TABLE OF CONTENTS

PREFACE

WHO THIS BOOK IS FOR

This book is for the FPGA enthusiast who's looking to do a more complex project. It's advisable that one have a good background in digital electronics and computer systems before attempting a read. Examples are provided in the SystemVerilog language, it would be helpful to have some understanding of HDL languages. Finally, a lot about computer architecture is contained within these pages, some previous knowledge would also be helpful. If you're into electronics and computers as a hobby FPGA's can be a lot of fun. This book primarily describes the Qupls ISA. It is for anyone interested in instruction set architectures.

ABOUT THE AUTHOR

First a warning: I'm an enthusiastic hobbyist like yourself, with a ton of experience. I've spent a lot of time at home doing research and implementing several soft-core processors, almost maniacally. 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 25 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 diploma in electronics engineering technology. Some of the cores I work on these days are too complex and too large to do at home on an inexpensive FPGA. I await bigger, better, faster boards yet to come. To some extent larger boards have arrived. The author is a bit wary of larger boards. Larger FPGAs increase build times by their nature.

MOTIVATION

The author desired a CPU core supporting 128-bit floating-point operations for the precision. He also wanted a core he could develop himself. The simplest approach to supporting 128-bit floats is to use 128-bit wide registers, which leads to 128-bit wide busses in the CPU and just generally a 128-bit design. It was not the author's original goal to develop a 128-bit machine. There are good ways of obtaining 128-bit floating-point precision on 64-bit or even 32-bit machines, but it adds some complexity. Complexity is something the author must manage to get the project done and a flat 128-bit design is simpler.

Good single thread performance is also a goal.

Having worked on Thor2023 for several months, the author finally realized that it did not have very good code density. Having a reasonably good code density is desirable as it is unknown where the CPU will end up. Thor2022 was better in that regard. So, Thor2024 arrived and is a mix of the best from previous designs. Thor2024 aims to improve code density over earlier versions. Qupls code density is worse than Thor2024.

Some efficiency is being traded off for design simplicity. Some of the most efficient designs are 32-bit.

The processor presented here isn't the smallest, most efficient, and fastest RISC processor. It's also not a simple beginner's example. Those weren't my goals. Instead, it offers reasonable performance and hopefully design simplicity. It's also designed around the idea of using a simple compiler. Some operations like multiply and divide could have been left out and 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 a 64-bit design supporting 128-bit ops in part anticipating more than 4GB

of memory available sometime down the road. A 64-bit architecture is doable in FPGA's today, although it uses two or more times the resources that a 32-bit design would.

HISTORY

Qupls is a work in progress beginning November 2023. It is a major re-write from earlier versions. Thor which originated from RiSC-16 by Dr. Bruce Jacob. RiSC-16 evolved from the Little Computer (LC-896) developed by Peter Chen at the University of Michigan. The author has tried to be innovative with this design borrowing ideas from many other processing cores.

Qupls's graphics engine originate from the ORSoC GFX accelerator core posted at opencores.org by Per Lenander, and Anton Fosselius.

FEATURES OF QUPLS

- Fixed 40-bit instructions.
    - *The design has gone through several iterations of variable length instructions. A fixed length instruction set makes the design simpler and seems to require less hardware.*
- Four way out-of-order superscalar operation
- Four operating modes, machine, hypervisor, supervisor, and user.
- 64-bit data path, support for 128-bit floats
- 16 (or more) entry re-order buffer
- 32 general purpose registers. The register file is unified; it may contain either integer or float data.
- 24 general purpose vector registers
- Register renaming to remove dependencies, vector elements are also renamed.
- Dual operation instructions, Rt = Ra **op1** Rb **op2** Rc
- Standard suite of ALU operations, add subtract, compare, multiply and divide.
- Pair shifting instructions. Arithmetic right shift with rounding.
- Bitfield operations.
- Conditional branches with 20 effective displacement bits.
- 128 Entry Two-way TLB shared between data and code.

GETTING STARTED

To get started designing an ISA or CPU core some basic tools are required.

CHOOSING AN IMPLEMENTATION LANGUAGE

You will need a high-level hardware description language (HDL) of some sort to develop a processor.

It is a good idea to become accustomed to any number of languages. It helps to review the work of others and a lot can be learned by studying code from existing projects.

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. Additional languages include things like Java or C++ classes that people have developed to output HDL. Or language translators such as a 'C' to Verilog translator, for people who wish to work in 'C'. 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 System Verilog. VHDL is more verbose than Verilog and has tighter control of types. Qupls is implemented in the System 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 required to do the work you may have to roll some of your own. It can be quite an investment and it's up to you to decide. You have the power and control over your hobby. 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 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, 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 few 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. A spreadsheet 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

To produce an implementation some sort of FPGA developer tools will be required. The FPGA devices typically must be programmed with a bit file generated by tools supplied by the FPGA vendor. It's the vendors who know the requirements for programming their devices; I don't know of any third-party software that can generate bitstreams from source code. I've used both free toolsets from Altera and Xilinx.

SOFTWARE FOR THE TARGET ARCHITECTURE

The problem with an original home-grown processor is that there's no software for it. Fortunately, there is a lot of free software with source code available on the internet. One of the first things one will need is an assembler for the target architecture. One can assemble opcodes by hand with a reference chart handy, but it gets boring quickly. I usually end up doing some hand assembly to do some simple tests on the processor before the assembler is working. I then take an existing assembler and modify it for the new processor. One assembler I found on the net for the 6809 (listed in the resources section) was modified for a 6809-enhancement core. I have two assemblers one written in C++ the other in Visual Basic. Visual Basic's a little easier to work in for string handling. Some sort of text scripting language is a good place to start with a simple assembler. Some projects use Python. Much (older) software is written in C. It's a good language to know.

Once an assembler is working 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. Once some of the simpler pieces of software are working, one may want to try one's hand at a toolset.

There are several toolsets available that can be utilized during development of soft-core processors like Qupls. One of these is the LCC compiler. I used the LCC compiler for the Butterfly32 project. It's 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 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.

TESTING AND DEBUGGING

A lot of testing is required to get something working. 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. The key to avoiding backtracking and wasted time down the road is lots of testing along the way. Every bug fix is a test. When one bug is fixed, the next one shows up. Sometimes they seem like a two-headed hydra. Good testing skills are a requirement for developing and debugging a processor. Once you've managed to get such a thing working you're probably an ace at testing. Sometimes the processor and programming cannot help you to find a bug in the processor itself. You must 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 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 you're going to 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. A test bench is an artificial environment setup specifically to test a component. Inputs simulating a real environment are sent to the component then the output of the component is monitored for correctness. In the test bench usually so-called corner cases are tested, which are cases testing the extremes to which the component should work. If the component works in the extremes of the test bench it'll certainly work when it's put to real use is the general idea. A simulator is a tool built specifically for running test benches. The simulator has features to aid in debugging logic. One may set breakpoints, points which force the logic to stop at a particular place, and view the outputs of a component.

A simple test bench for the Thor divider circuit is shown below. Note that most test bench files don't have any input or output ports. Instead, signals are selected in the simulator for viewing.

In this case parameters for the divider were manually altered in the test bench to check for specific cases.

```
module Thor_divider_tb();
parameter WID=64;
reg rst;
reg clk;
reg ld;
wire done;
wire [WID-1:0] qo,ro;

initial begin
        clk = 1;
        rst = 0;
        #100 rst = 1;
        #100 rst = 0;
        #100 ld = 1;
        #150 ld = 0;
end

always #10 clk = ~clk;      //  50 MHz


Thor_divider #(WID) u1
(
        .rst(rst),
        .clk(clk),
        .ld(ld),
        .sgn(1'b1),
        .isDivi(1'b0),
        .a(64'd10005),
        .b(64'd27),
        .imm(64'd123),
        .qo(qo),
        .ro(ro),
        .dvByZr(),
        .done(done)
);
```

```
endmodule
```

Note that it is possible to automate test cases and even use file I/O in some tools. Test benches can become quite complex. Test benches for the float components often use a test input file containing the operands for the design under test, DUT, and output the results along with the input operands in a results output file. The output file can then be studied at leisure for issues to correct. Having a file output allows different revisions of the core to be compared and may make regression testing easier.

It is extremely unlikely that one would get the HDL 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 minor and well-known, the author runs simulator traces before attempting to run the code in an FPGA.

As a first test running software code in the FPGA try something 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.

This idea is popular for debugging hardware. The IBM PC had a “post-code” which was a byte value periodically written to an I/O port during startup for debugging. Depending on the display of the byte one could tell where in start-up it failed. Something like a missing or bad display adapter would end up with a specific code.

Another suggestion for test-benches is to use the actual system being loaded into the FPGA device as a component of the test-bench. If one keeps the system simple enough to start with then it’s possible to debug using the test-bench.

USING EMULATORS

An invaluable tool for debugging software prior to the processor being finished is the software emulator. A software emulator is an emulation of the device or system written as a software program to run on a workstation. Software emulators are often significantly slower than the real hardware. It’s also a tool where events applied to the system can be generated by user input. The code for the software emulation of a system mirrors the code for processor implementation itself. The code is just written in a different language. Having an emulator available allows for consistency checks between the emulation and the “real” device. Ideally the emulator should produce the same results as the real device would, except that it’s in a virtual environment of the emulator. The emulator can help resolve software problems that would be too difficult to do using the logic simulator.

Emulators can be cycle-exact, meaning they emulate what happens during each cycle of the processor’s clock. Cycle-exact emulators are often slower than non-cycle exact ones. An emulator that is not cycle exact may only emulate running software, interpreting object code, rather than performing all the internal operations that the CPU does.

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 well too). Many low-cost FPGA boards come with numeric LED displays for output and buttons for input. It's slightly more challenging to drive a numeric display and may make a good second test. 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 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 very simple, and pauses the whole machine waiting for a character.

DATA ALIGNMENT

Are your variables mysteriously getting over-written ? There could be a problem with address generation in the processor, or perhaps a problem with the external address decoding.

One approach to aligning data structures in memory is to ensure that the structures don't have partially overlapping addresses. This may help if there are memory addressing problems. For instance, if data structure addresses all end in xxx000, then if there is an address decoding problem, all the structures may get overwritten by values intended for other variables. If the variable addresses are somewhat mangled for example 0xxxx004,xx1018, xx2036 (ending in different LSB's) then it may be less likely for data to be corrupted. This is a temporary debugging approach. One would want to have the var's properly listed in a program.

GET RID OF COMPLEXITY

One of the best ways to be able to debug something is to get rid of all the extra complexities involved with it. Many is the time that the author has backtracked on a project and removed features in favor of getting something to work. Add one feature at a time, make it a component that can be easily disabled or removed from the design. Disable the complex features of the design. It's great to be able to do a complex design. But all the complicated stuff started out small and simple. One doesn't need caches, interrupts, branch predictors, and so on to have a working design. It's very rewarding to have even the simplest design working.

DISABLING INTERRUPTS

This bit only applies if you've managed to get some sort of interrupt facility working. Several smaller, simpler systems don't make use of interrupts. The original Apple computer did not use interrupts. Interrupts aren't something that one must get working right away. They would be part of a longer-term project goal (if at all). Start small and simple and expand from there. There are alternatives to interrupts the main one being polling in a loop.

When working with the real hardware having a set of switches available can be invaluable. The switches can be wired to key signals in the design to offer a manual override option. There may be times when one desires to disable a feature under development while other aspects of the project are taking place. For instance, eventually at some point in time one might want to venture into the world of interrupt processing. Interrupts are a challenge to get working. It's nice to be able to disable interrupts using an external switch. Also, 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. Development of an interrupt system is something for a later stage of development. Get the processor running longer sequences of code successfully first before trying to deal with interrupts.

THE IRQ LIVE INDICATOR

The IRQ live indicator is one of the first debug techniques the author uses once the core can run some code. 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. Another common approach is to use an LED to indicate the presence of IRQ's. Turning a LED on and off at a low frequency can be handy to visually detect the presence of IRQs.

DISABLE CACHING

This tip applies only if a cache is present. Implementing a cache isn't priority number one. The first few projects I did, did not include any caching. It was too complex to add a cache to begin with. 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 caching is enabled. If results seem flaky try disabling the cache.

CLOCK FREQUENCY

Be conservative when choosing a clock frequency. Don't try to run at the fastest possible frequency until the design is thoroughly debugged. Sometimes changing the clock frequency will provide clues to timing or synchronization problems. If the problem varies with a change in clock frequency, then maybe it's a timing problem. If the problem is consistent regardless of the clock frequency, it's likely some other problem. Note we are dealing with debugging probabilities here. Just because a problem is consistent at different clock frequencies doesn't mean it's not a timing problem.

Another nice aspect of a conservative clock frequency is that the tools used for building the system often work much faster if it's easy for the tools to meet the timing requirements. A conservative clock frequency is a way to speed up the development cycle.

MORE ADVANCED DEBUGGING OPTIONS

The following debugging mechanisms fall under the category of being more sophisticated in nature and more difficult to do, but they can sometime prove invaluable. They require interrupts or exceptions.

DEBUG REGISTERS

One option that aids primarily software debugging is the presence and use of debug registers. Adding debug registers to the core may make software debugging easier to do. Typically, there are one or more address matching registers that cause an interrupt or exception when the processor's program counter or data address matches the one in the debug register. One must have a working interrupt system for this to be usable.

TRACE / PROGRAM COUNTER HISTORY

One of the debug facilities that I've added to cores is the capability to capture the history of the program counter. While the processor is running at full speed, the program counter is stored in a small history table which is usually some sort of shift register. When an exceptional condition occurs in the processor core the history capture is turned off. In the exception processing routine, the program counter history can then be dumped to the screen showing where the program went awry.

The technique is called "trace". A good trace history will often be able to be triggered perhaps at a specific address or via debug match register. The trace may record all instructions, but it is common to record only the branch history, and then a few of the instruction addresses for synchronization purposes. Since branches are either taken or not taken a single bit can be used to record the history making trace very compact. With only a couple of block RAMs a trace history of thousands of instructions is possible.

STUCK ON A BUG ?

This is a brain trick. Try changing the code around the bug. Sometimes just by changing the code, refactoring without really changing operation, 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 20 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 a binary "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.

Bugs in toolsets are often obvious. The most recent one caused the toolset to crash and quit running depending on how simulation was started. There was a work-around by restarting the simulation fresh every time which takes longer than the usual restart.

If you suspect a bug in the toolset try searching the web for information on it. If it's a common problem it's bound to be posted on the web somewhere. There are also usually forums on the web where one can post about problems, and even sometimes get replies.

NOMENCLATURE

There has been some mix-up in the naming of load and store instructions as computer systems have evolved. A while ago, a "word" referred to a 16-bit quantity. This is reflected in the mnemonics of instructions where move instructions are qualified with a ".w" for a 16-bit move. Some machines referred to 32-bits as a word. Times have changed and 64-bit workstations are now more common. In the author's parlance a word refers to the word size of a machine, which may be 16, 32, 64 bits or some other size. What does ".w" or ".d", and ".l" refer to? To some extent it depends on the architecture.

The ISA refers to primitive object sizes following the convention suggested by Knuth of using Greek.

Number of Bits		Instructions	Comment
8	byte	LDB, STB	UTF8 usage
16	wyde	LDW, STW	
32	tetra	LDT, STT	
64	octa	LDO, STO	
128	hexi	LDH, STH	

The register used to address instructions is referred to as the instruction pointer or IP register. The instruction pointer is a synonym for program counter or PC register.

DESIGN CHOICES

For something as complex as a CPU there are many design choices to be made.

RISC VS CISC

No computer book would be complete without mentioning the RISC vs CISC paradigms.

There are two extremes to processor architecture. Most machines fall somewhere in-between. Qupls is somewhere in-between, leaning towards being a RISC machine. At the extreme end of RISC the architecture may support as little as single instruction, or just a handful like eight or sixteen. At the other extreme a CISC architecture may support thousands of instruction variants. RISC architectures are typically load/store, large register array, and few instructions of a fixed format size. CISC architectures tend to have memory operands, varying register array sizes, lots of instructions of varying formats and sizes. The goal behind a RISC architecture is high performance by using a simple processor that operates at a high clock frequency. The goal behind a CISC architecture is high performance by providing a more customized instruction set. CISC architectures may combine multiple operations into a single instruction in an attempt to increase performance. Examples include stack linkage instructions, looping constructs, and complex memory addressing modes.

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 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. The author is partial to little endian machines; it just seems more natural to him although he knows people who swear by the opposite. Whichever endian is chosen, often the machine has instructions(s) for converting from one endian to the other. The author does not bother with endian conversion; it's a feature that he probably wouldn't use. Some implementations even allow the endian of the machine to be set by the

user. This seems like overkill to the author. The endian of data is important because some file types depend on data being in little or big-endian format. Qupls is a little-endian machine.

ENDIAN

Qupls is a little-endian machine. The difference between big endian and little endian is in the ordering of bytes in memory. Bits are also numbered from lowest to highest for little endian and from highest to lowest for big endian.

Shown is an example of a 32-bit word in memory.

Little Endian:

Address	3	2	1	0
Byte	3	2	1	0

Big Endian:

Address	3	2	1	0
Byte	0	1	2	3

For Qupls the root opcode is in byte zero of the instruction and bytes are shown from right to left in increasing order. As the following table shows.

Address 3	Address 2	Address 1	Address 0
Byte 3	Byte 2	Byte 1	Byte 0

▼

31 19	18 13	12 7	6 0
$Constant_{13}$	$Raspec_6$	$Rtspec_6$	$Opcode_7$

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. The author wanted to be able to implement complex instructions easily using state machines. He has found non-pipelined designs easier to debug as well. The following chart shows the relationship ship between pipelining, clock frequency, and design complexity. It's based on the author's own experiences developing processor for FPGA's. It's a little bit of an Apple's to Orange's comparison, but it may be good for a general sense.

CPU	Max Clock Frequency	Clocks per Instruction	MIPS	Logic Cells	Processor Architecture
	100 MHz	3	33	2000	Sequential, non-overlapped
Raptor[64]	60 MHz	1.5	40	10000	Overlapped pipeline
Thor	40 MHz	0.75	53	100000	Superscalar 2 way
Qupls	40 MHz	0.75	53	160000	Superscalar 4-way

Note that what one chooses to do can depend on resource budgets of the whole system. If the cpu is going to stall waiting for a shared memory access most of the time, then it might as well be using multiple clock cycles to accomplish tasks. It doesn't matter how fast the cpu is if memory access is limited.

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. A well-known bus standard makes it possible to use peripheral cores developed by others.

As an example, 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 other projects. The external bus used by Table888 is 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 two word burst memory accesses to load the instruction cache. A burst access is several accesses that occur rapidly in a row in a sequence. 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.

Having mentioned the use of a standard bus, for Qupls the author decided to use his own bus standard with the goal of achieving higher performance. Qupls uses a bus called 'FTA bus' standing for Finitron Asynchronous bus.

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 many 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 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 forego to a byte-oriented design because I was interested in performance and planning to program in high level languages.

PLANNING FOR THE FUTURE

If one leaves no room for future instructions, it'll be difficult to upgrade the processor later. This has been a problem for several commercial processors. Table888's 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.

Qupls uses a seven-bit primary opcode. Of the 128 possible primaries there are about 20 open codes left that could be assigned.

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. A question I've heard from time to time is "How do I get my micro-controller to access more memory ?". Needing to access more memory is a common problem. What might be needed is a processor with greater memory accessing capacity. One can only shoehorn so much before the shoe splits.

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.

VARIABLE LENGTH INSTRUCTION SETS

One of the goals of a variable length instruction set is to minimize the number of bytes required to represent a program. Shorter code can sometimes execute faster because it makes better use of the cache. For embedded systems a shorter code may allow the use to smaller less expensive ROMs. Implementing a variable length instruction set adds some hurdles to the project. Instruction cache design for instructions varying in width is a challenge as well. A sample of a processor with varying sized instructions is the RTF65003 which makes use of a table to track instruction sizes. If choosing a variable length instruction set I'd advise setting up the instruction set so that the first few bits of the opcode can be used to determine

the instruction size. RISC-V processing core uses this approach. For the Thor core the size of the instruction can be determined primarily by looking at the opcode byte.

The author has found that decoding the length of an instruction for a variable sized instruction set can be on the critical timing path, at least for an FPGA based processor. He decided to use a fixed length encoding for Qupls to help improve timing.

INSTRUCTION BUNDLES

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, one solution in addition to having an instruction size of 40 bits would be to pack the 40-bit instructions into a 128-bit instruction bundle. A suggested bundle format:

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

The extra bits in the bundle would be used for debugging information. In some processors the extra bits serve a different function. For instance, in the IA64 architecture these are template bits which control the classes of instructions in the instruction slots.

For Qupls using an instruction bundle was seriously considered but ultimately discarded. Instead, a more complex instruction cache is used which allows instructions spanning cache-lines to be fetched. An issue with using a bundled format is that one may be restricted to processing instructions in specific groups. A non-bundled format can vary the number of instructions fetched and processing more easily.

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. Make an effort to find out what existing data formats are. If your application uses a specific type of data object, it's likely that someone else has already run into the same type of object. They may have encountered issues with using the object that one hasn't thought about yet.

