in memory. It is also often and perhaps more aptly called an instruction pointer. The program counter got it’s name because it usually increments (or counts) automatically after an instruction is fetched.

## SIMD:

An acronym that stands for ‘Single Instruction Multiple Data’. SIMD instructions are usually implemented with extra wide registers. The registers contain multiple data items, such as a 128 bit register containing four 32 bit numbers. The same instruction is applied to all the data items in the register at the same time. For some applications SIMD instructions can enhance performance considerably.

Stack Pointer:

A processor register dedicated to addressing stack memory. Sometimes this register is assigned from the general register pool. This register may also sometimes index into a small dedicated stack memory that is not part of the main memory system.

# Major Opcode Table

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | -0 | -1 | -2 | -3 | -4 | -5 | -6 | -7 | -8 | -9 | -A | -B | -C | -D | -E | -F |
| 0- | BRK |  | {RR} |  | ADD# | SUB# | CMP# | MUL# | DIV# | MOD# |  |  | AND# | OR# | EOR# |  |
| 1- |  |  |  |  | ADDU# | SUBU# | LD# | MULU# | DIVU# | MODU# |  |  |  |  |  |  |
| 2- |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 3- |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 4- | BEQ | BNE | BVS | BVC | BMI | BPL | BRA | BRN | BGT | BLE | BGE | BLT | BHI | BLS | BHS | BLO |
| 5- | JMP | JSR | JMP (,x) | JSR (,x) | JMP d(Rn) | JSR d(Rn) | BSR |  | BRZ | BRNZ | DBNZ |  |  |  |  |  |
| 6- | RTS |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 7- |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 8- | LB | LBU | LC | LCU | LH | LHU | LW |  | LBx | LBUx | LCx | LCUx | LHx | LHUx | LWx |  |
| 9- |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| A- | SB | SC | SH | SW |  |  | PUSH | POP | SBx | SCx | SHx | SWx |  |  |  |  |
| B- |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| C- |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| D- |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| E- |  |  |  |  |  |  |  |  |  |  | NOP |  |  |  |  |  |
| F- |  |  |  |  |  |  |  |  |  |  |  |  |  | IMM1 | IMM2 |  |

# Func Table for RR instructions

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | -0 | -1 | -2 | -3 | -4 | -5 | -6 | -7 | -8 | -9 | -A | -B | -C | -D | -E | -F |
| 0- |  |  |  |  | ADD | SUB | CMP | MUL | DIV | MOD |  |  |  |  |  |  |
| 1- |  |  |  |  | ADDU | SUBU |  | MULU | DIVU | MODU |  |  |  |  |  |  |
| 2- | AND | OR | EOR | ANDN | NAND | NOR | ENOR | ORN |  |  |  |  |  |  |  |  |
| 3- |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 4- | SHL | ROL | SHR | ROR | ASR |  |  |  |  |  |  |  |  |  |  |  |
| 5- | SHL # | ROL # | SHR # | ROR # | ASR # |  |  |  |  |  |  |  |  |  |  |  |
| 6- |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 7- |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 8- |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 9- |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| A- |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| B- |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| C- |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| D- |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| E- |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| F- |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

# Func Table for R instructions

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | -0 | -1 | -2 | -3 | -4 | -5 | -6 | -7 | -8 | -9 | -A | -B | -C | -D | -E | -F |
| 0- |  |  |  |  |  | NEG | COM | NOT | SXB | SXC | SXH |  |  |  |  |  |
| 1- |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 2- |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 3- | SEI | CLI | PHP | PLP | ICON | ICOFF | DCON | DCOFF |  |  |  |  |  |  |  |  |
| 4- | RTI |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 5- |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 6- |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 7- |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 8- |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 9- |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| A- |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| B- |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| C- |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| D- |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| E- |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| F- |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

## 01 Func Table

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Func8 |  |  | |  | | Opcode8 |  |
| Call Code32 | | | | | | 00h | BRK |
| Func8 |  | | Rt8 | | Ra8 | 01h | {R} |
| 05h |  | | Rt8 | | Ra8 | 01h | NEG |
| 06h |  | | Rt8 | | Ra8 | 01h | COM |
| 07h |  | | Rt8 | | Ra8 | 01h | NOT |
| 08h |  | | Rt8 | | Ra8 | 01h | SXB |
| 09h |  | | Rt8 | | Ra8 | 01h | SXC |
| 0Ah |  | | Rt8 | | Ra8 | 01h | SXH |
| 30h |  | |  | |  | 01h | SEI |
| 31h |  | |  | |  | 01h | CLI |
| 32h |  | |  | |  | 01h | PHP |
| 33h |  | |  | |  | 01h | PLP |
| 40h |  | |  | |  | 01h | RTI |