REGISTERS

NUMBER OF GENERAL PURPOSE REGISTERS

Some research reveals that typically somewhere around 24 registers is a sweet spot for performance when dealing with high-level compiled languages. Machines with fewer registers start to suffer ill effects of moving data between registers and memory. Machines with more registers don't improve very much in performance over having 20 or so registers. Having more registers impacts the task switch time because they must be 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. If starting out small, it might be advisable to leave some means to extend the architecture with more registers.

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 chosen to use 32 registers. Another consideration is that within the FPGA memory resources are allocated in blocks. These blocks are typically 512 or 4096 bytes in size.

Yet another consideration is technical. Use of register renaming in the processor requires more registers than what appear to the programmer. About three times as many registers is a good number of rename registers. So, 96 registers are required for a 32-register machine.

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.

Qupls, as a four-way superscalar processor, has lots of register read ports. There are about 20 read ports in a smaller configuration of Qupls. This is to support executing multiple instructions at the same time. For instance, two ALUs require eight read ports, three for source operands and one for the target operand. With two ALUs, two FPUS, a flow control unit, and two data memory units a lot of ports are needed. Even more ports are required to support multiple write ports. To write multiple results at the same time requires multiplying the number of reads ports by the number of write ports.

SEGMENT REGISTERS

As part of the memory management portion of a cpu segment registers are often provided. There are usually multiple segment registers 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 an easy to understand and a low cost, low overhead memory management approach. The memory address from an instruction is added to a value from a segment register to form a final address. The segment register is often shifted left as it is added 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 the actual physical address of the program is much different. Once a

design seems to be working well, the author tends 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.

For Qupls the author decided not to include segment registers. Qupls uses a paged memory management unit and segment registers would be largely redundant.

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. Table888 has a handful of special registers that are accessed with the 'mtspr' (move to special register) and 'mfspr'. Qupls follows the convention of calling these registers CSRs for control and status registers. The CSRs are accessible with CSR register read and write instructions. The author encourages the reader to review the Qupls instruction set for these instructions, found later in the book.

MOVING REGISTER VALUES

A common operation is transferring data from one register to another. This operation is commonly done with a move instruction of some sort (MOV). Some simpler processors don't supply a register-to-register move operation. Instead, they rely on using another instruction that doesn't affect the data transfer, such as a register 'or' instruction. For example, or r1,r2,r0 effectively moves r2 to r1 because r1 is or'd with zero. It can be confusing looking at an assembly language dump, because it looks like there is an 'or' instruction. Another puzzle piece is that an explicit register move instruction uses only a single register read port. This is sometimes important in more advanced processors. Another related instruction that is less often used is the exchange registers instruction. Exchanging two registers can be tricky to implement because two register updates must take place. Exchanging registers is not always offered in processor architectures, when it is supported it is often a multi-cycle operation. Qupls supports an explicit register move instruction because the move operation can move between more registers than what is possible with other instructions like OR.

REGISTER USAGE

While the general-purpose register array may be considered general in nature, and any register may be used for any purpose, registers are often given specific usages by convention for software purposes. As far as hardware is concerned it doesn't care how general registers are used. But from a software perspective it is beneficial to assign specific registers to some tasks. For instance, often a general register is reserved for use by the operating system, meaning that application programs should not use it. This is a convention enforced by a compiler, not the hardware itself. Qupls has some basic register usage constraints.

HANDLING IMMEDIATE VALUES

First some background information. A significant proportion of instructions (for example 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 five 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) IMMxx – is an immediate postfix for the previous instruction
4) LW table – placing constants in a table
5) Half-operand or shifted operand instructions – instructions operating on only part of a register

Qupls architecture uses the last method listed.

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. A 64-bit processor may also require more set instructions to handle middle bits of a constant.
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 - PREFIX

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.

IMMXX - POSTFIX

Third solution: use an immediate postfix instruction. This is one of the author's favorites. For a long time, the Qupls design used postfix immediates. The constant postfix instruction simply contains the bits of the constant that wouldn't fit in the previous instruction. It looks like the following:

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

Advantages:

It requires less memory space as the postfix needs only to contain bits to specify an immediate. Often the postfix 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. The postfix also has the advantage that it can be read from the cache-line as if it were part of the current instruction. It is a little bit easier to process a postfix instead of a prefix.

Disadvantages:

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

LW TABLE

Fourth solution: place the large constants in a table in memory, then use regular load and store operations to load the constant into a register. This is the author's least favorite method. He is of the opinion that constants belong in the instruction stream and are better stored in the instruction cache rather than polluting the data cache. However, many systems use the load from table method.

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 OR SHIFTED OPERAND INSTRUCTIONS

Fifth solution: provide instructions that can operate on part 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.
3) Does not require instruction interlocks

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.

This is the method that Qupls uses to process large constants. Qupls includes several instructions that shift an immediate value by multiples of 20-bits before use.

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 must 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. The same paradigm was used for the Raptor64 processor. For the Table888 processor there is a 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.

Qupls uses combined compare-and-branch instructions to help reduce the dynamic instruction count. Effectively a compare operation and a branch operation are fused together.

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 what is 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 lot of software expects at least a sixteen-bit branch displacement. Eighteen effective bits is recommended. Qupls has effectively over 20 bits of displacement. 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.

As mentioned previously, Qupls has effectively a 20-bit displacement. The displacement constant is encoded as an 18-bit value, but it is in terms of the number of instructions are opposed to the number of bytes. Since instructions are five bytes in size the displacement in terms of bytes is five times as much. 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 most of the time. However, if one is using an automated code generator, the code generator may generate larger subroutines.

Unconditional branches in Qupls use a byte displacement value rather than an instruction displacement because subroutines could be located at any byte address. Unconditional branches are often used to enter or exit routines.

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.

Qupls includes two branch predictors, one called a branch-target-buffer operating at the fetch stage of the processor, and a second predictor called a gselect predictor operating at the decode stage of the processor.

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 some 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. Qupls supports increment-and-branch instructions. Incrementing a loop counter at the end of a loop is more common than a decrement.

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.

Qupls supports branch-to-subroutine, BSR, with 28-bit displacements which should be sufficient for most software. It also supports jumping to a subroutine at an absolute address, JSR. To use the full address range the target address of the subroutine must be loaded into a register before using JSR. Otherwise, the address range is limited to 21 bits.

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. Qupls return-and-deallocate instruction, RTD, returns from a subroutine, deallocates the stack, and allows a return a few instructions past the calling point.

SYSTEM CALLS

System calls are used to call the system. They are often called software interrupts or traps. The 80x88 uses the name 'int'. 6809 calls this a 'SWI' for software interrupt. In 6502 parlance it's the BRK instruction. They are called TRAPs on the 680x0 series. All these instructions do much the same thing. They are almost like a jump to subroutine instruction with an implied address. The system call instruction usually saves more machine state on the stack than a subroutine call would. These instructions may also switch the processor operating mode into a more protected level. Table888 calls this a break (BRK) instruction. Qupls uses the CHK instruction that always fails to perform system calls. CHK accepts a cause code argument encoded in the instruction which determines which exception vector will be invoked.

Qupls uses an internal state stack to store CPU state during a system call. This is much faster than using external memory. Multiple items such as the instruction pointer, and status register are stored on the state stack in a single clock cycle. This stack is small, only eight entries deep.

RETURNING FROM INTERRUPT ROUTINES

Like 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. Qupls uses the RTI also called RTE instruction to return from an interrupt or system call. This instruction loads both the instruction pointer and status register from the internal stack. A feature of the Qupls RTE instruction is the ability to perform a two-up return, returning twice from a system call which would otherwise be difficult to do since an internal state stack is used.

JUMPS

Strange as it may seem, unconditional jumps are 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.

CONDITIONAL MOVES

Conditional moves are available in many modern 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 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. Qupls has conditional moves in the form of the CMOVZ and CMOVNZ instructions. It also supports zero-or-set, or just plain set instructions which are also a form of conditional move.

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 they're 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 must be present and bypassed in an overlapped or superscalar design.

The Thor processing core features uses a whole byte for predicates, but gains back some of the opcode space by using redundant forms of the predicates as single byte instructions.

Qupls uses the PRED instruction modifier to perform predicated operations. The modifier may be applied before a group of instructions to be predicated. The author got the idea for the PRED modifier by browsing the comp.arch newsgroup and learning about its use in the My66000 CPU. Use of a modifier partly solves the issue of wasting instruction bits specifying predicate registers, and the issue of accessing another register for predicate operation. In Qupls any general-purpose register may be used as a predicate. Predicates may also be applied for vector instruction.

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 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 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. For Qupls any register may be used to store flag results.

DUAL OPERATION INSTRUCTIONS

Dual operation instructions are not commonly done. Qupls uses them because there are instruction bits available to represent them and they are occasionally useful to reduce register usage and instruction counts. A dual operation instruction performs two operations on data instead of one. The first operation is performed between two source registers followed by a second operation on the result of the first and an additional source register. An example of a dual-operation instruction is the AND_OR instruction.

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 in Qupls with the ADD, SUB, MUL, and DIV instructions.

There are both signed and unsigned versions of the arithmetic operations.

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.

Qupls 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.

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.

In Table888 the shift immediate instructions are implemented as a subset of the RR (register to register) 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.

Qupls shift instructions have their own instruction group, more opcode bits are used as the shift instructions may shift pairs of registers. This is done in some architectures as it allows the implementation of rotate operations. The Qupls arithmetic shift right instruction, ASR, features rounding options on the result.

MYSTERY OPERATIONS

There is a class of instructions available on some processors that I like to call ‘Mystery Operations’. For a mystery operation the operation to be performed isn’t known until runtime, and hence is a mystery. This class of instructions is present to aid in the avoidance of writing self-modifying code. In some cases, it’s desirable to control the code itself without resorting to complex (and slow) branching. For instance, in a graphics plot routine, it may be desirable to control the raster operation (AND, OR, XOR, COPY, etc.). Rather than use a case statement with many branches, instead the raster operation code is loaded into a register. The program then executes the code from the register rather than branching. Code executed with mystery op’s can run substantially faster than code using branches.

However, mystery operations are not typically available in modern CPUs they can wreck-havoc on a pipeline depending on how they are implemented. Instead, other means like just-in-time, JIT, compilation are used.

OTHER INSTRUCTIONS

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. Many high-level languages do not support bit-fields. The 'C' language does support bitfields. The arpl compiler directly supports bitfield operations on primitive data types.

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 straightforward to implement, however they take up a lot of room because of the parallel hardware. 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. The Qupls ISA supports SIMD instructions. Most instructions have a precision field that indicates how to treat values in registers.

String instructions. String type operations include block moving, block set, and block compare operations. The 80x88 has some 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.

HARDWARE INTERRUPTS

Hardware interrupts are in some ways like software exceptions and many 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. Software exceptions are usually *synchronous*, occurring when a specific instruction is executed. Hardware interrupts are *asynchronous* events. 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. Qupls also supports hardware interrupts and software interrupts with the CHK instruction. Having made a

big boo about hardware interrupts it should be noted that it's possible to get by without them. The original Apple machine didn't make use of interrupts. Even something as sophisticated as the Apple Macintosh used a system of co-operative multi-tasking rather than interrupt driven tasking. It's entirely possible to setup a decent polling system, many embedded systems work this way.

INTERRUPT VECTORING

A design question to answer is "how does the processor know where to go when an interrupt occurs " ? About the simplest mechanism to use is to have the processor vector to code at fixed addresses when an interrupt occurs. This mechanism is used by many RISC processors. This is like "when hardware interrupt #1 occurs, go to address $100, when hardware interrupt #2 occurs go to address $200, and so on. The original Table888 used a variation of this method, where the upper address bits were determined by another register in the processor. The z80 uses a similar mechanism.

A slightly more sophisticated method of determining the vector address is to use an interrupt vector table. The vector table contains a list of addresses of where to vector to for a given interrupt. This mechanism is used on a lot of processors including but not limited to the x86 series, the 68x00 series, SPARC and even many 8-bit machines like the 6502, 6800, and 6809.

INTERRUPT VECTOR TABLE

The interrupt vector table is a table full of addresses of the interrupt routines. The vector table may be located at a fixed memory address meaning, usually that's where the system ROM would be placed. Many eight-bit machines have a fixed address for this table. In a slightly more advanced, and more expensive system, the location of the interrupt vector table is relocatable in memory via an interrupt base address register. This is one piece in providing the capability of writing hyper-visor's for the machine. Table888 calls this register the VBR (vector base register).

In Table888 the interrupt vector number supplied by the BRK instruction indexes into the interrupt vector table in order to determine which vector to load. The BRK instruction acts like a memory indirect jump then. There are 512 interrupt vectors allowed for by Table888.

Qupls uses a separate vector table for each operating mode of the processor. The location of vector tables is stored in one of the TVEC CSRs. The exception vector table and exceptions in general for Qupls are outline in the Qupls specific section of the document.

GETTING AND PUTTING DATA

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 the author 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 un-cached memory load / store instructions. Some designs have explicit un-cached 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 wyde. 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 tetra(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. The author finds push / pop instructions easier to read and understand while reading code. Is that store for a subroutine push ? or a general memory op ? Explicit push and pop instructions may also have better code density if they can support pushing and popping multiple registers with a single instruction.

ALIGNED AND UNALIGNED MEMORY ACCESS

Memory access alignment is an issue that crops up on a machine supporting multiple data sizes larger than a byte. On a machine that's byte addressable with varying data sizes, one must decide how to support unaligned memory accesses. If the data-bus size of the machine is eight bytes wide, a word access could potentially start at any one of those bytes. If the access starts at any byte other than zero then it would wrap around into the next memory word. This is called an unaligned access. Directly supporting unaligned memory accesses requires multiple bus accesses for unaligned data. This isn't too bad to do in a state machine driven design, but it's difficult to do in an overlapped pipelined design. Many machines simply stipulate that unaligned memory access is not allowed. Other machines implement unaligned accesses using traps where software can implement the unaligned access (this makes unaligned memory access quite slow). Table888 does not currently support unaligned memory accesses. Because the bus size is only 32 bits in the current implementation there is a potential to easily allow words to be half-word aligned in memory. Qupls ISA supports unaligned memory access.

LOAD / STORE MULTIPLE

With a machine with a lot of registers there is often a means to load or store more than a single register at a time. For instance, the PowerPC has a LMW instruction standing for 'load multiple words'. Table888 supports loading and storing multiple registers at the same time with the LMR (load multiple registers) and SMR (store multiple registers) instructions. The range of registers between Ra and Rb is loaded or stored to an address identified by Rc. Part of the value of the LMR and SMR instruction is that they can save considerable cache space over having many independent load / store instructions. For example, one might use 16 SMR instructions rather than 256 SW instructions to save the register state during a task switch. Qupls supports loading and storing multiple registers with the PUSH, POP, PUSHA, POPA, LDCTX and STCTX instructions.

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 R31 is used as the stack pointer. The stack may be used to pass parameters to functions. There are instructions supporting stack operations which include PUSH and POP.

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 like Table888. 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. In Table888 the push / pop instructions push or pop any four of 256 registers. Qupls allows up to five registers to be pushed or popped. Note that the same register may be pushed or popped multiple times with the instruction.

A push tends to be used more often than pops. Instead of popping arguments off the stack after a subroutine call, usually the stack pointer is simply incremented because we don't care about getting the argument values back. Adding onto the stack pointer turns the pop operation into a single instruction, and often a single cycle operation, rather than a series of memory operations.

One consideration for the POP instruction is whether to support popping multiple items with one instruction. It adds a level of complexity to the processor to pop more than one item per instruction because most other instructions only update one register. Register write ports are expensive so often there's only a single write port to the register array. That means doing multiple updates to the register array requires multiple clock cycles. In an overlapped pipelined design, it's desirable to stick to a single register update per instruction. Providing for multiple register updates makes the design really complex. If I were going to turn Table888 into an overlapped pipelined design one thing I would seriously consider is limiting the pop instruction to a single register.

Qupls handles popping multiple values off the stack by implementing pop as a macro instruction. The pop instruction invokes micro-code which uses a separate load instruction for each value popped from the stack.

DATA CACHING

Qupls has a 64kB 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 critical applications data isn't even allowed to be cached.

There are several policies associated with caches. A given cache may implement multiple policies as options. I mention the most prominent ones here.

A write-back cache policy delays the writing of data back to main memory until the dirty cache line is dumped. This may be when the data cache controller decides it's a good time to update memory, and also when a new incoming cache data would replace the dirty line. The cache is updated immediately on a data store operation, but main memory is not.

A write-through cache policy updates the data in both the cache (if it is in the cache) and main memory immediately.

A write-allocate cache policy is a cache that loads the cache line from memory when a write cycle takes place.

ADDRESS MODES:

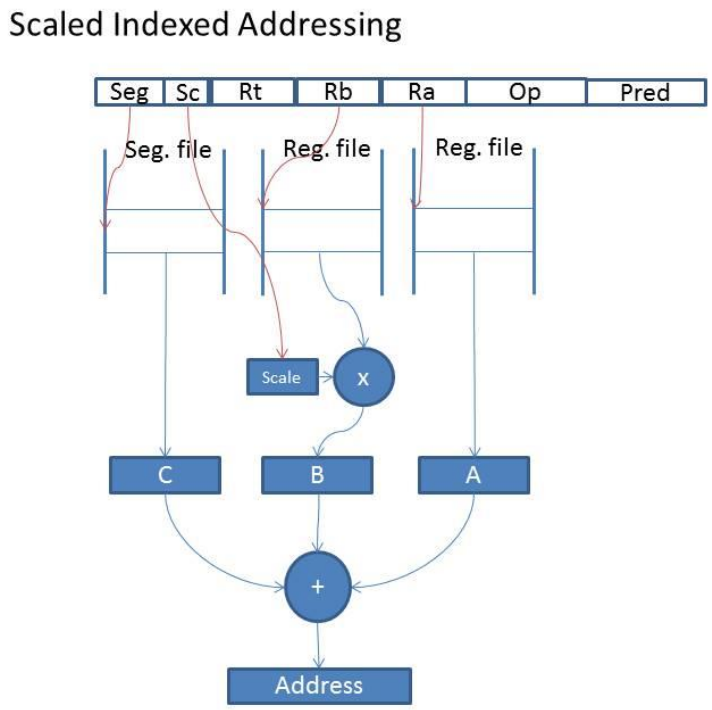
A point of sale from a marketing perspective in the past has been the number and type of address modes available in the processor. "Use any address mode with any instruction." was a statement about the

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 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. Table888 implements two address modes for load and store instructions. 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. Qupls, like Table888 has both register indirect with displacement and scaled indexed addressing.

SCALED INDEX ADDRESSING

Indexed addressing with a scaled index register works by adding two registers together with an offset 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 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. A compiler will output the necessary multiply and add instructions to do indexing. It's a bit of a trick to get the compiler to use scaled indexing which only applies when the object size is 2,4, or 8 bytes in size. 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 eleven bits. If a larger offset or displacement is required it will have to be managed using registers.

The following diagram shows how scaled indexed addresses are formed. The diagram is pertinent to the Thor processing core and so shows a predicate field in the instruction, but illustrates the address formation.



REGISTER INDIRECT WITH DISPLACEMENT ADDRESSING

The other addressing mode that is highly useful is register indirect with displacement. In this address mode a register is added to a displacement 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 21 bits in size.

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 many 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.

Table888 did not support a compare-and-swap or other atomic memory operations. Instead, semaphores 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.

Qupls supports compare-and-swap and other atomic memory operations.

MEMORY MANAGEMENT

SEGMENTATION AND PAGING

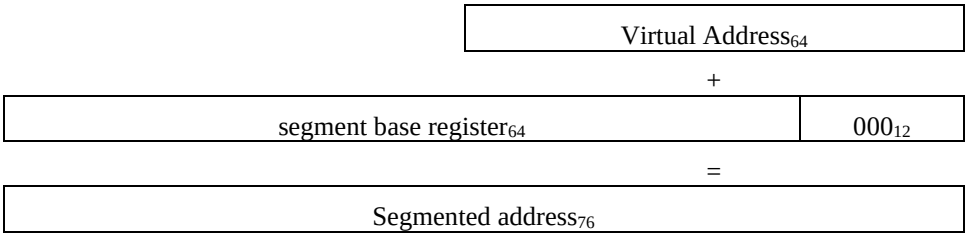
Segmentation and paging are the two main choices for memory management beyond some simpler mechanisms like bank switching. The goal for Table888 was to implement both. Several commercial processors implement both segmentation and paging. Although segmentation has fallen out of favor somewhat it is still used. Typically, the segmentation part of the cpu has a handful of segment registers loaded with a flat memory model, then is for the most part ignored. One place where segmentation is still used in a modern OS is in establishing the global storage area, and the thread local storage area.

SEGMENTATION OVERVIEW

As part of the memory management portion of a cpu segment registers are often provided. There are usually multiple segment registers 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 or the PA-RISC machine also use them as well. Segment registers are an easy to understand, and a low-cost memory management approach. The memory address from an instruction is combined (added) to a value from a segment register to form a final address. The segment register is often shifted left as it is added 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 the actual physical address of the program is much different. Once a design seems to be working well, the author tends to add segment registers to the design as a first step at providing memory management features. Table888mmu uses sixteen segment registers. Table888mmu's segmentation system is modelled after the INTEL 80286 segmentation model. The basic concepts are the same, but the layout and size of fields has been altered. Qupls does not use segmentation.

ADDRESS FORMATION

The virtual address is added to a segment base register to form a final address.



The Table888mmu sample shifts the segment base register left by 12 bits before adding it to the virtual address. The resulting segmented address could be 76 bit is size, however fewer bits are implemented in the sample.

The number of bits shifted to the left is referred to as the paragraph size.

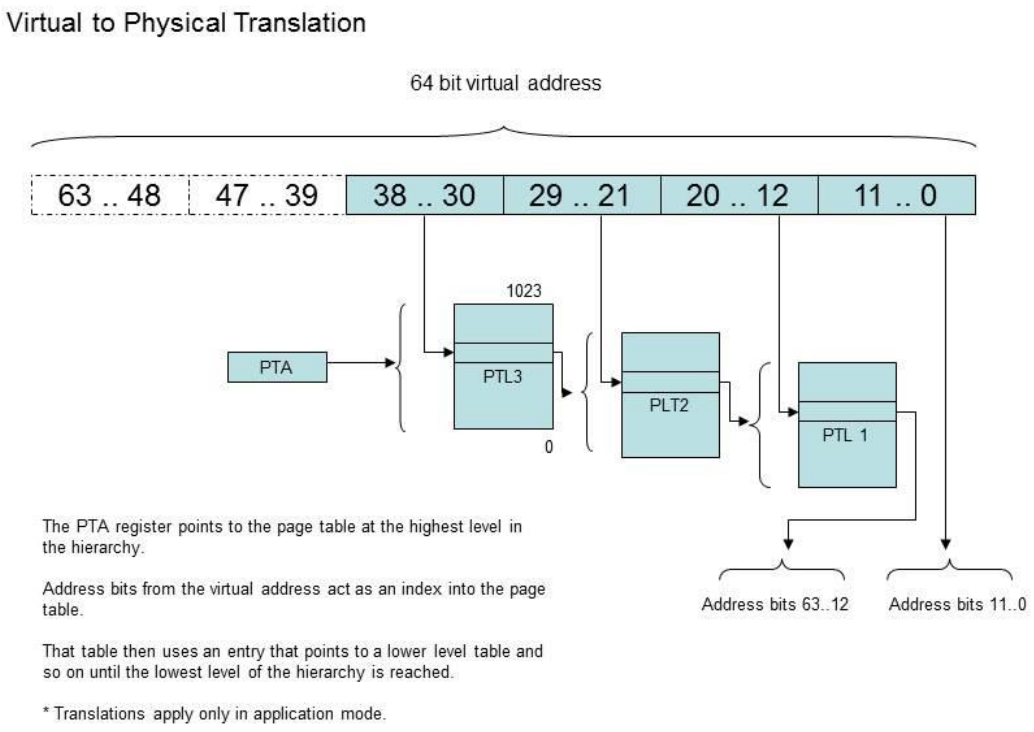
NUMBER OF REGISTERS

The number of segment registers that are useful seems not to be quantified as closely as the number of general-purpose registers. However, four registers were deemed not enough for the 80x86 architecture and two more segment registers were added. Also, a couple of additional registers in the 80x86 design were added to support the segmentation architecture and they act a lot like segment registers. These include the task register and the local descriptor table register. So, we have about eight segment registers in the 80x86 architecture. PA-RISC uses eight “space” registers that act a bit like segment base registers. PowerPC uses an array of sixteen registers. Table888 uses sixteen segment registers. Segments registers are typically initialized to a flat memory model then forgotten about.

The segment registers in Table888mmu contain a selector which is 24 bits in size. This size was chosen as they are used as an index into a segment descriptor table. The segment descriptor table in this case contains a maximum of 512k entries.

PAGING OVERVIEW

Paging uses a set of tables to perform mapping of virtual addresses to physical ones. Unlike segmentation, paging cannot resolve maps right down to individual bytes. Instead, memory is broken up into pages and managed on that basis. A typical page size is 4kB. The virtual address is divided up and each part of the virtual address is used to index into a table.



The table at the highest level of the hierarchy is usually permanently resident in the computer's memory for performance reasons. Because there is a fair amount of work to be done in mapping addresses, address mappings are usually cached in an additional until called a translation look-aside buffer (or TLB). This unit is also sometimes called an Address Translation Cache (ATC). A paging system tends to have more overhead associated with it compared to a segmented system.

One of the nice features of paging is that it is almost invisible from a software perspective. There aren't any registers like segment registers, to worry about when paging is active. Paging simply works behind the scenes.

The Qupls paging system can map the entire 64-bit address space. A multi-level system of page directories and subdirectories is used. In most cases the address space mapped will be less than a full 64-bit address space. The paging system accommodates this by using a smaller directory hierarchy. For instance, if an application is less than 512B in size, a single level page system is used. If the application can fit within 4TB only two levels are required. The depth of the directory system is controllable on an application basis. The page memory management unit takes care of walking the page tables in hardware to find a translation. Translations are stored in a translation look-aside buffer (TLB) which is a translation cache, so that the page tables don't have to be walked for every translation.

REGISTERS

The primary register that controls paging is the page table address register (PTA), page table base register, PTBR or as it is alternately called control register number three. (CR3). This register contains the base

address of the root page table in memory. Once the PTBR is set, the processor knows where to begin looking up virtual to physical address translations.

PAGE TABLES

Page tables are the central piece of a paging system. There are two types of page tables, page directory tables and page leaf tables. For Qupls page directory tables are 64kB in size and contain 8192, 8-byte entries. Page Leaf tables are also 64kB in size and contain 8192, 8-byte entries. Details of the Qupls paged memory management system are available later in the book. The paged mmu knows where in the hierarchy the page table is, so it can determine if the table is a directory table or a leaf table. Page tables at the bottom of the hierarchy are always leaf tables.

NOP RAMPS

A brief word about NOP ramps used with the Tabl888 architecture. It shows that sometimes software can aid resolving hardware issues.

In a paged memory management unit, address translations take place continuously, not just at segment load time as would be for a segmentation unit. One nice feature about segmentation is that one can be sure that if the segment is loaded it is a continuously available memory range.

With paging on the other hand, page faults may occur at 4kB boundaries. When the page isn't present in memory, it must be loaded then the instruction can be executed. In Table888 most instructions are a single instruction word in length, so they won't cross a page boundary. However, there are several prefix instructions, when combined with a prefix an instruction might cross a page boundary. This is bad news. The problem is that the prefixing would get lost in the shuffle to move the missing page into memory. There are two solutions, one is to go back a page in memory and re-execute the prefix after the missing page is loaded. This is complicated by the fact that both pages are required to be present in memory, otherwise the processor would thrash back and forth trying to execute the instruction. In Table888 it is safe to re-execute the prefix instructions as they don't modify the processor's state until the following instruction is executed. That means going back a page and re-executing the prefixed instruction is simple to do. The second solution is to force the instruction stream to output the prefixes so that they don't cross page boundaries. This can be handled by the assembler. The assembler handles the occasional case where a prefix instruction would cause an instruction to span a page boundary by outputting a series of NOPs to force the instruction onto the next memory page. The following example shows that the prefix instruction to a store byte operation is forced onto the next page of memory.

```
00008FF0     41 F8 2A 90 00            bne          fl0,kbdi2
00008FF5     16 01 24 00 00            ldi          r1,#36
00008FFA     EA EA EA EA EA       ; imm
00009000     EA EA EA EA EA
00009005     FD 70 FF 03 10
0000900A     A0 00 01 00 18            sb           r1,LEDS
```

Note that the NOP ramp's won't work if address space defined by a segment is paged out of memory and the segment isn't aligned on a 4kB boundary. This shouldn't be the case in Table888 as the segment paragraph size is the same as the paged mmu page size.

PROTECTION MECHANISMS

Table888's protection mechanism is like that of the 80x86 series. If you are comfortable with the 80x86 protection mechanism then Table888's mechanism will seem familiar. The format of descriptors is different, but the fields are similar. Table888 has sixteen protection ring levels.

Qupls reduces the number of rings to four, called operating modes.

Often there are hardware supported protection mechanisms for a given architecture. Many architectures are bi-modal, with kernel / system / supervisor modes and user/application modes. Even processors supporting more modes are often used in a bi-modal fashion by the OS. The x86 series processor has four levels of privilege. Usually, all the features of the processor are available in kernel mode, while a subset of features are available in application mode. Limiting application code to a subset of the processor features is one way of protecting the system. Qupls supports 256 privilege levels in addition to having four operating modes.

The segment limit in a segmented system protects against memory access outside of the segment. If an access is attempted beyond the limit a bounds violation exception occurs. If the segment descriptors are only accessible from kernel mode, then application code can't modify them to gain access to data outside of the segment.

PROTECTION RULES

Code cannot access data at more protected level. This is checked by comparing the processor's current privilege level to the privilege level of the segment that is pending access. The check is performed when a segment register is loaded using the mtspr instruction. In paged MMU system the check is performed when the page is accessed.

Code at a high protection level cannot call code at a lower level. In Table888's case this is checked when a jump to subroutine or jump instruction is executed with a jump selector prefix. For Qupls, a CHK instruction is used to switch protection levels.

The code segment may only be loaded with code type descriptors. If an attempt is made to load a data descriptor into a code segment a segment type violation exception occurs.

The data segments may only be loaded with data type descriptors. If an attempt is made to load a code segment into a data segment a segment type violation exception occurs.

The protection rules refer to a privilege level. The current privilege level of the processor is maintained in the status register, this is called the CPL.

TRIPLE MODE REDUNDANCY (TMR)

Triple mode redundancy is a feature to improve the reliability of the system. Operations are performed in triplicate and majority logic used to determine the correct results.

The Table888 MMU features a triple redundancy mode that was spawned when it was believed that problems due to bad memory were cropping up. In triple redundant mode memory loads and stores are performed in triples. Ideally each load/store operation of the triple goes to a different memory bank. For example, a store operation stores the data to an address in memory bank#1, then repeats the store to memory bank#2 and #3. Part of the output address is supplied by a two-bit counter which selects the DRAM bank. This reduces the effective amount of memory that the processor sees by a factor of four. The triple redundant memory loads and stores also use three times as many memory cycles, and so slow the program down considerably.

During a load operation, the operation is repeated three times, each time loading into a separate load buffer. When all three loads are complete the values are compared using majority logic. Whichever values are the most common (2 or all 3 bits the same) are deemed the correct values.

There is currently no provision to correct values in memory if a bit error occurs.

Table888's CR0 bits 6, 7 and 8 enable triple mode redundancy for reads, writes, and instruction fetches (execute) respectively.

Not that triple mode redundancy should be turned off while performing I/O operations. Some I/O devices reset internal values automatically on the first read of a register. These devices would not return valid data if triple mode redundancy were on. Additionally, performing three writes to an I/O device likely wouldn't have the desired effect. For instance, one wants to transmit a single character using a UART device, not a character repeated three times which is what happens with triple redundant stores.

Triple mode redundancy on the register file is also available as a build option.

PERFORMANCE MEASUREMENT / COUNTERS

In some processors there are performance measurement registers present. These are particularly present in machines designed for research purposes. A common register is the tick count register which simply increments every clock cycle. The tick count register may be settable, or it may simply count beginning at zero after a reset.

There may be other performance registers available such as a breakdown of counts for various types of instructions. For example, the number of instruction fetches, the number of load or store operations.

It's also common in embedded systems to have built in counters. The counters are used with comparators to provide periodic interrupt capabilities. A periodic interrupt source is a commonly present hardware component, often integrated with the processor. Many simple software task switchers use a periodic interrupt in their operation.

Table888 uses an external interrupt to provide periodic interrupts. The Qupls MPU has a interval timer component to it, which connects to the interrupt controller.

TICK COUNT

The Qupls tick count register begins counting from zero after a reset and is not alterable. This register may be used to measure things like the number of clock cycles per instruction. It may also be used as a micro-delay timer. It is not recommended to use the tick count for precise timing measurement. The frequency of the tick count varies with processor frequency. A wall-clock time register is better suited to many forms of measurement.

POWER MANAGEMENT

Related to performance is power management. Decreasing the amount of power consumed often mean decreasing the performance. One means to decrease power consumption is to limit the clock frequency. Power consumed is proportional to frequency, so lowering the operating frequency lowers the power consumption. Another means to reduce power consumption is to gate off functional units that are not in use. Table888 or Qupls doesn't provide this capability.

Some cpu's provide instructions that allow the clock to be stopped until an external event occurs. This often called a stop (STP) or halt (HLT) instruction.

FLOATING POINT

The author must confess his knowledge of floating point is somewhat limited, but constantly growing. The author's experience is limited mainly to using double precision numbers with banking applications. One can get by without hardware floating point. Most early micro-processors did not include floating point support. Floating point support is often implemented in software. External floating-point units were later offered as an optional co-processor. Finally floating-point operations were incorporated directly into the cpu chip. Floating-point support has become more common as transistor budgets have increased. Floating point almost exclusively follows the IEEE standard. Very few floating-point units are non-standard these days.

PRECISION

One of the issues with floating point is the available precision. There are a number of standard precisions possible. Sometimes software is used in lieu of greater precision than that available with hardware. Higher precision numbers may be built up out of lower precision representations by using multiple values but it gets to be complex. Try having a look at the double-double precision library. There is movement lately towards support for quadruple precision numbers in hardware. It seems that people like their precision, especially in financial, scientific and engineering applications. Achieving higher precision is often slower and lower performance than lower precision. For some applications where performance is critical, and precision is not required low precision floating point may be in use. The precision required is application dependent. Table888 only supports double precision operations. The Qupls ISA has support for four precisions, half, single, double, and quad precision. Precision can also be applied to integer operations.

OPERATIONS

Not too long ago, floating-point coprocessors typically supported a wide range of operations including trigonometric and exponential / logarithmic functions all in hardware. A more recent trend is to provide only basic operations in hardware.

Qupls includes floating point operations as part of the instruction set. Only basic operations such as FADD, FSUB, FMUL, FDIV and multiply-add are supported. Fortunately, the floating-point operations don't take many cycles to complete (they are faster than an integer divide for instance).

The floating-point operations unit itself is a module separate from the processor, but incorporated within it. The unit is capable of much higher performance than achieved in the processor implementation. Most operations can be pipelined and a new operation can start every clock cycle (excepting divide). However, this feature is not used when implemented in Qupls.

FLOATING POINT NUMBER FORMAT

The floating-point number format used by Qupls is the IEEE standard double precision format:

1	1	10	52 + 1 hidden bit
S_m	S_e	EEEEEEEEEE	.MMMMMMM…..MMMMMMMM

S_m = sign of mantissa

S_e = sign of exponent

The exponent and mantissa are both represented as two's complement numbers, however the sign bit of the exponent is inverted.

S_eEEEEEEEEEE	
11111111111	Maximum exponent
….	
01111111111	exponent of zero
….	
00000000000	Minimum exponent

The exponent ranges from -1024 to +1023

Half, single, and quad precision are also supported.

Qupls also supports quad precision *decimal* floating-point.

FLOATING POINT REGISTERS

Many architectures keep separate register files for floating-point and general-purpose registers. This helps improve floating-point performance which often relies on a lot of registers. It also compartmentalizes the floating-point making it possible to configure without floating-point support.

For Qupls floating-point values may be stored in the general-purpose register array.

PIPELINE DESIGN

Qupls is a four-way superscalar out-of-order design. The pipeline is very complex and deep with about nine stages to it.

Table888 is a non-overlapped pipelined design. A pipelined design implements the processor with a number of pipeline stages that data and instructions pass through. Most processors are pipelined in one fashion or another. 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. The Tabl888 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.

A superscalar pipeline design has parallel pipeline lanes associated with it.

PROCESSOR STAGES / STATES

This section gives a general overview of what is done during each pipeline stage. The description of these stages is not 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. There are usually registers that need to be reset before the processor can begin operations. For instance, in Qupls the TLB registers must be preset to allow access to the ROM boot memory.

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). This stage may have a branch-target-buffer predictor associated with it.

IALIGN

Instruction align – in a CPU instructions are often not where they need to be for subsequent processing. This stage typically shifts the instruction into a better position. Hardware decoders may be present only at specific positions relative to the beginning of an instruction.

EXTRACT / PARSE

This stage extracts the instruction(s) from the output of the align stage.

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.

Decoding instructions is done with a big case statement. All instructions are processed by the instruction decoder. Some of the simpler instructions may also be executed at this stage depending on the pipeline. 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.

REGISTER FILE ACCESS

During the decode stage, register file access may begin as well.