## 02 Func Table

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Func8 | Rt8 | Rb8 | | Ra8 | 02h | {RR} |
| 04h | Rt8 | Rb8 | | Ra8 | 02h | ADD Rt,Ra,Rb |
| 05h | Rt8 | Rb8 | | Ra8 | 02h | SUB Rt,Ra,Rb |
| 06h | Rt8 | Rb8 | | Ra8 | 02h | CMP Rt,Ra,Rb |
| 07h | Rt8 | Rb8 | | Ra8 | 02h | MUL Rt,Ra,Rb |
| 08h | Rt8 | Rb8 | | Ra8 | 02h | DIV Rt,Ra,Rb |
| 09h | Rt8 | Rb8 | | Ra8 | 02h | MOD Rt,Ra,Rb |
| 14h | Rt8 | Rb8 | | Ra8 | 02h | ADDU Rt,Ra,Rb |
| 15h | Rt8 | Rb8 | | Ra8 | 02h | SUBU Rt,Ra,Rb |
| 16h |  |  | |  | 02h |  |
| 17h | Rt8 | Rb8 | | Ra8 | 02h | MULU Rt,Ra,Rb |
| 18h | Rt8 | Rb8 | | Ra8 | 02h | DIVU Rt,Ra,Rb |
| 19h | Rt8 | Rb8 | | Ra8 | 02h | MODU Rt,Ra,Rb |
| 20h | Rt8 | Rb8 | | Ra8 | 02h | AND Rt,Ra,Rb |
| 21h | Rt8 | Rb8 | | Ra8 | 02h | OR Rt,Ra,Rb |
| 22h | Rt8 | Rb8 | | Ra8 | 02h | EOR Rt,Ra,Rb |
| 23h | Rt8 | Rb8 | | Ra8 | 02h | ANDN Rt,Ra,Rb |
| 24h | Rt8 | Rb8 | | Ra8 | 02h | NAND Rt,Ra,Rb |
| 25h | Rt8 | Rb8 | | Ra8 | 02h | NOR Rt,Ra,Rb |
| 26h | Rt8 | Rb8 | | Ra8 | 02h | ENOR Rt,Ra,Rb |
| 27h | Rt8 | Rb8 | | Ra8 | 02h | ORN Rt,Ra,Rb |
| 40h | Rt8 | Rb8 | | Ra8 | 02h | SHL Rt,Ra,Rb |
| 41h | Rt8 | Rb8 | | Ra8 | 02h | ROL Rt,Ra,Rb |
| 42h | Rt8 | Rb8 | | Ra8 | 02h | SHR Rt,Ra,Rb |
| 43h | Rt8 | Rb8 | | Ra8 | 02h | ROR Rt,Ra,Rb |
| 44h | Rt8 | Rb8 | | Ra8 | 02h | ASR Rt,Ra,Rb |
| 50h | Rt8 | ~ | Imm6 | Ra8 | 02h | SHL Rt,Ra,#i6 |
| 51h | Rt8 | ~ | Imm6 | Ra8 | 02h | ROL Rt,Ra,#i6 |
| 52h | Rt8 | ~ | Imm6 | Ra8 | 02h | SHR Rt,Ra,#i6 |
| 53h | Rt8 | ~ | Imm6 | Ra8 | 02h | ROR Rt,Ra,#i6 |
| 54h | Rt8 | ~ | Imm6 | Ra8 | 02h | ASR Rt,Ra,#i6 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | | | 03h | Reserved |
| Immediate16 | Rt8 | Ra8 | 04h | ADD Rt,Ra,#imm |
| Immediate16 | Rt8 | Ra8 | 05h | SUB Rt,Ra,#imm |
| Immediate16 | Rt8 | Ra8 | 06h | CMP Rt,Ra,#imm |
| Immediate16 | Rt8 | Ra8 | 07h | MUL Rt,Ra,#imm |
| Immediate16 | Rt8 | Ra8 | 08h | DIV Rt,Ra,#imm |
| Immediate16 | Rt8 | Ra8 | 09h | MOD Rt,Ra,#imm |
|  |  |  | 0Ah | Reserved |
|  |  |  | 0Bh | Reserved |
| Immediate16 | Rt8 | Ra8 | 0Ch | AND Rt,Ra,#imm |
| Immediate16 | Rt8 | Ra8 | 0Dh | OR Rt,Ra,#imm |
| Immediate16 | Rt8 | Ra8 | 0Eh | EOR Rt,Ra,#imm |