In theTable888 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. For Qupls all the register ports are always located in the same position of the instruction. This allows the incoming instruction to feed the register port number directly to the register file 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;
```

RENAME

Registers are renamed at this stage to remove false dependencies. The rename stage is present only in higher performance designs.

ENQUEUE

This stage places a decoded and renamed instruction into a queue for further processing.

SCHEDULE

Instructions are scheduled for execution. This stage may not be present depending on the pipelining. For a simpler pipeline the instructions simply execute in sequence, there is no reason to schedule them. For a more complex machine where instructions are in a queue the scheduler will attempt to pick the best instruction to execute that has ready arguments. The scheduler in a superscalar design can pick multiple instructions to execute at the same time.

EXECUTE

At this stage instructions are “executed”. Results are calculated based on the decoding of the previous stage.

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.

Qupls has two ALUs.

MEMORY STAGE

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

WRITEBACK

At this stage results are written to the register file.

COMMIT

At this stage the result of instruction execution is committed to the machine's architectural state.

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 its 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. Perhaps a bit-serial resource scrounging processor could be developed.

If one wants instructions to fetch from an instruction cache that must 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.

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.

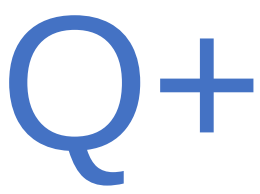
Checking for unaligned memory access. Currently the processor does not validate that the address for data is properly aligned. It'll go ahead and try to load data from unaligned addresses if they are specified that way; but it won't work properly.

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.

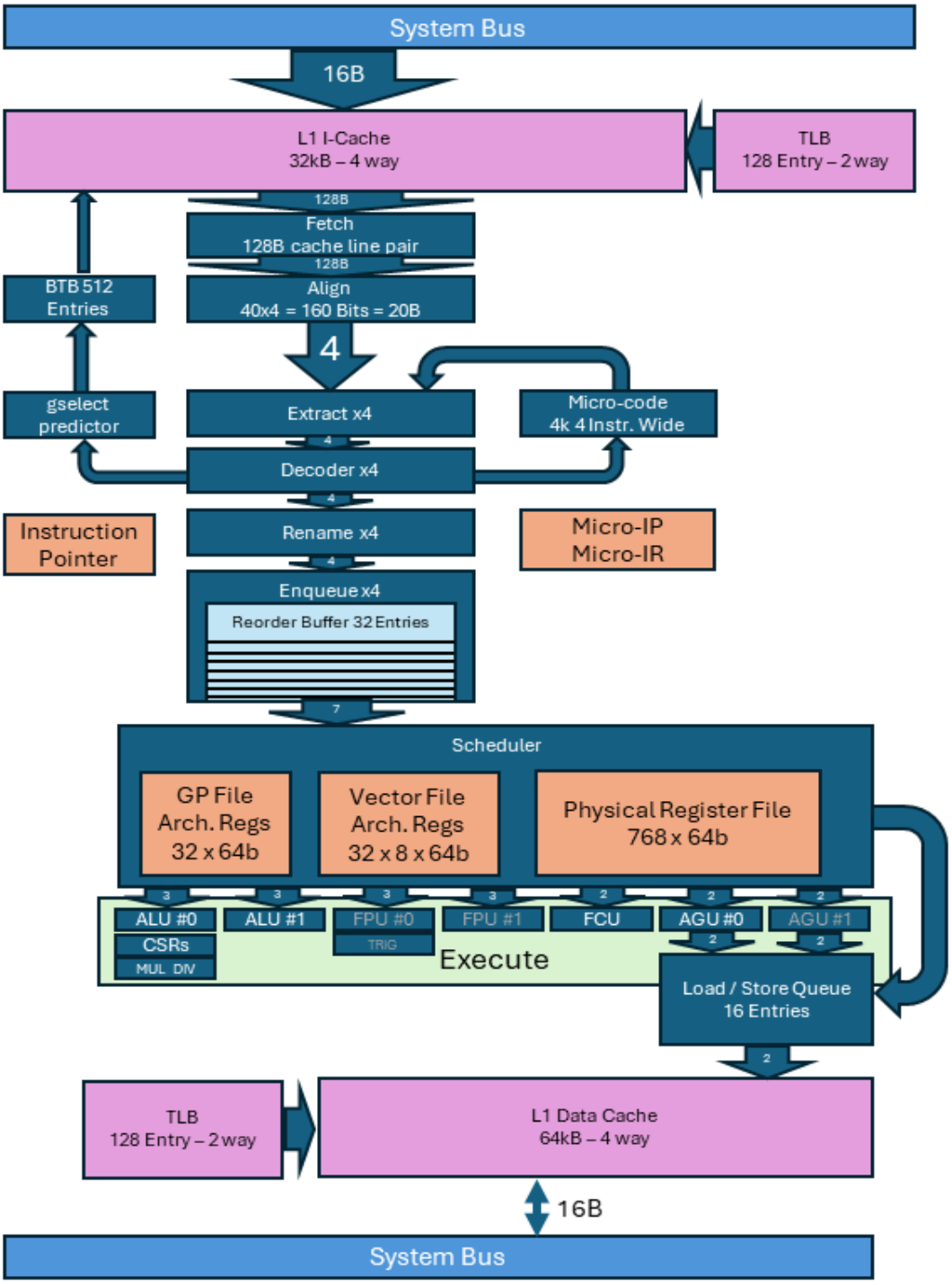
Additional arithmetic operations such as square root, minimum and maximum functions.

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

Protection mechanism.



Q+ Block Diagram



PROGRAMMING MODEL

REGISTER FILE

RN – GENERAL PURPOSE REGISTERS

The register file contains 32 64-bit general purpose registers.

The register file is *unified* and may hold either integer or floating-point values. The stack pointer is register 31. Register r31 is special in that it is banked depending on the operating mode or interrupt level of the CPU.

Register r0 is special in that it always reads as a zero.

The general-purpose registers are also aliases of vector registers zero to seven.

REGISTER ABI

Regno	ABI	ABI Usage
0	0	Always zero
1	A0	First argument / return value register
2	A1	Second argument / return value register
3	A2	Third argument register
4 to 8	A3 to A7	Argument registers
9 to 17	T0 to T8	Temporary register, caller save
18 to 26	S0 to S8	Saved register, register variables
27	LR0	Link register #0
28	LR1	Link register #1 (millicode)
29	GP	Global Pointer – data
30	FP	Frame Pointer
31	SP	App Stack pointer
31	SSP	Supervisor Stack pointer
31	HSP	Hypervisor Stack pointer
31	MSP	Machine Stack pointer (interrupt stack pointers too)

PN - PREDICATE REGISTERS

Predicate registers are part of the general-purpose register file and may be manipulated using the same instructions as for other registers. Any register of the general-purpose register array may be dedicated to predicate storage. Each byte of a predicate value corresponds to a vector element. Each bit of the byte is a predicate for a vector lane. If there are four lanes in the vector element then four bits of the predicate byte are used for masking and the other four bits are ignored.

The PRED instruction modifier may be used to apply a predicate to a group of instructions.

Predicate registers are used to mask off vector operations so that a vector instruction doesn't perform the operation on all elements of the vector. They are also used as Boolean predicate values for scalar operations.

CODE ADDRESS REGISTERS

Many architectures have registers dedicated to addressing code. Almost every modern architecture has a program counter or instruction pointer register to identify the location of instructions. Many architectures also have at least one link register or return address register holding the address of the next instruction after a subroutine call. There are also dedicated branch address registers in some architectures. These are all code addressing registers.

The original Thor lumped these registers together in a code address register array.

It is possible to do an indirect method call using any register.

LRN – LINK REGISTERS

There are two registers in the Qupls ABI reserved for subroutine linkage. These registers are used to store the address after the calling instruction. They may be used to implement fast returns for two levels of subroutines or to used to call milli-code routines. The jump to subroutine, JSR, and branch to subroutine, BSR, instructions update a link register. The return from subroutine,. RTS, instruction is used to return to the next instruction.

IP – INSTRUCTION POINTER

This register points to the currently executing instruction. The instruction pointer increments as instructions are fetched, unless overridden by another flow control instruction. The instruction pointer may be set to any byte address. There is no alignment restriction. It is possible to write position independent code, PIC, using IP relative addressing.

SR - STATUS REGISTER (CSR 0X?004)

The processor status register holds bits controlling the overall operation of the processor, state that needs to be saved and restored across interrupts. The bits have individual bit set / clear capability using the CSRRS, CSRRC instructions. Only the user interrupt enable bit is available in user mode, other bits will read as zero.

Bit		Usage
0	uie	User interrupt enable
1	sie	Supervisor interrupt enable
2	hie	Hypervisor interrupt enable
3	mie	Machine interrupt enable
4	die	Debug interrupt enable
5 to 7	ipl	Interrupt level
8	ssm	Single step mode
9	te	Trace enable
10 to 11	om	Operating mode
12 to 13	ps	Pointer size
14 to 15	~	reserved
16	mprv	memory privilege
17 to 19	~	reserved
20 to 23	~	reserved
24 to 31	cpl	Current privilege level

CPL is the current privilege level the processor is operating at.

T indicates that trace mode is active.

OM processor operating mode.

PS: indicates the size of pointers in use. This may be one of 32, 64 or 128 bits.

AR: Address Range indicates the number of address bits in use. 0 = near or short (32-bit) addressing is in use. When short addressing is in use only the low order 32-bit are significant and stored or loaded to or from the stack.

IPL is the interrupt mask level

MPRV Memory Privilege, indicates to use previous operating mode for memory privileges

VECTOR PROGRAMMING MODEL

REGISTER FILE

VN – VECTOR REGISTERS

The SIMD register file contains 32 512-bit registers. The SIMD registers are organized as 8 x 64-bit registers. A 64-bit register of the vector register is referred to as an element. Within each element may be multiple lanes of execution.

Regno	ABI	ABI Usage
0 to 6		These are the GPRs
7	VA0	First argument / return value
8	VA1	Second argument / return value
9 to 11	VA3 to VA5	
13 to 20	VT0 to VT7	
21 to 30	VS0 to VS9	

VECTOR RELATED REGISTERS / CSRS

Mnem.	Regno	Description
VGM		Global mask register
VRM	54	Restart mask register
VEX	55	Exception register

The number of lanes is limited to 64 as that is the width of a predicate register.

VECTOR GLOBAL MASK REGISTER (VGM)

The global mask register contains predicate bits indicating which vector elements are active. Vector elements of the target are updated only when the corresponding global mask bit is set. The global mask register takes the place of the vector length register in other architectures. Normally the global mask contains a right aligned bitmask of all ones up to the number of elements to be processed.

VECTOR RESTART MASK REGISTER (VRM) – REG #54

The restart mask register is a read-only register that contains a bitmask indicating the vectors elements to be processed after a restart. The restart mask register is set to all ones before vector operation begins. To restart a vector operation the predicate for the vector should be set to the bitwise ‘and’ of the global mask register and the restart mask register.

VECTOR EXCEPTION REGISTER (VEX) – REG #55

The vector exception register is a read-only register that contains the exception number for each vector element indicating if an exception occurred. The exception register is set to all zeros before a vector operation begins.

SPECIAL PURPOSE REGISTERS

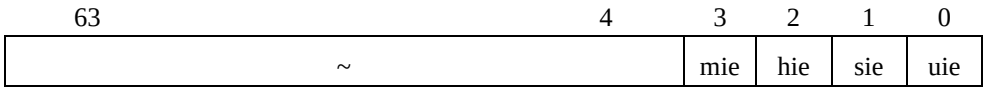
SC - STACK CANARY (GPR 53)

This special purpose register is available in the general register file as register 53. The stack canary register is used to alleviate issues resulting from buffer overflows on the stack. The canary register contains a random value which remains consistent throughout the run-time of a program. In the right conditions, the canary register is written to the stack during the function's prolog code. In the function's epilog code, the value of the canary on stack is checked to ensure it is correct, if not a check exception occurs.

[U/S/H/M]_IE (0X?004)

See status register.

This register contains interrupt enable bits. The register is present at all operating levels. Only enable bits at the current operating level or lower are visible and may be set or cleared. Other bits will read as zero and ignore writes. Only the lower four bits of this register are implemented. The bits have individual bit set / clear capability using the CSRRS, CSRRC instructions.

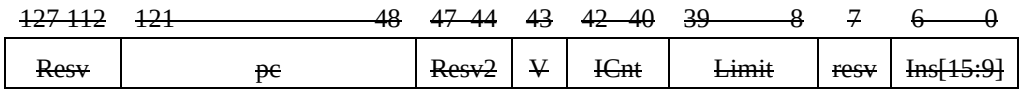


[U/S/H/M]_CAUSE (CSR- 0X?006)

This register contains a code indicating the cause of an exception or interrupt. The break handler will examine this code to determine what to do. Only the low order 8 bits are implemented. The high order bits read as zero and are not updateable.

U_REPBUF - (CSR – 0X008)

~~This register contains information needed for the REP instruction that must be saved and restored during context switches and interrupts. Note that the loop counter should also be saved.~~



~~Pc: (64 bits) the address of the instruction following the REP~~

~~V: REP valid bit, 1 only if a REP instruction is active~~

~~ICnt: the current instruction count, distance from REP instruction.~~

~~Limit: a 32-bit amount to compare the loop counter against.~~

~~Ins: bits 9 to 15 of the REP instruction which contains the instruction count of instruction included in the repeat and condition under which the repeat occurs.~~

[U/S/H/M]_SCRATCH – CSR 0X?041

This is a scratchpad register. Useful when processing exceptions. There is a separate scratch register for each operating mode.

~~S_PTBR (CSR 0X1003)~~

This register is now located in the page table walker device.

S_ASID (CSR 0X101F)

This register contains the address space identifier (ASID) or memory map index (MMI). The ASID is used in this design to select (index into) a memory map in the paging tables. Only the low order sixteen bits of the register are implemented.

S_KEYS (CSR 0X1020 TO 0X1027)

These eight registers contain the collection of keys associated with the process for the memory lot system. Each key is twenty-four bits in size. All eight registers are searched in parallel for keys matching the one associated with the memory page. Keyed memory enhances the security and reliability of the system.

			23 0
1020			key0
1021			key1
…			…
1027			key7

M_CORENO (CSR 0X3001)

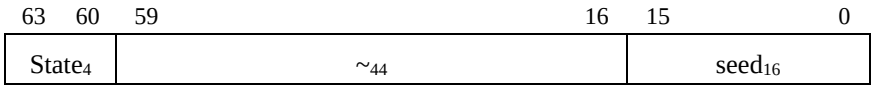
This register contains a number that is externally supplied on the coreno_i input bus to represent the hardware thread id or the core number. It should be non-zero.

M_TICK (CSR 0X3002)

This register contains a tick count of the number of clock cycles that have passed since the last reset. Note that this register should not be used for precise timing as the processor's clock frequency may vary for performance and power reasons. The TIME CSR may be used for wall-clock timing as it has its own timing source.

M_SEED (CSR 0X3003)

This register contains a random seed value based on an external entropy collector. The most significant bit of the state is a busy bit.



State$_4$ Bit	
0	dead
1	test
2	valid, the seed value is valid
3	Busy, the collector is busy collecting a new seed value

M_BADADDR (CSR 0X3007)

This register contains the address for a load / store operation that caused a memory management exception or a bus error. Note that the address of the instruction causing the exception is available in the EIP register.

M_BAD_INSTR (CSR 0X300B)

This register contains a copy of the exceptioned instruction.

M_SEMA (CSR 0X300C)

This register contains semaphores. The semaphores are shared between all cores in the MPU.

M_TVEC – CSR 0X3030 TO 0X3034

These registers contain the address of the exception handler table for a given operating mode. TVEC[0] to TVEC[2] are used by the REX instruction.

A sync instruction should be used after modifying one of these registers to ensure the update is valid before continuing program execution.

Reg #	
0x3030	TVEC[0] – user mode
0x3031	TVEC[1] - supervisor mode
0x3032	TVEC[2] – hypervisor mode
0x3033	TVEC[3] – machine mode
0x3034	TVEC[4] - debug

M_SR_STACK (CSR 0X3080 TO CSR 0X3087)

This set of registers contains a stack of the status register which is pushed during exception processing and popped on return from interrupt. There are only eight slots as that is the maximum nesting depth for interrupts.

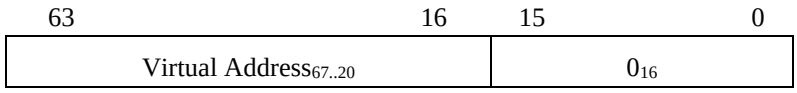
M_MC_STACK (CSR 0X3090 TO CSR 0X3097)

This set of registers is a stack for the micro-code instruction register (MCIR) and the micro-code instruction pointer (MCIP). MCIR and MCIP need to be retained through exception processing.

Bits 52 to 63 of the register contain the MCIP. Bits 0 to 51 contain the MCIR.

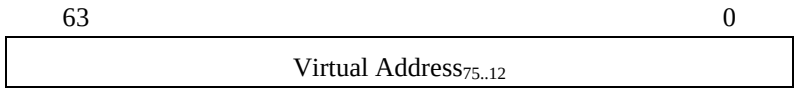
M_IOS – IO SELECT REGISTER (CSR 0X3100)

The location of IO is determined by the contents of the IOS control register. The select is for a 1MB region. This address is a virtual address. The low order 16 bits of this register should be zero and are ignored.



M_CFGS – CONFIGURATION SPACE REGISTER (CSR 0X3101)

The location of configuration space is determined by the contents of the CFGS control register. The select is for a 256MB region. This address is a virtual address. The low order 12 bits of this address are assumed to be zero. The default value of this registers is $FF…FD0000



M_EIP (CSR 0X3108 TO 0X310F)

This set of registers contains the address stack for the program counter used in exception handling.

Reg #	Name
0x3108	EIP0

…	
0x310F	EIP7

OPERATING MODES

The core operates in one of four basic modes: application/user mode, supervisor mode, hypervisor mode or machine mode. Machine mode is switched to when an interrupt or exception occurs, or when debugging is triggered. On power-up the core is running in machine mode. An RTI instruction must be executed to leave machine mode after power-up.

A subset of instructions is limited to machine mode.

Mode Bits	Mode
0	User / App
1	Supervisor
2	Hypervisor
3	Machine

Each operating mode has its own vector table.

EXCEPTIONS

EXTERNAL INTERRUPTS

There is little difference between an externally generated exception and an internally generated one. An externally caused exception will set the exception cause code for the currently fetched instruction. A hardware interrupt displaces the instruction at the point the interrupt occurred with a TRAP.

There are eight priority interrupt levels for external interrupts. When an external interrupt occurs the mask level is set to the level of the current interrupt. A subsequent interrupt must exceed the mask level to be recognized.

EFFECT ON MACHINE STATUS

The operating mode is always switched to machine mode on exception. It is up to the machine mode code to redirect the exception to a lower operating mode when desired. Further exceptions at the same or lower interrupt level are disabled automatically. Machine mode code must enable interrupts at some point.

EXCEPTION STACK

The status register and program counter are pushed onto an internal stack when an exception occurs. This stack is at least 8 entries deep to allow for nested interrupts and multiply nested traps and exceptions. The stack pointer is also switched to one corresponding to the machine's operating mode. A hardware interrupt will also cause the stack pointer to change to one specific to the interrupt level.

EXCEPTION TABLE

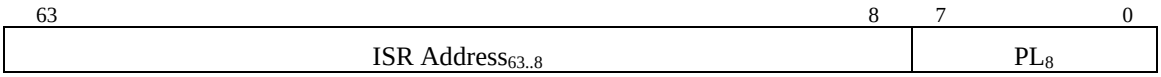
There is a separate kernel vector for each operating mode. The machine mode kernel vector is always used to locate the exception routine. The exception routine may then redirect the exception to a lower operating mode using the REX instruction.

Vector	Usage
0	Debug Breakpoint (BRK)
1	
2	Bus Error
3	Address Error
4	Unimplemented Instruction
5	
6	Page fault
7	
8	Privilege Violation
9	Instruction trace
10	
11	Stack Canary
12 to 23	Reserved
24	Spurious interrupt

25	Auto vector #1
26	Auto vector #2
27	Auto vector #3
28	Auto vector #4
29	Auto vector #5
30	Auto vector #6
31	Auto vector #7
32	Debug breakpoint – single step
33	
34	Instruction Address
33 to 63	Trap #1 to 31
	Applications Usage
64	Divide by zero
65	Overflow
66	Table Limit
67 to 251	Unassigned usage
252	Reset value of stack pointer
253	Reset value of instruction pointer
254, 255	Reserved

VECTOR FORMAT

Interrupt subroutine addresses are always 256-byte aligned. The ISR vector includes a 64-bit virtual address of the ISR and an eight-bit privilege level. The privilege level is checked when the ISR is invoked, and the privilege level of the vector must be at least equal to the privilege level of the processor, or a protection fault will occur.



RESET STACK POINTER VECTOR (0)

This vector contains the address the machine stack pointer is set to at reset.

RESET VECTOR VECTOR (1)

This vector contains the address that the processor begins running at.

BUS ERROR FAULT (2)

The bus error fault is performed if the bus error signal was active during the bus transaction. This could be due to a bad or missing device.

UNIMPLEMENTED INSTRUCTION FAULT (4)

An unimplemented instruction causes this fault.

STACK CANARY FAULT (11)

This fault is caused if the stack canary was overwritten. A load instruction using the canary register did not match the value in the canary register.

BREAKPOINT FAULT (33)

The breakpoint instruction, 0, was encountered.

INSTRUCTION ADDRESS FAULT (34)

An error occurred addressing instructions. This could be due to a bad instruction sequence, for instance executing multiple postfixes in a row.

RESET

Reset is treated as an exception. The reset routine should exit using an RTE instruction. The status register should be setup appropriately for the return.

The core begins executing instructions at the address defined by the reset vector in the exception table. At reset the exception table is set to the last 512 words of memory $FF…FF000. All registers are in an undefined state.

PRECISION

Exceptions in Qupls are precise. They are processed according to program order of the instructions. If an exception occurs during the execution of an instruction, then an exception field is set in the pipeline buffer. The exception is processed when the instruction commits which happens in program order. If the instruction was executed in a speculative fashion, then no exception processing will be invoked unless the instruction makes it to the commit stage.

HARDWARE DESCRIPTION

CACHES

OVERVIEW

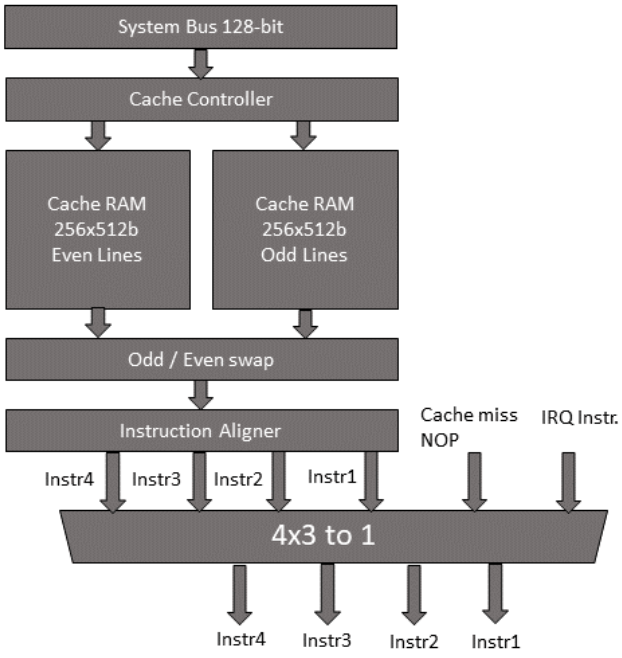
The core has both instruction and data caches to improve performance. Both caches are single level. The cache is four-way associative. The cache sizes of the instruction and data cache are available for reference from one of the info lines return by the CPUID instruction.

INSTRUCTIONS

Since the instruction format affects the cache design it is mentioned here. For this design instructions are of a fixed 40-bit parcels format. Specific formats are listed under the instruction set description section of this book. A 40-bit parcel was chosen because it's simpler for a hobbyist design and to limit the amount of multiplexing taking place for an instruction read. The author found that determining the length of an instruction and selecting the next instruction to fetch based on a varying length affected the timing of the core. A simpler design makes it easier to achieve a higher clock rate.

L1 INSTRUCTION CACHE

L1 is 32kB in size and made from block RAM with a single cycle of latency. L1 is organized as an odd, even pair of 256 lines of 64 bytes. The following illustration shows the L1 cache organization for Qupls.



The cache is organized into odd and even lines to allow instructions to span a cache line. Two cache lines are fetched for every access; the one the instruction is located on, and the next one in case the instruction spans a line.

A 256-line cache was chosen as that matches the inherent size of block RAM component in the FPGA. It is the author's opinion that it would be better if the L1 cache were larger because it often misses due to its small size. In short the current design is an attempt to make it easy for the tools to create a fast implementation.

Note that supporting interrupts and cache misses, a requirement for a realistic processor design, adds complexity to the instruction stream. Reading the cache ram, selecting the correct instruction word and accounting for interrupts and cache misses must all be done in a single clock cycle.

While the L1 cache has single cycle reads it requires two clock cycles to update (write) the cache. The cache line to update needs to be provided by the tag memory which is unknown until after the tag updates.

DATA CACHE

The data cache organization is somewhat simpler than that of the instruction cache. Data is cached with a single level cache because it's not critical that the data be available within a single clock cycle at least not for the hobby design. Some of the latency of the data cache can be hidden by the presence of non-memory operating instructions in the instruction queue.

The data cache is organized as 512 lines of 64 bytes (32kB) and implemented with block ram. Access to the data cache is multicycle. The data cache may be replicated to allow more memory instructions to be processed at the same time; however, just a single cache is in use for the demo system. The policy for stores is write-through. Stores always write through to memory. Since stores follow a write-through policy the latency of the store operation depends on the external memory system. It isn't critical that the cache be able to update in single cycle as external memory access is bound to take many more cycles than a cache update. There is only a single write port on the data cache.

CACHE ENABLES

The instruction cache is always enabled to keep hardware simpler and faster. Otherwise, an additional multiplexor and control logic would be required in the instruction stream to read from external memory.

For some operations, it may desirable to disable the data cache so there is a data cache enable bit in control register #0. This bit may be set or cleared with one of the CSR instructions.

CACHE VALIDATION

A cache line is automatically marked as valid when loaded. The entire cache may be invalidated using the CACHE instruction. Invalidating a single line of the cache is not currently supported, but it is supported by the ISA. The cache may also be invalidated due to a write by another core via a snoop bus.

UN-CACHED DATA AREA

The address range $F…FDxxxxx is an un-cached 1MB data area. This area is reserved for I/O devices. The data cache may also be disabled in control register zero. There is also field in the load instructions that allows bypassing the data cache.

FETCH RATE

The fetch rate is four instructions per clock cycle.

RETURN ADDRESS STACK PREDICTOR (RSB)

There is an address predictor for return addresses which can in some cases can eliminate the flushing of the instruction queue when a return instruction is executed. The RETD instruction is detected in the fetch stage of the core and a predicted return address used to fetch instructions following the return. The return address stack predictor has a stack depth of 64 entries. On stack overflow or underflow, the prediction will be wrong, however performance will be no worse than not having a predictor. The return address stack predictor checks the address of the instruction queued following the RET against the address fetched for the RET instruction to make sure that the address corresponds.

There is a separate RSB for each thread while operating with SMT turned on.

BRANCH PREDICTOR

The branch predictor is a (2, 2) correlating predictor. The branch history is maintained in a 512- entry history table. It has four read ports for predicting branch outcomes, one port for each instruction fetched. The branch predictor may be disabled by a bit in control register zero. When disabled all branches are predicted as not taken, unless specified otherwise in the branch instruction.

To conserve hardware the branch predictor uses a fifo that can queue up to four branch outcomes at the same time. Outcomes are removed from the fifo one at a time and used to update the branch history table which has only a single write port. In an earlier implementation of the branch predictor, two write ports were provided on the history table. This turned out to be relatively large compared to its usefulness.

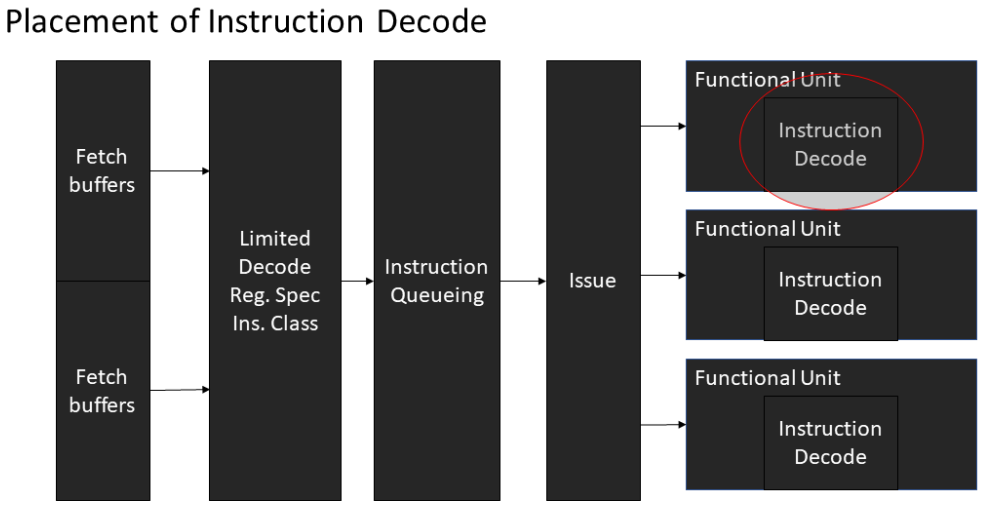
Correctly predicting a branch turns the branch into a single cycle operation. During execution of the branch instruction the address of the following instruction queued is checked against the address depending on the branch outcome. If the address does not match what is expected, then the queue will be flushed, and new instructions loaded from the correct program path.

BRANCH TARGET BUFFER (BTB)

The core has a 1k entry branch target buffer for predicting the target address of flow control instructions where the address is calculated and potentially unknown at time of fetch. Instructions covered by the BTB include jump-and-link, interrupt return and breakpoint instructions and branches to targets contained in a register.

DECODE LOGIC

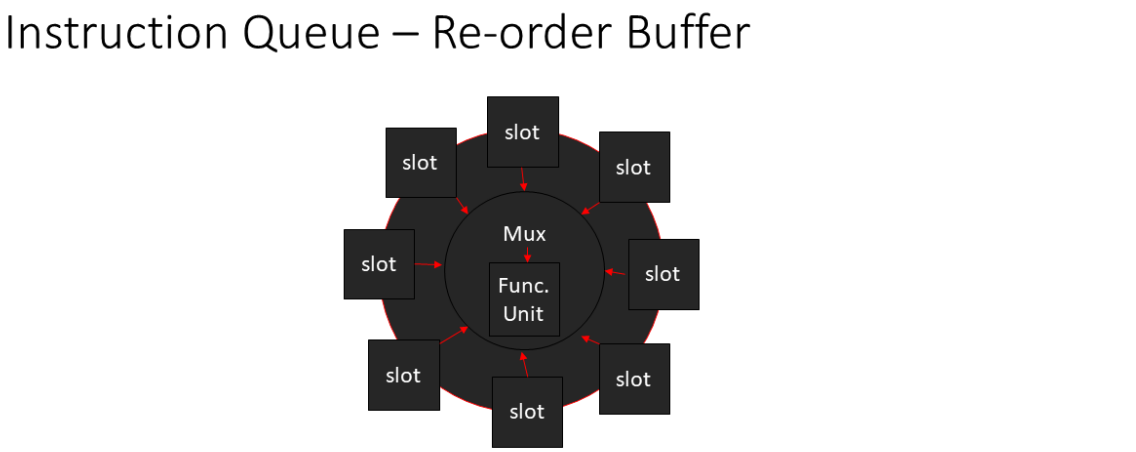
Instruction decode is distributed about the core. Although a number of decodes take place between fetch and instruction queue. Broad classes of instructions are decoded for the benefit of issue logic along with register specifications prior to instruction enqueue. Most of the decodes are done with modules under the decoder folder. Decoding typically involves reducing a wide input into a smaller number of output signals. Other decodes are done at instruction execution time with case statements.



Limited decode takes place between fetch and queue. Between fetch and queue register specifications are decoded along with general instruction classes for the benefit of issue. A handful of additional signals (like sync) that control the overall operation of the core are also decoded. Much of the instruction decode is actually done in the functional unit. The instruction register is passed right through to the functional units in the core.

INSTRUCTION QUEUE (ROB)

The instruction queue is a 32-entry re-ordering buffer (ROB). The instruction queue tracks an instructions progress. Each instruction in queue may be in one of several different states. The instruction queue is a circular buffer with head and tail pointers. Instructions are queued onto the tail and committed to the machine state at the head. Queue and commit takes place in groups of up to four instructions.



The instruction queue is circular with eight slots. Each slot feeds a multiplexor which in turn feeds a functional unit. Providing arguments to the functional unit is done under the vise of issue logic. Output from the functional unit is fed back to the same queue slot that issued to the functional unit.
The queue slots are fed from the fetch buffers.

QUEUE RATE

Up to four instructions may queue during the same clock cycle depending on the availability of queue slots.

SEQUENCE NUMBERS

The queue maintains a 7-bit instruction sequence number which gives other operations in the core a clue as to the order of instructions. The sequence number is assigned when an instruction queues. Branch instructions need to know when the next instruction has queued to detect branch misses. The program counter cannot be used to determine the instruction sequence because there may be a software loop at work which causes the program counter to cycle backwards even though it's really the next instruction executing.

INPUT / OUTPUT MANAGEMENT

Before getting into memory management a word or two about I/O management is in order. Memory management depends on several I/O devices. I/O in Qupls is memory mapped or MMIO. Ordinary load and store instructions are used to access I/O registers. I/O is mapped as a non-cacheable memory area.

DEVICE CONFIGURATION BLOCKS

I/O devices have a configuration block associated with them that allows the device to be discovered by the OS during bootup. All the device configuration blocks are located in the same 256MB region of memory in the address range $FF….D0000000 to $FF…DFFFFFFF. Each device configuration block is aligned on a 4kB boundary. There is thus a maximum of 64k device configuration blocks.

RESET

At reset the device configuration blocks are not accessible. They must be mapped into memory for access. However, the devices have default addresses assigned to them, so it may not be necessary to map the device control block into memory before accessing the device. The device itself also needs to be mapped into the memory space for access though.

DEVICES BUILT INTO THE CPU / MPU

Devices present in the CPU itself include:

Device	Bus	Device	Func	IRQ	Config Block Address	Default Address
Interrupt Controller	0	6	0	~	$FF..FD0030000	$FF..FEE2xxxx
Interval Timers	0	4	0	29	$FF..FD0020000	$FF..FEE4xxxx
Memory Region Table	0	12	0	~	$FF..FD0060000	$FF..FEEFxxxx
Page Table Walker	0	14	0	~	$FF..FD0070000	$FF..FFF4xxxx
Hardware Card Table	0		0	~		

Function is mapped to address bits 12 to 14

Device is mapped to address bits 15 to 19

Bus is mapped to address bits 20 to 27

MEMORY MANAGEMENT

BANK SWAPPING

About the simplest form of memory management is a single bank register that selects the active memory bank. This is the mechanism used on many early microcomputers. The bank register may be an eight bit I/O port supplying control over some number of upper address bits used to access memory.

THE PAGE MAP

The next simplest form of memory management is a single table map of virtual to physical addresses. The page map is often located in a high-speed dedicated memory. An example of a mapping table is the 74LS612 chip. It may map four address bits on the input side to twelve address bits on the output side. This allows a physical address range eight bits greater than the virtual address range. A more complicated page map is something like the MC6829 MMU. It may map 2kB pages in a 2MB physical address space for up to four different tasks.

REGIONS

In any processing system there are typically several different types of storage assigned to different physical address ranges. These include memory mapped I/O, MMIO, DRAM, ROM, configuration space, and possibly others. Qupls has a region table that supports up to eight separate regions.

The region table is a list of region entries. Each entry has a start address, an end address, an access type field, and a pointer to the PMT, page management table. To determine legal access types, the physical address is searched for in the region table, and the corresponding access type returned. The search takes place in parallel for all eight regions.

Once the region is identified the access rights for a particular page within the region can be found from the PMT corresponding to the region. Global access rights for the entire region are also specified in the region table. These rights are gated with value from the PMT and TLB to determine the final access rights.

REGION TABLE LOCATION

The region table in Q+ is a memory mapped I/O device and has a device configuration block associated with it. The default address of the device is $FF…FEEF0000.

REGION TABLE DESCRIPTION

Reg	Bits	Field	Description
0000	128	Pmt	associated PMT address
0010	128	cta	Card table address
0020	128	at	Four groups of 32-bit memory attributes, 1 group for each of user, supervisor, hypervisor and machine.
0030	128	…	Not used
0040 to 01F0		…	7 more register sets

PMT ADDRESS

The PMT address specifies the location of the associated PMT.

CTA – CARD TABLE ADDRESS

The card table address is used during the execution of the store pointer, STPTR instruction to locate the card table.

ATTRIBUTES

Bitno		
0	X	may contain executable code
1	W	may be written to
2	R	may be read
3	~	reserved
4-7	C	Cache-ability bits
8-10	G	granularity G: 0 byte accessible; 1 wyde accessible; 2 tetra accessible; 3 octa accessible; 4 hexi accessible; 5 to 7 reserved
11	~	reserved
12-14	S	number of times to shift address to right and store for telescopic STPTR stores.
16-23	T	device type (rom, dram, eeprom, I/O, etc)
24-31	~	reserved

PMA - PHYSICAL MEMORY ATTRIBUTES CHECKER

OVERVIEW

The physical memory attributes checker is a hardware module that ensures that memory is being accessed correctly according to its physical attributes.

Physical memory attributes are stored in an eight-entry region table. Three bits in the PTE select an entry from this table. The operating mode of the CPU also determines which 32-bit set of attributes to apply for the memory region.

Most of the entries in the table are hard-coded and configured when the system is built. However, they may be modified.

Physical memory attributes checking is applied in all operating modes.

The region table is accessible as a memory mapped IO, MMIO, device.

PAGE MANAGEMENT TABLE - PMT

OVERVIEW

For the first translation of a virtual to physical address, after the physical page number is retrieved from the TLB, the region is determined, and the page management table is referenced to obtain the access rights to the page. PMT information is loaded into the TLB entry for the page translation. The PMT contains an assortment of information most of which is managed by software. Pieces of information include the key needed to access the page, the privilege level, and read-write-execute permissions for the page. The table is organized as rows of access rights table entries (PMTEs). There are as many PMTEs as there are pages of memory in the region.

For subsequent virtual to physical address translations PMT information is retrieved from the TLB.

As the page is accessed in the TLB, the TLB may update the PMT.

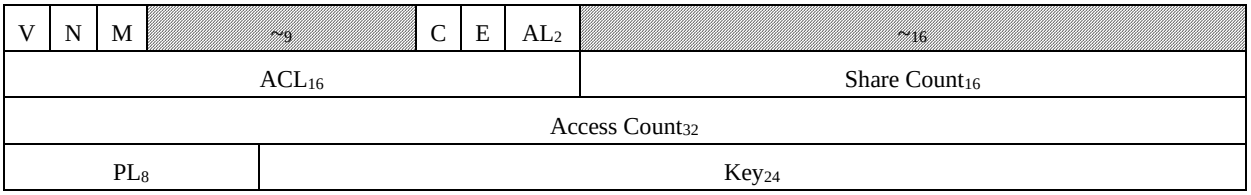
LOCATION

The page management table is in main memory and may be accessed with ordinary load and store instructions. The PMT address is specified by the region table.

PMTE DESCRIPTION

There is a wide assortment of information that goes in the page management table. To accommodate all the information an entry size of 128-bits was chosen.

Page Management Table Entry



ACCESS CONTROL LIST

The ACL field is a reference to an associated access control list.

SHARE COUNT

The share count is the number of times the page has been shared to processes. A share count of zero means the page is free.

ACCESS COUNT

This part uses the term ‘access count’ to refer to the number of times a page is accessed. This is usually called the reference count, but that phrase is confusing because reference counting may also refer to share counts. So, the phrase ‘reference count’ is avoided. Some texts use the term reference count to refer to the

share count. Reference counting is used in many places in software and refers to the number of times something is referenced.

Every time the page of memory is accessed, the access count of the page is incremented. Periodically the access count is aged by shifting it to the right one bit.

The access count may be used by software to help manage the presence of pages of memory.

KEY

The access key is a 24-bit value associated with the page and present in the key ring of processes. The keyset is maintained in the keys CSRs. The key size of 20 bits is a minimum size recommended for security purposes. To obtain access to the page it is necessary for the process to have a matching key OR if the key to match is set to zero in the PMTE then a key is not needed to access the page.

PRIVILEGE LEVEL

The current privilege level is compared with the privilege level of the page, and if access is not appropriate then a privilege violation occurs. For data access, the current privilege level must be at least equal to the privilege level of the page. If the page privilege level is zero anybody can access the page.

N

indicates a conforming page of executable code. Conforming pages may execute at the current privilege level. In which case the PL field is ignored.

M

indicates if the page was modified, written to, since the last time the M bit was cleared. Hardware sets this bit during a write cycle.

E

indicates if the page is encrypted.

AL

indicates the compression algorithm used.

C

The C indicator bit indicates if the page is compressed.

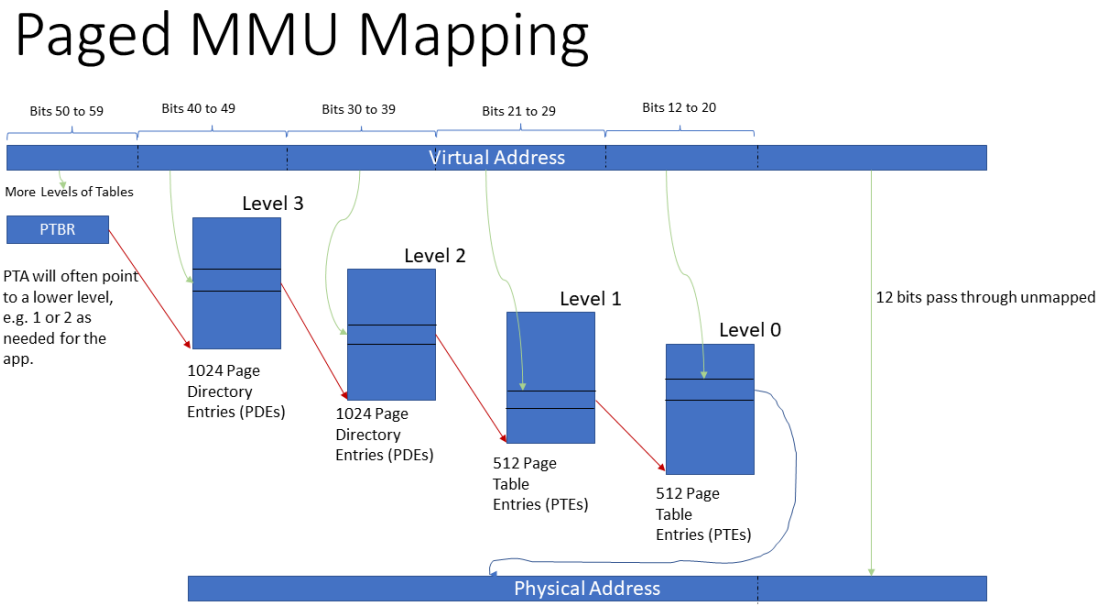
PAGE TABLES

INTRO

Page tables are part of the memory management system used map virtual addresses to real physical addresses. There are several types of page tables. Hierarchical page tables are probably the most common. Almost all page tables map only the upper bits of a virtual address, called a page. The lower bits of the virtual address are passed through without being altered. The page size often 4kB which means the low order 12-bits of a virtual address will be mapped to the same 12-bits for the physical address.

HIERARCHICAL PAGE TABLES

Hierarchical page tables organize page tables in a multi-level hierarchy. They can map the entire virtual address range but often only a subrange of the full virtual address space is mapped. This can be determined on an application basis. At the topmost level a register points to a page directory, that page directory points to a page directory at a lower level until finally a page directory points to a page containing page table entries. To map an entire 64-bit virtual address range approximately five levels of tables are required.

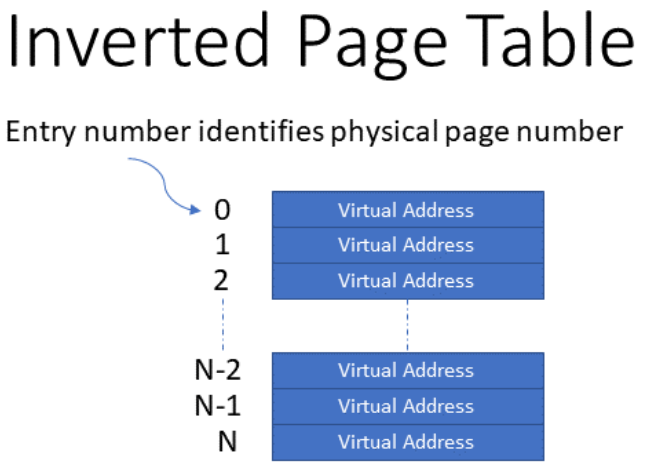


INVERTED PAGE TABLES

An inverted page table is a table used to store address translations for memory management. The idea behind an inverted page table is that there are a fixed number of pages of memory no matter how it is mapped. It should not be necessary to provide for a map of every possible address, which is what the hierarchical table does, only addresses that correspond to real pages of memory need be mapped. Each page of memory can be allocated only once. It is either allocated or it is not. Compared to a non-inverted paged memory management system where tables are used to map potentially the entire address space an inverted page table uses less memory. There is typically only a single inverted page table supporting all applications in the system. This is a different approach than a non-inverted page table which may provide separate page tables for each process.

THE SIMPLE INVERTED PAGE TABLE

The simplest inverted page table contains only a record of the virtual address mapped to the page, and the index into the table is used as the physical page number. There are only as many entries in the inverted page table as there are physical pages of memory. A translation can be made by scanning the table for a matching virtual address, then reading off the value of the table index. The attraction of an inverted page table is its small size compared to the typical hierarchical page table. Unfortunately, the simplest inverted page table is not practical when there are thousands or millions of pages of memory. It simply takes too long to scan the table. The alternative solution to scanning the table is to hash the virtual address to get a table index directly.



HASHED PAGE TABLES

HASHED TABLE ACCESS

Hashes are great for providing an index value immediately. The issue with hash functions is that they are just a hash. It is possible that two different virtual address will hash to the same value. What is then needed is a way to deal with these hash collisions. There are a couple of different methods of dealing with collisions. One is to use a chain of links. The chain has each link in the chain pointing the to next page table entry to use in the event of a collision. The hash page table is slightly more complicated then as it needs to store links for hash chains. The second method is to use open addressing. Open addressing calculates the next page table entry to use in the event of a collision. The calculation may be linear, quadratic or some other function dreamed up. A linear probe simply chooses the next page table entry in succession from the previous one if no match occurred. Quadratic probing calculates the next page table entry to use based on squaring the count of misses.

CLUSTERED HASH TABLES

A clustered hash table works in the same manner as a hashed page table except that the hash is used to access a cluster of entries rather than a single entry. Hashed values may map to the same cluster which can store multiple translations. Once the cluster is identified, all the entries are searched in parallel for the correct one. A clustered hash table may be faster than a simple hash table as it makes use of parallel searches. Often accessing memory returns a cache line regardless of whether a single byte or the whole cached line is referenced. By using a cache line to store a cluster of entries it can turn what might be multiple memory accesses into a single access. For example, an ordinary hash table with open addressing

may take up to 10 memory accesses to find the correct translation. With a clustered table that turns into 1.25 memory accesses on average.

SHARED MEMORY

Another memory management issue to deal with is shared memory. Sometimes applications share memory with other apps for communication purposes, and to conserve memory space where there are common elements. The same shared library may be used by many apps running in the system. With a hierarchical paged memory management system, it is easy to share memory, just modify the page table entry to point to the same physical memory as is used by another process. With an inverted page table having only a single entry for each physical page is not sufficient to support shared memory. There needs to be multiple page table entries available for some physical pages but not others because multiple virtual addresses might map to the same physical address. One solution would be to have multiple buckets to store virtual addresses in for each physical address. However, this would waste a lot of memory because much of the time only a single mapped address is needed. There must be a better solution. Rather than reading off the table index as the physical page number, the association of the virtual and physical address can be stored. Since we now need to record the physical address multiple times the simple mechanism of using the table index as the physical page number cannot be used. Instead, the physical page number needs to be stored in the table in addition to the virtual page number.

That means a table larger than the minimum is required. A minimally sized table would contain only one entry for each physical page of memory. So, to allow for shared memory the size of the table is doubled. This smells like a system configuration parameter.

SPECIFICS: QUPLS PAGE TABLES

QUPLS HASH PAGE TABLE SETUP

HASH PAGE TABLE ENTRIES - HPTE

We have determined that a page table entry needs to store both the physical page number and the virtual page number for the translations. To keep things simple, the page table stores only the information needed to perform an address translation. Other bits of information are stored in a secondary table called the page management table, PMT. The author did a significant amount of juggling around the sizes of various fields, mainly the size of the physical and virtual page numbers. Finally, the author decided on a 192-bit HPTE format.

V	LVL/BC_5	RGN_3	M	A	T	S	G	SW_2	$CACHE_4$	$MRWX_3$	$HRWX_3$	$SRWX_3$	$URWX_3$
$PPN_{31..0}$													
$PPN_{63..32}$													
$VPN_{37..6}$													
$VPN_{69..38}$													
$\sim_4$			$ASID_{11..0}$						$\sim_2$	$VPN_{83..70}$			

Fields Description

V	1	translation Valid
G	1	global translation
RGN	3	region
PPN	64	Physical page number
VPN	84	Virtual page number
RWX	3	readable, writeable, executable
ASID	12	address space identifier
LVL/BC	5	bounce count
M	1	modified
A	1	accessed
T	1	PTE type (not used)
S	1	Shared page indicator
SW	3	OS usage

The page table does not include everything needed to manage pages of memory. There is additional information such as share counts and privilege levels to take care of, but this information is better managed in a separate table.

SMALL HASH PAGE TABLE ENTRIES - SHPTE

The small HPTE is used for the test system which contains only 512MB of physical RAM to conserve hardware resources. The SHPTE is 72-bits in size. A 32-bit physical address is probably sufficient for this system. So, the physical page number could be 18-bits or less depending on the page size.

V	LVL/BC_5	RGN_3	M	A	T	S	G	SW	$CACHE_4$	$ASID_{3..0}$	$HRWX_3$	$SRWX_3$	$URWX_3$
$VPN_{15..0}$									$PPN_{15..0}$				
											$ASID_{7..4}$	$VPN_{19..16}$	

PAGE TABLE GROUPS – PTG

We want the search for translations to be fast. That means being able to search in parallel. So, PTEs are stored in groups that are searched in parallel for translations. This is sometimes referred to as a clustered table approach. Access to the group should be as fast as possible. There are also hardware limits to how many entries can be searched at once while retaining a high clock rate. So, the convenient size of 1024 bits was chosen as the amount of memory to fetch.

A page table group then contains five HPTE entries. All entries in the group are searched in parallel for a match. Note that the entries are searched as the PTG is loaded, so that the PTG group load may be aborted early if a matching PTE is found before the load is finished.

191 0

PTE0
PTE1
PTE2
PTE3
PTE4

Small Page Table Group

For the small page table, a fetch size of 576 bits was chosen. This allows eight SHPTEs to fit into one group.

SIZE OF PAGE TABLE

There are several conflicting elements to deal with, with regards to the size of the page table. Ideally, the hash page table is small enough to fit into the block RAM resources available in the FPGA. It may be practical to store the hash page table in block RAM as there would be only a single table for all apps in the system. This probably would not be practical for a hierarchical table.

About 1/6 of the block RAMs available are dedicated to MMU use. At the same time a multiple of the number of physical pages of memory should be supported to support page sharing and swapping pages to secondary storage. To support swapping pages, double the number of physical entries were chosen. To support page sharing, double that number again. Therefore, a minimum size of a page table would contain at least four times the number of physical pages for entries. By setting the size of the page table instead of the size of pages, it can be worked backwards how many pages of memory can be supported.

For a system using 256k block RAM to store PTEs. 256k / 8 = 32768 entries. 32,768 / 4 = 8,192 physical pages. Since the RAM size is 512MB, each page would be 512MB/8,192 = 64kB. Since half the pages may be in secondary storage, 1GB of address range is available.

Since there are 32,768 entries in the table and they are grouped into groups of eight, there are 4,096 PTGs. To get to a page table group fast a hash function is needed then that returns a 12-bit number.

Reworking things with a 64kB page size and 32,768 PTEs. The maximum memory size that can be supported is: 2.0 GB. This is only 4x the amount of RAM in the system, but may be okay for demo purposes.

HASH FUNCTION

The hash function needs to reduce the size of a virtual address down to a 10-bit number. The asid should be considered part of the virtual address. Including the asid of 10-bits and a 32-bit address is 42 bits. The first thing to do is to throw away the lowest eighteen bits as they pass through the MMU unaltered. We now have 24-bits to deal with. We can probably throw away some high order bits too, as a process is not likely to use the full 32-bit address range.

The hash function chosen uses the asid combined with virtual address bits 20 to 29. This should space out the PTEs according to the asid. Address bits 18 and 19 select one of four address ranges. the PTG supports seven PTEs. The translations where address bits 18 and 19 are involved are likely consecutive pages that would show up in the same PTG. The hash is the asid exclusively or'd with address bis 20 to 29.

COLLISION HANDLING

Quadratic probing of the page table is used when a collision occurs. The next PTG to search is calculated as the hash plus the square of the miss count. On the first miss the PTG at the hash plus one is searched. Next the PTG at the hash plus four is searched. After that the PTG at the hash plus nine is searched, and so on.

FINDING A MATCH

Once the PTG to be searched is located using the hash function, which PTE to use needs to be sorted out. The match operation must include both the virtual address bits and the asid, address space identifier, as part of the test for a match. It is possible that the same virtual address is used by two or more different address spaces, which is why it needs to be in the match.

LOCALITY OF REFERENCE

The page table group may be cached in the system read cache for performance. It is likely that the same PTG group will be used multiple times due to the locality of reference exhibited by running software.

ACCESS RIGHTS

To avoid duplication of data the access rights are stored in another table called the PMT for access rights table. The first time a translation is loaded the access rights are looked-up from the PMT. A bit is set in the TLB entry indicating that the access rights are valid. On subsequent translations the access rights are not looked up, but instead they are read from values cached in the TLB.

QUPLS HIERARCHICAL PAGE TABLE SETUP

PAGE TABLE ENTRIES - PTE

For hierarchical tables the structure is like that of hashed page tables except that there is no need to store the virtual address. We know the virtual address because it is what is being translated and there is no chance of collisions unlike the hash table. The structure is 96 bits in size. This allows 4096 PTEs to fit into an 64kB page. ¼ of the 64kB page is not used. Note the size of pages in the table is a configuration parameter used to build the system.

There are two types of page table entries. The first type, T=0, is a pointer to a page of memory, the second type, T=1, is an entry that points to lower-level page tables. PTE's that point to lower-level page tables are sometimes called page table pointers, PTPs.

PAGE TABLE ENTRY FORMAT – PTE

V	LVL/BC$_5$	RGN$_3$	M	A	T	S	G	SW$_2$	CACHE$_4$	MRWX$_3$	HRWX$_3$	SRWX$_3$	URWX$_3$
PPN$_{31..0}$													
PPN$_{63..32}$													

SMALL PAGE TABLE ENTRY FORMAT – SPTE

The small PTE format is used when the physical address space is less than 48-bits in size. The small PTE occupies only 64-bits. 8192 SPTEs will fit into an 64kB page.

V	LVL/BC$_5$	RGN$_3$	M	A	T	S	G	SW$_2$	CACHE$_4$	MRWX$_3$	HRWX$_3$	SRWX$_3$	URWX$_3$

$PPN_{31..0}$

Field	Size	Purpose
PPN	64	Physical page number
URWX	3	User read-write-execute override
SRWX	3	Supervisor read-write-execute override
HRWX	3	Hypervisor read-write-execute override
MRWX	3	Machine read-write-execute override
CACHE	4	Cache-ability bits
SW	2	OS software usage
A	1	1=accessed/used
M	1	1=modified
V	1	1 if entry is valid, otherwise 0
S	1	1=shared page
G	1	1=global, ignore ASID
T	1	0=page pointer, 1= table pointer
RGN	3	Region table index
LVL/BC	5	the page table level of the entry pointed to

SUPER PAGES

The hierarchical page table allows "super pages" to be defined. These pages bypass lower levels of page tables by using an entry at a high level to represent a block containing many pages.

Normally a PTE with LVL=0 is a pointer to a 64kB memory page. However, super-pages may be defined by specifying a page pointer with a LVL greater than zero. For instance, if T=0 and LVL=1 then the page pointed to is a super-page within an 512MB block of contiguous memory.

T=0, LVL=	Memory Size	Address Bits
0	512 MB	29
1	4 TB	42
2		55
3		68
4		81
5	reserved	
6	reserved	

7	reserved	

A super page pointer contains both a pointer to the block of pages and a super page length field. The length field is provided to restrict memory access to an address range between the super page pointer and the super page pointer plus the number of pages specified in the length. A typical use would be to point to the system ROM which may be several megabytes and yet shorter than the maximum size of the super page.

For example, a system ROM is located 512 MB before the end of physical memory. The ROM is only 1MB in size. So, it is desired to setup a super page pointer to the ROM and restrict access to a single megabyte. The PTE for this would look like:

V	1_5	RGN_3	M	A	0	S	G	SW_2	$\sim_4$	$MRWX_3$	$HRWX_3$	$SRWX_3$	$URWX_3$
$PPN=0x7FFFF_{19}$											$NPG=0x00F_{13}$		
$PPN=0xFFFFFFFF_{63..32}$													

The PTE would be pointed to by a LVL=1 pointer resulting in a 512MB memory block size. 512MB is 1 page before the end of memory, reflected in the value $0x7FFFF_{19}$ for the PPN above. There are 16 x 64kB pages in 1MB so the length field, NPG, is set to $0x00f_{10}$.

PTE FORMAT FOR 512MB PAGE

V	1_5	RGN_3	M	A	0	S	G	SW_2	$\sim_4$	$MRWX_3$	$HRWX_3$	$SRWX_3$	$URWX_3$
$PPN_{31..13}$											NPG_{13}		
$PPN_{63..32}$													

PTE FORMAT FOR 4TB PAGE

V	2_5	RGN_3	M	A	0	S	G	SW_2	$\sim_4$	$MRWX_3$	$HRWX_3$	$SRWX_3$	$URWX_3$
$PPN_{31..28}$	NPG_{26}												
$PPN_{63..32}$													

Root Pointers

A single SPTE provides the address of a 64kB page of memory. With 8192 SPTEs in a page, 512MB of memory can be mapped. This is enough for many applications. A full 32-bit address range can be mapped beginning with root pointers into the space. Only eight pages of MMU tables need be supplied to map an entire 32-bit address space. It would be wasteful and slow to use a 64kB page of main memory to provide pointers to these eight pages. Instead, a dedicated pointer memory which is 4kB in size is used. The pointer memory contains 256 groups of 8 pointers. There is a group of 8 pointers for each address space. 256 address spaces are supported. The pointer memory is indexed by a combination of the address space identifier and the upper three bits of a 32-bit address. Each entry in the pointer memory contains a 16-bit

page number plus a valid bit. The root pointer memory may be accessed via the IO address space located at $FEFCxxxx. This address is relocatable by device configuration.

$PPN_{63..8}$ (0)	$PPN_{7..0}$

TLB – TRANSLATION LOOKASIDE BUFFER

OVERVIEW

A simple page map is limited in the translations it can perform because of its size. The solution to allowing more memory to be mapped is to use main memory to store the translations tables.

However, if every memory access required two or three additional accesses to map the address to a final target access, memory access would be quite slow, slowed down by a factor or two or three, possibly more. To improve performance, the memory mapping translations are stored in another unit called the TLB standing for Translation Lookaside Buffer. This is sometimes also called an address translation cache ATC. The TLB offers a means of address virtualization and memory protection. A TLB works by caching address mappings between a real physical address and a virtual address used by software. The TLB deals with memory organized as pages. Typically, software manages a paging table whose entries are loaded into the TLB as translations are required.

The TLB is a cache specialized for address translations. Qupls's TLB contains 128 two-way associative entries. On a TLB miss the page table is searched for a translation by a hardware- based page table walker and if found the translation is stored in one of the ways of the TLB. The way selected is determined randomly.

SIZE / ORGANIZATION

The TLB has 128 entries per set.

TLB ENTRIES - TLBE

Closely related to page table entries are translation look-aside buffer, TLB, entries. TLB entries have additional fields to match against the virtual address. The count field is used to invalidate the entire TLB. Note that the least significant 7-bits of the virtual address are not stored as these bits are used as an index for the TLB entry.

$Count_6$	LRU_3

V	LVL/BC_5	RGN_3	M	A	T	S	G	SW_2	$CACHE_4$	$MRWX_3$	$HRWX_3$	$SRWX_3$	$URWX_3$
$PPN_{31..0}$													
$PPN_{63..32}$													

$VPN_{38..7}$		
$VPN_{70..39}$		
$ASID_{15..0}$	~3	$VPN_{83..71}$

SMALL TLB ENTRIES - TLBE

The small TLB is used for the test system which contains only 512MB of physical RAM to conserve hardware resources. The address ranges are more limited, 40-bits for the physical address and 70-bits for the virtual address.

$Count_6$	LRU_3

V	LVL/BC_5	RGN_3	M	A	T	S	G	SW_2	$CACHE_4$	$MRWX_3$	$HRWX_3$	$SRWX_3$	$URWX_3$
$\sim_8$								$PPN_{23..0}$					

$VPN_{38..7}$		
$ASID_{15..0}$	PS	$VPN_{53..39}$

WHAT IS TRANSLATED?

The TLB processes addresses including both instruction and data addresses for all modes of operation. It is known as a *unified* TLB.

PAGE SIZE

Because the TLB caches address translations it can get away with a much smaller page size than the page map can for a larger memory system. 4kB is a common size for many systems. There are some indications in contemporary documentation that a larger page size would be better. In this case the TLB uses 64kB. For a 512MB system (the size of the memory in the test system) there are 8192 64kB pages.

WAYS

The TLB is two-way associative.

MANAGEMENT

The TLB unit is updated by a hardware page table walker.

?RWX_3

If RWX3 attributes are specified non-zero, then they will override the attributes coming from the region table. Otherwise RWX attributes are determined by the region table.

$CACHE_4$

The $cache_4$ field is combined with the cache attributes specified in the region table. The region table takes precedence; however, if the $cache_4$ field indicates non-cache-ability then the data will not be cached.

TLB ENTRY REPLACEMENT POLICIES

The TLB uses random replacement. Random replacement chooses a way to replace at random.

FLUSHING THE TLB

The TLB maintains the address space (ASID) associated with a virtual address. This allows the TLB translations to be used without having to flush old translations from the TLB during a task switch.

RESET

On a reset the TLB is preloaded with translations that allow access to the system ROM.

GLOBAL BIT

In addition to the ASID the TLB entries contain a bit that indicates that the translation is a global translation and should be present in every address space.

PTW - PAGE TABLE WALKER

The page table walker is a CPU device used to update the TLB. Whenever a TLB miss occurs the page table walker is triggered. The page table walker walks the page tables to find the translation. Once found the TLB is updated with the translation. If a translation cannot be found then a page fault occurs.

The page table walker managers several variables associated with memory management. These include the page table base register, PT_BASE, page fault address and ASID. These registers are available to software using load and store instructions.

For a page fault the miss address and ASID are stored in a register in the page-table-walker. The PTW also contains the PT_BASE(page table base register) which is used to locate the page table.

The page table walker is a device located in the CPU and has a device configuration block associated with it. The default address of the device is $FF…FF40000.

Register	Name	Description
$FF00	PF_ADDR	Page fault address
$FF10	PF_ASID	Page fault asid
$FF20	PT_BASE	Page table base register
$FF30	PT_ATTR	Page table attributes

PAGE TABLE BASE REGISTER

The page table base register locates the page table in memory. Address bits 3 to 63 are specified. The page table must be octa-byte aligned. Normally the root page table will occupy 64kB of memory and be 64kB aligned. However, for smaller apps it may be desirable to share the memory page the page table is located in with multiple applications.

63 … 3	2 … 0
Page Table Address$_{63..3}$	0

Default Reset Value = 0XFFFFFFFFFFF80000

PAGE TABLE ATTRIBUTES REGISTER

The attributes register contains attributes of the page table.

63 26	25	24 12	11 8	7 6	5 4	3	2 1	0
$\sim_{38}$	~	Root Page Table Limit$_{12..0}$	Levels	AL_2	$\sim_2$	S	~	Type

Type: 0 = inverted page table, 1 = hierarchical page table

S: 1=software managed TLB miss, 0 = hardware table walking, 0 is the only currently supported option.

AL_2: TLB entry replacement algorithm, 0=fixed,1=LRU,2=random,3=reserved, 2 is the only currently supported option.

Levels are ignored for the inverted page table. For a normal page table gives the top entry level.

Root Page Table Limit specifies the number of entries in the root page table. A maximum of 8192 entries is supported.

DEFAULT RESET VALUE = 0X1FFF081

63 26	25	24 12	11 8	7 6	5 4	3	2 1	0
$\sim_{38}$	~	1FFFh	0	2	$\sim_2$	0	~	1

CARD TABLE

OVERVIEW

Also present in the memory system is the Card table. The card table is a telescopic memory which reflects with increasing detail where in the memory system a pointer write has occurred. This is for the benefit of garbage collection systems. Card table is updated using a write barrier when a pointer value is stored to memory, or it may be updated automatically using the STPTR instruction.

ORGANIZATION

At the lowest level memory is divided into 256-byte card memory pages. Each card has a single byte recording whether a pointer store has taken place in the corresponding memory area. To cover a 512MB memory system 2MB card memory is required at the outermost layer. A byte is used rather than a bit to allow byte store operations to update the table directly without having to resort to multiple instructions to perform a bit-field update.

To improve the performance of scanning a hardware card table, HCT, is present which divides memory at an upper level into 8192-byte pages. The hardware card table indicates if a pointer store operation has taken place in one of the 8192-byte pages. It is then necessary to scan only cards representing the 8192-byte page rather than having to scan the entire 2MB card table. Note that this memory is organized as 2048 32-bit words. Allowing 32-bits at a time to be tested.

To further improve performance a master card table, MCT, is present which divides memory at the uppermost layer into 16-MB pages.

Layer	Resolving Power	
0	2 MB	256B pages
1	64k bits	8kB pages
2	32 bits	16 MB pages

There is only a single card memory in the system, used by all tasks.

LOCATION

The card memory location is stored in the region table. A card table may be setup for each region of memory.

OPERATION

As a program progresses it writes pointer values to memory using the write barrier. Storing a pointer triggers an update to all the layers of card memory corresponding to the main memory location written. A

bit or byte is set in each layer of the card memory system corresponding to the memory location of the pointer store.

The garbage collection system can very quickly determine where pointer stores have occurred and skip over memory that has not been modified.

SAMPLE WRITE BARRIER

```
; Milli-code routine for garbage collect write barrier.
; This sequence is short enough to be used in-line.
; Three level card memory.
; a2 is a register pointing to the card table.
; STPTR will cause an update of the master card table, and hardware card table.
;
GCWriteBarrier:
        STPTR   a0,[a1]         ; store the pointer value to memory at a1
        LSR     t0,a1,#8        ; compute card address
        STB     r0,[a2+t0]      ; clear byte in card memory
```

INSTRUCTION SET

OVERVIEW

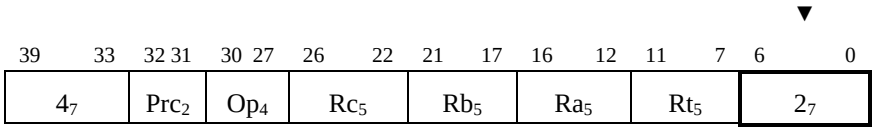
Qupls is a fixed length instruction set with 40-bit instructions. There are several different classes of instructions including arithmetic, memory operate, branch, floating-point and others.

CODE ALIGNMENT

Program code may be relocated at any byte address. However, within a subroutine code should be contiguous.

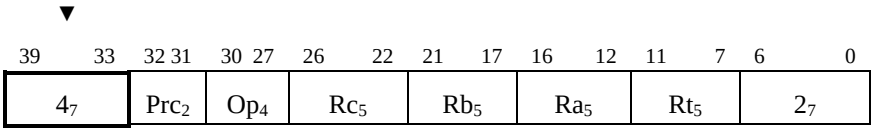
ROOT OPCODE

The root opcode determines the class of instructions executed. Some commonly executed instructions are also encoded at the root level to make more bits available for the instruction. The root opcode is always present in all instructions as the lowest seven bits of the instruction.



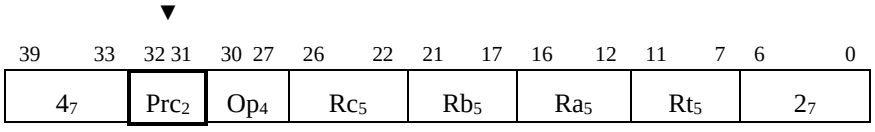
PRIMARY FUNCTION CODE

For register to register operate instructions the primary function code is in the most significant seven bits of the instruction, bits 33 to 39.



PRECISION

Most instructions have a precision specifier. The precision specifier controls how values are treated during the operation. A register may be treated as 1 64-bit value, 2 32-bit values, or 4 16-bit values. The same operation is applied for each value. The location of the precision field varies depending on the class of the instruction.

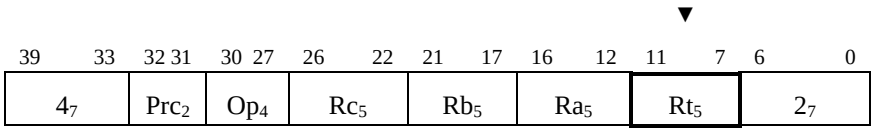


Prc_2	Values
0	4 x 16-bit

1	2 x 32-bit
2	1 x 64-bit
3	reserved

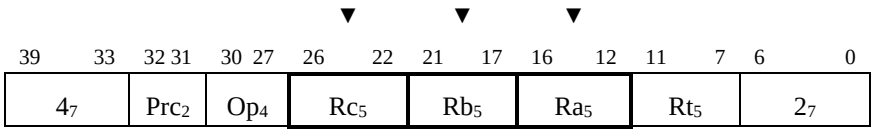
TARGET REGISTER SPEC

Most instructions have a target register. The register spec for the target register is always in the same position, bits 7 to 11 of an instruction.



SOURCE REGISTER SPEC

Most instruction have at least one source register. The register spec for source registers is always in the same position. Bits 12 to 16 for Ra, bits 17 to 21 for Rb, and bits 22 to 26 for Rc.

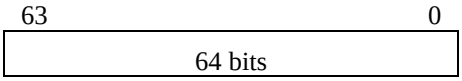


INSTRUCTION DESCRIPTIONS

ARITHMETIC OPERATIONS

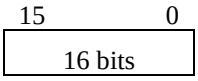
REPRESENTATIONS

INT

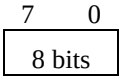


SHORT INT

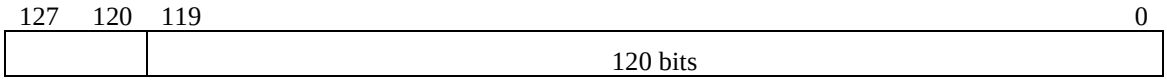
WYDE



BYTE



DECIMAL



Decimal integers use densely packed decimal format which provide 36 digits of precision.

ARITHMETIC OPERATIONS

Arithmetic operations include addition, subtraction, multiplication and division. These are available with the ADD, SUB, CMP, MUL, and DIV instructions. There are several variations of the instructions to deal with signed and unsigned values. The format of the typical immediate mode instruction is shown below:

ADD.sz Rt,Ra,Imm$_{19}$

Instruction Format: RI

39 19	18 17	16 12	11 7	6 0
Immediate$_{20..0}$	Prc$_2$	Ra$_5$	Rt$_5$	4$_7$

There are both signed and unsigned versions of the arithmetic operations.

PRECISION

Four different precisions are supported encoded by the Prc$_2$ field of an instruction. The precision of an operation may be specified with an instruction qualifier following the mnemonic as in ADD.T to add tetras together. The assembler assumes an octa-byte operation if the size is not specified.

Prc$_2$	Register treated as: Bits x lanes	Vector Register
0	16 x 8	16 x 32
1	32 x 4	32 x 16
2	64 x 2	64 x 8
3	128 x 1	128 x 4

VECTOR OPERATIONS

Almost all arithmetic operations have vector forms. The vector opcodes are shown following the scalar ones for the instruction in the instruction description. Vector operations can be identified with the use of vector register specs (Va, Vb, Vc, and Vt).

ADD Rt,Ra,Imm$_{21}$

Instruction Format: RI

39 19	18 17	16 12	11 7	6 0	
Immediate$_{20..0}$	Prc$_2$	Ra$_5$	Rt$_5$	4$_7$	Scalar op
Immediate$_{20..0}$	Prc$_2$	Va$_5$	Vt$_5$	28$_7$	Vector op

CLOCK CYCLES

Vector operations require more clock cycles than scalar ones depending on the number of ALUs available. Shown below is a table for the multiply operation.

Size	Clocks
Octa-byte	4
Vector – 1 ALU	18
Vector – 2 ALU	9

ABS – ABSOLUTE VALUE

Description:

This instruction computes the absolute value of the contents of the source operand and places the result in Rt.

Instruction Format: R1

39 33	32 31	30 27	26 22	21 17	16 12	11 7	6 0
4_7	Prc_2	Op_4	Rc_5	Rb_5	Ra_5	Rt_5	2_7
4_7	Prc_2	Op_4	Vc_5	Vb_5	Va_5	Vt_5	38_7
4_7	Prc_2	Op_4	Vc_5	Rb_5	Va_5	Vt_5	39_7

OP_4		Mnemonic
12	Rt = ABS((Ra + Rb) + Rc)	ABS_SUM
13	Rt = ABS((Ra + Rb) – Rc)	ABS_DIF

Operation:

```
If Source < 0
   Rt = -Source
else
   Rt = Source
```

Execution Units: Integer ALU #0

Clock Cycles: 1

Exceptions: none

Notes:

ADD - REGISTER-REGISTER

Description:

Add three registers and place the sum in the target register. All register values are integers. If Rc is not used, it is assumed to be zero.

Instruction Format: R3

39 33	32 31	30 27	26 22	21 17	16 12	11 7	6 0
4_7	Prc_2	Op_4	Rc_5	Rb_5	Ra_5	Rt_5	2_7
4_7	Prc_2	Op_4	Vc_5	Vb_5	Va_5	Vt_5	38_7
4_7	Prc_2	Op_4	Vc_5	Rb_5	Va_5	Vt_5	39_7

OP_4		Mnemonic
0	Rt = (Ra + Rb) & Rc	ADD_AND
1	Rt = (Ra + Rb) & ~Rc	ADD_ANDC
2	Rt = (Ra + Rb) | Rc	ADD_OR
3	Rt = (Ra + Rb) | ~Rc	ADD_ORC
4	Rt = (Ra + Rb) ^ Rc	ADD_EOR
5	Rt = (Ra + Rb) ^ ~Rc	ADD_EORC
8	Rt = (Ra + Rb) + Rc	ADD_ADD
9	Rt = (Ra + Rb) – Rc	ADD_SUB
10	Rt = (Ra + Rb) + Rc + 1	ADD_ADDPO
11	Rt = (Ra + Rb) + Rc - 1	ADD_ADDMO
12	Rt = ABS((Ra + Rb) + Rc)	ABS_SUM
13	Rt = ABS((Ra + Rb) – Rc)	ABS_DIF
14 to 15	Reserved	

Operation: R3

Rt = Ra + Rb + Rc

Clock Cycles:

Size	Clocks
Octa-byte	1
Vector – 1 ALU	9
Vector – 2 ALU	5

Execution Units: All Integer ALU's

Exceptions: none

Notes:

ADDI - ADD IMMEDIATE

Description:

Add a register and immediate value and place the sum in the target register. The immediate is sign extended to the machine width. This instruction may also be used to calculate a virtual address. It has the same number of displacement bits as a load or store instruction.

Instruction Format: RI

39 19	1817	16 12	11 7	6 0
$Immediate_{20..0}$	Prc_2	Ra_5	Rt_5	4_7
$Immediate_{20..0}$	Prc_2	Va_5	Vt_5	28_7

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = Ra + immediate

Exceptions:

Notes:

ADDSI - ADD SHIFTED IMMEDIATE

Description:

Add an immediate value to a target register. The immediate is shifted left a multiple of 20 bits and sign extended to the machine width. Note the shift is a multiple of only 20 bits while the constant may provide up to 23 bits. The extra three bits may be set to zero or sign extended when building a constant.

The 20-bit increment was chosen to closely match the size supported by other immediate operation instructions like ADDI. Note also that Rt is both a source register and a target register. This provides more bits for the immediate constant. It is envisioned that the vast majority of the time this instruction will follow one which has separate source and target operands.

Instruction Format: RIS

39 17	16 15	14 12	11 7	6 0
$Immediate_{22..0}$	Prc_2	Sc_3	Rt_5	49_7
$Immediate_{22..0}$	Prc_2	Sc_3	Vt_5	48_7

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = Ra + immediate << Sc * 20

Exceptions:

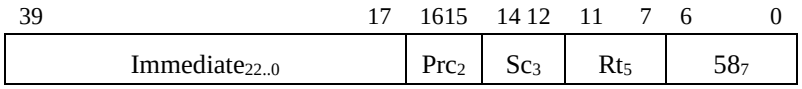
Notes:

AIPSI - ADD SHIFTED IMMEDIATE TO INSTRUCTION POINTER

Description:

Add an immediate value to the instruction pointer and place the result in a target register. The immediate is shifted left a multiple of 20 bits and sign extended to the machine width. Note the shift is a multiple of only 20 bits while the constant may provide up to 23 bits. The extra three bits may be set to zero or sign extended when building a constant. This instruction may be used in the formation of instruction pointer relative addresses.

Instruction Format: RIS



Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = IP + (immediate << (Sc * 20))

Exceptions:

Notes:

BMM – BIT MATRIX MULTIPLY

BMM Rt, Ra, Rb

Description:

The BMM instruction treats the bits of register Ra and register Rb as an 8x8 matrix and performs a bit matrix multiply of the two registers and stores the result in the target register. An alternate mnemonic for this instruction is MOR.

Instruction Format: R3

39 33	3231	30 22	21 17	16 12	11 7	6 0
34_7	2_2	$\sim_9$	Rb_5	Ra_5	Rt_5	2_7
34_7	2_2	$\sim_9$	Vb_5	Va_5	Vt_5	38_7
34_7	2_2	$\sim_9$	Rb_5	Va_5	Vt_5	39_7

Operation:

```
for I = 0 to 7
    for j = 0 to 7
        Rt.bit[i][j] = (Ra[i][0]&Rb[0][j]) | (Ra[i][1]&Rb[1][j]) | … | (Ra[i][7]&Rb[7][j])
```

Clock Cycles: 1

Execution Units: First Integer ALU

Exceptions: none

Notes:

The bits are numbered with bit 63 of a register representing I,j = 0,0 and bit 0 of the register representing I,j = 7,7.

BYTENDX – CHARACTER INDEX

Description:

This instruction searches Ra, which is treated as an array of characters, for a character value specified by Rb and places the index of the character into the target register Rt. If the character is not found -1 is placed in the target register. A common use would be to search for a null character. The index result may vary from -1 to +7. The index of the first found byte is returned (closest to zero). The result is -1 if the character could not be found.

A masking operation may be performed on the Ra operand to allow searches for ranges of characters according to an immediate constant. For instance, the constant could be set to 0xF8 and the mask 'anded' with Ra to search for any ascii control character.

Supported Operand Sizes: .b, .w, .t

Instruction Format: R3 (byte)

39 33	3231	30	29 22	21 17	16 12	11 7	6 0
37_7	Op_2	~	imm_8	Rb_5	Ra_5	Rt_5	2_7
37_7	Op_2	~	imm_8	Vb_5	Va_5	Vt_5	38_7
37_7	Op_2	~	imm_8	Rb_5	Va_5	Vt_5	39_7

Op2	Mask Operation
0	a
1	a & imm
2	a | imm
3	a ^ imm

Operation:

Rt = Index of (Rb in Ra)

Execution Units: All Integer ALU's

Exceptions: none

Notes:

CHARNDX – CHARACTER INDEX

Description:

This instruction searches Ra, which is treated as an array of characters, for a character value specified by Rb and places the index of the character into the target register Rt. If the character is not found -1 is placed in the target register. A common use would be to search for a null character. The index result may vary from -1 to +7. The index of the first found byte is returned (closest to zero). The result is -1 if the character could not be found.

A masking operation may be performed on the Ra operand to allow searches for ranges of characters according to an immediate constant. For instance, the constant could be set to 0xF8 and the mask 'anded' with Ra to search for any ascii control character.

Supported Operand Sizes: .b, .w, .t

Instruction Format: R3 (byte)

39 33	3231	30	29 22	21 17	16 12	11 7	6 0
37_7	Op_2	~	imm_8	Rb_5	Ra_5	Rt_5	2_7
37_7	Op_2	~	imm_8	Vb_5	Va_5	Vt_5	38_7
37_7	Op_2	~	imm_8	Rb_5	Va_5	Vt_5	39_7

Op2	Mask Operation
0	a
1	a & imm
2	a | imm
3	a ^ imm

Operation:

Rt = Index of (Rb in Ra)

Execution Units: All Integer ALU's

Exceptions: none

Notes:

CHK – CHECK REGISTER AGAINST BOUNDS

Description:

A register, Ra, is compared to two values. If the register is outside of the bounds defined by Rb and Rc then an exception specified by the cause code will occur. Comparisons may be signed or unsigned, indicated by 'S', 1 = signed, 0 = unsigned. The constant $Offs_2$ is multiplied by five and added to the instruction pointer address of the CHK instruction and stored on an internal stack. This allows a return to a point up to 15 bytes after the CHK. Typical values are zero or one. The interrupt privilege level IPL is checked against the processor's current IPL and the CHK will be ignored if the complement of the IPL_3 field of the instruction is lower. The IPL_3 field would be set to 0 for most uses.

The CHK instruction cannot be used from within a non-maskable interrupt routine as the IPL will always cause the instruction to be ignored.

Instruction Format: R2

39 36	35	34 27	26 22	21 17	16 12	11 9	8 7	6 0
Op_4	S	$Cause_8$	Rc_5	Rb_5	Ra_5	IPL_3	O_2	0_7

Op_4	exception when not	
0	Ra >= Rb and Ra < Rc	
1	Ra >= Rb and Ra <= Rc	
2	Ra > Rb and Ra < Rc	
3	Ra > Rb and Ra <= Rc	
4	Not (Ra >= Rb and Ra < Rc)	
5	Not (Ra >= Rb and Ra <= Rc)	
6	Not (Ra > Rb and Ra < Rc)	
7	Not (Ra > Rb and Ra <= Rc)	
8	Ra >= CPL	CHKCPL – code privilege level
9	Ra <= CPL	CHKDPL – data privilege level
10	Ra == SC	Stack canary check

Operation:

IF check failed and ~IPL > CPU's IPL

PUSH SR onto internal stack

PUSH IP plus O_2 * 5 onto internal stack

IP = vector at (tvec[3] + cause*8)

Clock Cycles: 1

Execution Units: Integer ALU

Exceptions: bounds check

Notes:

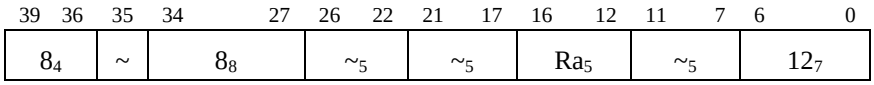
The system exception handler will typically transfer processing back to a local exception handler.

CHKCPL – CHECK CODE PRIVILEGE LEVEL

Description:

A register, Ra, is compared against the CPUs current privilege level. If the register is below the CPL then an exception will occur.

Instruction Format: R2



Op_4	exception when not	
8	Ra >= CPL	CHKCPL – code privilege level

Clock Cycles: 1

Execution Units: Integer ALU

Exceptions: bounds check

Notes:

CLMUL – CARRY-LESS MULTIPLY

Description:

Compute the low order product bits of a carry-less multiply.

Instruction Format: R3

39 33	3231	30 22	21 17	16 12	11 7	6 0
70_7	Prc_2	$\sim_9$	Rb_5	Ra_5	Rt_5	2_7
70_7	Prc_2	$\sim_9$	Vb_5	Va_5	Vt_5	38_7
70_7	Prc_2	$\sim_9$	Rb_5	Va_5	Vt_5	39_7

Exceptions: none

Execution Units: First Integer ALU

Operations

Rt = Ra * Rb

Vector Operation

for x = 0 to VL - 1

if (Vm[x]) Vt[x] = Va[x] * Vb[x]

else if (z) Vt[x] = 0

else Vt[x] = Vt[x]

Exceptions: none

CNTLZ – COUNT LEADING ZEROS

Description:

This instruction counts the number of consecutive zero bits beginning at the most significant bit towards the least significant bit.

Instruction Format: R3

39 33	33	3231	30 27	26 22	21 17	16 12	11 7	6 0
26_7	~	Prc_2	$\sim_4$	0_5	$\sim_5$	Ra_5	Rt_5	2_7
26_7	~	Prc_2	$\sim_4$	0_5	$\sim_5$	Va_5	Vt_5	38_7

Operation:

Execution Units: Integer ALU #0

Clock Cycles: 1

Exceptions: none

Notes:

CNTLO – COUNT LEADING ONES

Description:

This instruction counts the number of consecutive “one” bits beginning at the most significant bit towards the least significant bit.

Instruction Format: R3

39 33	33	32 31	30 27	26 22	21 17	16 12	11 7	6 0
26_7	~	Prc_2	$\sim_4$	1_5	$\sim_5$	Ra_5	Rt_5	2_7
26_7	~	Prc_2	$\sim_4$	1_5	$\sim_5$	Va_5	Vt_5	38_7

Operation:

Execution Units: Integer ALU #0

Clock Cycles: 1

Exceptions: none

Notes:

CNTPOP – COUNT POPULATION

Description:

This instruction counts the number of bits set in a register.

Instruction Format: R3

39 33	33	3231	30 27	26 22	21 17	16 12	11 7	6 0
26_7	~	Prc_2	$~_4$	2_5	$~_5$	Ra_5	Rt_5	2_7
26_7	~	Prc_2	$~_4$	2_5	$~_5$	Va_5	Vt_5	38_7

Operation:

Execution Units: Integer ALU #0

Clock Cycles: 1

Exceptions: none

Notes:

CNTTZ – COUNT TRAILING ZEROS

Description:

This instruction counts the number of consecutive zero bits beginning at the least significant bit towards the most significant bit. This instruction can also be used to get the position of the first one bit from the right-hand side.

Instruction Format: R3

39 33	33	3231	30 27	26 22	21 17	16 12	11 7	6 0
26_7	~	Prc_2	$\sim_4$	6_5	$\sim_5$	Ra_5	Rt_5	2_7
26_7	~	Prc_2	$\sim_4$	6_5	$\sim_5$	Va_5	Vt_5	38_7

Operation:

Execution Units: Integer ALU #0

Clock Cycles: 1

Exceptions: none

Notes:

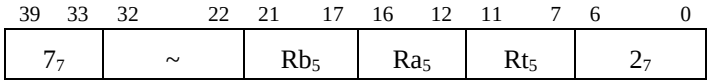
CPUID – GET CPU INFO

Description:

This instruction returns general information about the core. The sum of Rb and register Ra is used as a table index to determine which row of information to return.

Supported Operand Sizes: N/A

Instruction Formats: R3



Clock Cycles: 1

Execution Units: ALU #0 only

Operation:

Rt = Info[Ra+Rb]

Exceptions: none

Index	bits	Information Returned
0	0 to 63	The processor core identification number. This field is determined from an external input. It would be hard wired to the number of the core in a multi-core system.
2	0 to 63	Manufacturer name first eight chars “Finitron”
3	0 to 63	Manufacturer name last eight characters
4	0 to 63	CPU class “64BitSS”
5	0 to 63	CPU class
6	0 to 63	CPU Name “Qupls”
7	0 to 63	CPU Name
8	0 to 63	Model Number “M1”
9	0 to 63	Serial Number “1234”
10	0 to 63	Features bitmap
11	0 to 31	Instruction Cache Size (32kB)
11	32 to 63	Data cache size (64kB)
12	0 to 7	Maximum vector length

CSR – CONTROL AND SPECIAL REGISTERS OPERATIONS

Description:

Perform an operation on a CSR. The previous value of the CSR is placed in the target register.

Operation	**Op_3**	**Mnemonic**
Read CSR	0	CSRRD
Write CSR	1	CSRRW
Or to CSR (set bits)	2	CSRRS
And complement to CSR (clear bits)	3	CSRRC
reserved	4	
Write Immediate CSR	5	CSRRW
Or Immediate to CSR	6	CSRRS
And Immediate complement to CSR	7	CSRRC

Supported Operand Sizes: N/A

Instruction Formats: CSR

39 37	36 31	30 17	16 12	11 7	6 0
Op_3	$\sim_6$	$Regno_{13..0}$	Ra_5	Rt_5	7_7

Instruction Formats: CSRI

39 37	36 31	30 17	16 12	11 7	6 0
Op_3	$Ui_{10..5}$	$Regno_{13..0}$	$Ui_{4..0}$	Rt_5	7_7

Notes:

The top two bits of the Regno field correspond to the operating mode.

DIV – SIGNED DIVISION

Description:

Divide source dividend operand by divisor operand and place the quotient in the target register. All registers are integer registers. Arithmetic is signed twos-complement values.

Instruction Format: R3

39 33	32 31	30 27	26 22	21 17	16 12	11 7	6 0
17_7	Prc_2	$\sim_4$	$\sim_5$	Rb_5	Ra_5	Rt_5	2_7
17_7	Prc_2	$\sim_4$	$\sim_5$	Vb_5	Va_5	Vt_5	38_7
17_7	Prc_2	$\sim_4$	$\sim_5$	Rb_5	Va_5	Vt_5	39_7

Operation:

Rt = Ra / Rb

Size	Clocks	2 ALU
Octa-byte	34	34
Vector Octa-byte	276	138

Execution Units: All Integer ALU's

Exceptions: DBZ

Notes:

DIVI – SIGNED IMMEDIATE DIVISION

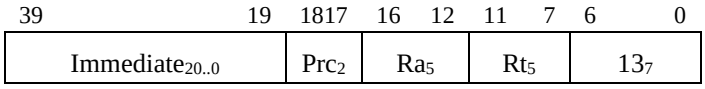
Description:

Divide source dividend operand by divisor operand and place the quotient in the target register. All registers are integer registers. Arithmetic is signed twos-complement values.

Operation:

Rt = Ra / Imm

Instruction Format: RI



Execution Units: All Integer ALU's

Exceptions: none

Notes:

DIVU – UNSIGNED DIVISION

Description:

Divide source dividend operand by divisor operand and place the quotient in the target register. All registers are integer registers. Arithmetic is unsigned twos-complement values.

Instruction Format: R3

39 33	32 31	30 27	26 22	21 17	16 12	11 7	6 0
20_7	Prc_2	$\sim_4$	$\sim_5$	Rb_5	Ra_5	Rt_5	2_7
20_7	Prc_2	$\sim_4$	$\sim_5$	Vb_5	Va_5	Vt_5	38_7
20_7	Prc_2	$\sim_4$	$\sim_5$	Rb_5	Va_5	Vt_5	39_7

Operation:

Rt = Ra / Rb

Size	Clocks	2 ALU
Octa-byte	34	34
Vector Octa-byte	276	138

Execution Units: All Integer ALU's

Exceptions: none

Notes:

DIVUI – UNSIGNED IMMEDIATE DIVISION

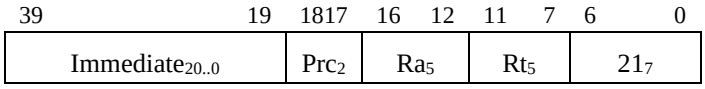
Description:

Divide source dividend operand by divisor operand and place the quotient in the target register. All registers are integer registers. Arithmetic is unsigned twos-complement values.

Operation:

Rt = Ra / Imm

Instruction Format: RI



Execution Units: All Integer ALU's

Exceptions: none

Notes:

LDA – LOAD ADDRESS

Description:

This is an alternate mnemonic for the ADDI instruction. Add a register and immediate value and place the sum in the target register. The immediate is sign extended to the machine width.

Instruction Format: RI

39 19	18 17	16 12	11 7	6 0
$Immediate_{20..0}$	Prc_2	Ra_5	Rt_5	4_7
$Immediate_{20..0}$	Prc_2	Va_5	Vt_5	28_7

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = Ra + immediate

Exceptions:

Notes:

LDAX – LOAD INDEXED ADDRESS

Description:

This instruction computes the scaled indexed virtual address and places it in the target register.

Instruction Format: d[Ra+Rb*Sc]

39 35	34 27	2625	24 19	18 13	12 7	6 0
$\sim_5$	$Disp_{7..0}$	Sc_2	Rb_6	Ra_6	Rt_6	57_7

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = Ra + Rb *Scale + displacement

Exceptions:

Notes:

MADD – MULTIPLY AND ADD

Description:

Multiply two registers add a third and place the product in the target register. All registers are integer registers. Values are treated as signed integers.

Instruction Format: R3

39 33	3231	30 25	24 19	18 13	12 7	6 0
16_7	0	Rc_6	Rb_6	Ra_6	Rt_6	2_7
16_7	0	Vc_6	Vb_6	Va_6	Vt_6	38_7
16_7	0	Vc_6	Rb_6	Va_6	Vt_6	39_7

Size	Clocks
Octa-byte	4
Vector	36

Operation: R2

Rt = Ra * Rb + Rc

Execution Units: All Integer ALU's

Exceptions: none

Notes:

MAJ – MAJORITY LOGIC

Description:

Determines the bitwise majority of three values in registers Ra, Rb and Rc and places the result in the target register Rt.

Instruction Format: R3

39 33	32 31	30 27	26 22	21 17	16 12	11 7	6 0
1_7	Prc_2	15_4	Rc_5	Rb_5	Ra_5	Rt_5	2_7
1_7	Prc_2	15_4	Vc_5	Vb_5	Va_5	Vt_5	38_7
1_7	Prc_2	15_4	Vc_5	Rb_5	Va_5	Vt_5	39_7

Execution Units: ALU #0 only

Operation:

Rt = (Ra & Rb) | (Ra & Rc) | (Rb & Rc)

MUL – MULTIPLY REGISTER-REGISTER

Description:

Multiply two registers and place the product in the target register. All registers are integer registers. Values are treated as signed integers.

Instruction Format: R3

39 33	32 31	30 27	26 22	21 17	16 12	11 7	6 0
16_7	Prc_2	Op_4	Rc_5	Rb_5	Ra_5	Rt_5	2_7
16_7	Prc_2	Op_4	Vc_5	Vb_5	Va_5	Vt_5	38_7
16_7	Prc_2	Op_4	Vc_5	Rb_5	Va_5	Vt_5	39_7

OP_4		Mnemonic
0	Rt = (Ra * Rb) & Rc	MUL_AND
1	Rt = (Ra * Rb) & ~Rc	MUL_ANDC
2	Rt = (Ra * Rb) | Rc	MUL_OR
3	Rt = (Ra * Rb) | ~Rc	MUL_ORC
4	Rt = (Ra * Rb) ^ Rc	MUL_EOR
5	Rt = (Ra * Rb) ^ ~Rc	MUL_EORC
8	Rt = (Ra * Rb) + Rc	MUL_ADD
9	Rt = (Ra * Rb) - Rc	MUL_SUB
14	Rt = -((Ra * Rb) + Rc)	NMUL_ADD
15	Rt = -((Ra * Rb) – Rc)	NMUL_SUB
others	Reserved	

Size	Clocks
Octa-byte	4
Vector	36

Operation: R2

Rt = Ra * Rb + Rc

Execution Units: All Integer ALU's

Exceptions: none

Notes:

MULW – MULTIPLY WIDENING

Description:

Compute the product of two values. Both operands must be in registers. Both the operands are treated as signed values, the result is a signed result.

Instruction Format: R3

39 33	32 31	30 27	26 22	21 17	16 12	11 7	6 0
24_7	Prc_2	$\sim_4$	Rc_5	Rb_5	Ra_5	Rt_5	2_7
24_7	Prc_2	$\sim_4$	Vc_5	Vb_5	Va_5	Vt_5	38_7
24_7	Prc_2	$\sim_4$	Vc_5	Rb_5	Va_5	Vt_5	39_7

Exceptions: none

Execution Units: ALUs (uses both)

Operation

Rt = low bits (Ra * Rb)

Rc = high bits (Ra * Rb)

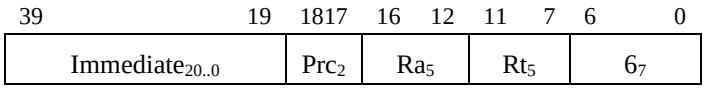
Exceptions: none

MULI - MULTIPLY IMMEDIATE

Description:

Multiply a register and immediate value and place the product in the target register. The immediate is sign extended to the machine width. Values are treated as signed integers.

Instruction Format: RI



Clock Cycles: 4

Execution Units: All ALU's

Operation:

Rt = Ra * immediate

Exceptions:

Notes:

MULSU – MULTIPLY SIGNED UNSIGNED

Description:

Multiply two registers and place the product in the target register. All registers are integer registers. The first operand is signed, the second unsigned.

Instruction Format: R3

39 33	32 31	30 27	26 22	21 17	16 12	11 7	6 0
21_7	Prc_2	$\sim_4$	Rc_5	Rb_5	Ra_5	Rt_5	2_7
21_7	Prc_2	$\sim_4$	Vc_5	Vb_5	Va_5	Vt_5	38_7
21_7	Prc_2	$\sim_4$	Vc_5	Rb_5	Va_5	Vt_5	39_7

Operation: R2

Rt = Ra * Rb + Rc

Clock Cycles: 1

Execution Units: All Integer ALU's

Exceptions: none

Notes:

MULSUH – MULTIPLY SIGNED UNSIGNED HIGH

Description:

Multiply two registers and place the high order product bits in the target register. All registers are integer registers. The first operand is signed, the second unsigned.

Instruction Format: R3

39 33	32 31	30 27	26 22	21 17	16 12	11 7	6 0
29_7	Prc_2	$\sim_4$	Rc_5	Rb_5	Ra_5	Rt_5	2_7
29_7	Prc_2	$\sim_4$	Vc_5	Vb_5	Va_5	Vt_5	38_7
29_7	Prc_2	$\sim_4$	Vc_5	Rb_5	Va_5	Vt_5	39_7

Operation: R2

Rt = Ra * Rb + Rc

Clock Cycles: 1

Execution Units: All Integer ALU's

Exceptions: none

Notes:

MULU – UNSIGNED MULTIPLY REGISTER-REGISTER

Description:

Multiply two registers and place the product in the target register. All registers are integer registers. Values are treated as unsigned integers.

Instruction Format: R3

39 33	32 31	30 27	26 22	21 17	16 12	11 7	6 0
19_7	Prc_2	$\sim_4$	Rc_5	Rb_5	Ra_5	Rt_5	2_7
19_7	Prc_2	$\sim_4$	Vc_5	Vb_5	Va_5	Vt_5	38_7
19_7	Prc_2	$\sim_4$	Vc_5	Rb_5	Va_5	Vt_5	39_7

Operation: R2

Rt = Ra * Rb + Rc

Clock Cycles: 1

Execution Units: All Integer ALU's

Exceptions: none

Notes:

MULUH – UNSIGNED MULTIPLY HIGH

Description:

Multiply two registers and place the high order product bits in the target register. All registers are integer registers. Values are treated as unsigned integers.

Instruction Format: R3

39 33	32 31	30 27	26 22	21 17	16 12	11 7	6 0
27_7	Prc_2	$\sim_4$	Rc_5	Rb_5	Ra_5	Rt_5	2_7
27_7	Prc_2	$\sim_4$	Vc_5	Vb_5	Va_5	Vt_5	38_7
27_7	Prc_2	$\sim_4$	Vc_5	Rb_5	Va_5	Vt_5	39_7

Operation: R2

Rt = Ra * Rb + Rc

Clock Cycles: 1

Execution Units: All Integer ALU's

Exceptions: none

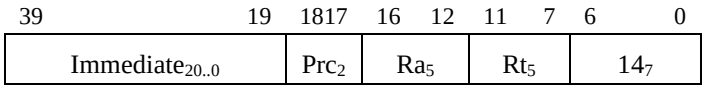
Notes:

MULUI - MULTIPLY UNSIGNED IMMEDIATE

Description:

Multiply a register and immediate value and place the product in the target register. The immediate is sign extended to the machine width. Values are treated as unsigned integers.

Instruction Format: RI



Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = Ra * immediate

Exceptions:

Notes:

PFXN – CONSTANT POSTFIX

Description:

The PFX instruction postfix is used to provide large constants for use in the preceding instruction as the immediate constant for the instruction. The constant postfix may override a source operand of most instructions. PFXA is used to override the A source operand, PFXB is used to override the B source operand, PFXC is used to override the C source operand. Postfixes must be specified in the order PFXA, PFXB, PFXC. A postfix may be omitted if the constant is not needed for the corresponding operand.

Postfixes are normally caught at the decode stage and do not progress further in the pipeline. They are treated as a NOP instruction.

Instruction Format: PFXA

This format provides a 32-bit constant bucket.

39 ——— 8	7	6 — 0
$Immediate_{31..0}$	~	56_7

PTRDIF – DIFFERENCE BETWEEN POINTERS

Asm: PTRDIF Rt, Ra, Rb, Rc

Description:

Subtract two values then shift the result right. Both operands must be in a register. The right shift is provided to accommodate common object sizes. It may still be necessary to perform a divide operation after the PTRDIF to obtain an index into odd sized or large objects. Sc may vary from zero to fifteen.

Instruction Format: R3

39 33	32 31	30 28	27	26 22	21 17	16 12	11 7	6 0
32_7	Prc_2	Op_3	~	Rc_5	Rb_5	Ra_5	Rt_5	2_7
32_7	Prc_2	Op_3	~	Vc_5	Vb_5	Va_5	Vt_5	38_7
32_7	Prc_2	Op_3	~	Vc_5	Rb_5	Va_5	Vt_5	39_7

39 33	32 31	30 28	27 22	21 17	16 12	11 7	6 0
32_7	Prc_2	Op_3	Imm_6	Rb_5	Ra_5	Rt_5	2_7
32_7	Prc_2	Op_3	Imm_6	Vb_5	Va_5	Vt_5	38_7
32_7	Prc_2	Op_3	Imm_6	Rb_5	Va_5	Vt_5	39_7

Op3		Operation	Comment
0	PTRDIF	Rt = abs(Ra – Rb) >> $Rc_{[5:0]}$	
1	AVG	Rt = (Ra + Rb) >> $Rc_{[5:0]}$, trunc	Arithmetic shift right
2	AVG	Rt = (Ra + Rb) >> $Rc_{[5:0]}$, round up	Arithmetic shift right
3		Reserved	
4	PTRDIF	Rt = abs(Ra – Rb) >> Imm_6	
5	AVG	Rt = (Ra + Rb) >> Imm_6, trunc	Arithmetic shift right
6	AVG	Rt = (Ra + Rb) >> Imm_6, round up	Arithmetic shift right
7		Reserved	

Operation:

Rt = Abs(Ra – Rb) >> $Rc_{[3:0]}$

Clock Cycles: 1

Execution Units: ALU #0 only

Exceptions:

None

REM – SIGNED REMAINDER

Description:

Divide source dividend operand by divisor operand and place the remainder in the target register. All registers are integer registers. Arithmetic is signed twos-complement values.

Operation:

Rt = Ra % Rb

Instruction Format: R3

39 33	32 31	30 27	26 22	21 17	16 12	11 7	6 0
25_7	Prc_2	$\sim_4$	Rc_5	Rb_5	Ra_5	Rt_5	2_7
25_7	Prc_2	$\sim_4$	Vc_5	Vb_5	Va_5	Vt_5	38_7
25_7	Prc_2	$\sim_4$	Vc_5	Rb_5	Va_5	Vt_5	39_7

Size	Clocks
Octa-byte	34

Execution Units: All Integer ALU's

Exceptions: DBZ

Notes:

REMU – UNSIGNED REMAINDER

Description:

Divide source dividend operand by divisor operand and place the remainder in the target register. All registers are integer registers. Arithmetic is unsigned twos-complement values.

Operation:

Rt = Ra % Rb

Instruction Format: R3

39 33	32 31	30 27	26 22	21 17	16 12	11 7	6 0
28_7	Prc_2	$\sim_4$	Rc_5	Rb_5	Ra_5	Rt_5	2_7
28_7	Prc_2	$\sim_4$	Vc_5	Vb_5	Va_5	Vt_5	38_7
28_7	Prc_2	$\sim_4$	Vc_5	Rb_5	Va_5	Vt_5	39_7

Size	Clocks
Octa-byte	34

Execution Units: All Integer ALU's

Exceptions: DBZ

Notes:

REVBIT – REVERSE BIT ORDER

Description:

This instruction reverses the order of bits in Ra and stores the result in Rt.

Instruction Format: R1

39 34	33	3231	30 17	16 12	11 7	6 0
5_6	~	Prc_2	$~_{14}$	Ra_5	Rt_5	1_7
5_6	~	Prc_2	$~_{14}$	Va_5	Vt_5	37_7

Operation:

Execution Units: I

Clock Cycles: 1

Exceptions: none

Notes:

SQRT – SQUARE ROOT

Description:

This instruction computes the integer square root of the contents of the source operand and places the result in Rt.

Instruction Format: R3

39 33	33	3231	30 27	26 22	21 17	16 12	11 7	6 0
26_7	~	Prc_2	$~_4$	4_5	$~_5$	Ra_5	Rt_5	2_7
26_7	~	Prc_2	$~_4$	4_5	$~_5$	Va_5	Vt_5	38_7

Operation:

Rt = SQRT(Ra)

Execution Units: Integer ALU #0

Clock Cycles: 1

Exceptions: none

Notes:

SUB – SUBTRACT REGISTER-REGISTER

Description:

Subtract three registers and place the difference in the target register. All registers are integer registers.

Instruction Format: R3

39 33	32 31	30 27	26 22	21 17	16 12	11 7	6 0
5_7	Prc_2	$\sim_4$	Rc_5	Rb_5	Ra_5	Rt_5	2_7
5_7	Prc_2	$\sim_4$	Vc_5	Vb_5	Va_5	Vt_5	38_7
5_7	Prc_2	$\sim_4$	Vc_5	Rb_5	Va_5	Vt_5	39_7

Operation: R3

Rt = Ra – Rb - Rc

Clock Cycles: 1

Execution Units: All Integer ALU's

Exceptions: none

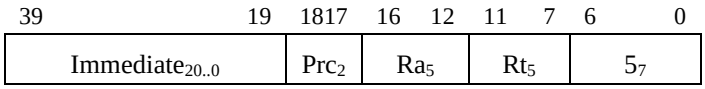
Notes:

SUBFI – SUBTRACT FROM IMMEDIATE

Description:

Subtract a register from an immediate value and place the difference in the target register. The immediate is sign extended to the machine width.

Instruction Format: RI



Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = immediate - Ra

Exceptions:

Notes:

TETRANDX – CHARACTER INDEX

Description:

This instruction searches Ra, which is treated as an array of characters, for a character value specified by Rb and places the index of the character into the target register Rt. If the character is not found -1 is placed in the target register. A common use would be to search for a null character. The index result may vary from -1 to +1. The index of the first found tetra is returned (closest to zero). The result is -1 if the character could not be found.

A masking operation may be performed on the Ra operand to allow searches for ranges of characters according to an immediate constant. For instance, the constant could be set to 0x1F8 and the mask 'anded' with Ra to search for any ascii control character.

Supported Operand Sizes: .b, .w, .t

Instruction Format: R3 (tetra)

39 33	3231	30 22	21 17	16 12	11 7	6 0
39_7	Op_2	Imm_9	Rb_5	Ra_5	Rt_5	2_7
39_7	Op_2	Imm_9	Vb_5	Va_5	Vt_5	38_7
39_7	Op_2	Imm_9	Rb_5	Va_5	Vt_5	39_7

Op2	Mask Operation
0	a
1	a & imm
2	a | imm
3	a ^ imm

Operation:

Rt = Index of (Rb in Ra)

Execution Units: All Integer ALU's

Exceptions: none

Notes:

WYDENDX – CHARACTER INDEX

Description:

This instruction searches Ra, which is treated as an array of characters, for a character value specified by Rb and places the index of the character into the target register Rt. If the character is not found -1 is placed in the target register. A common use would be to search for a null character. The index result may vary from -1 to +3. The index of the first found wyde is returned (closest to zero). The result is -1 if the character could not be found.

A masking operation may be performed on the Ra operand to allow searches for ranges of characters according to an immediate constant. For instance, the constant could be set to 0x1F8 and the mask 'anded' with Ra to search for any ascii control character.

Supported Operand Sizes: .b, .w, .t

Instruction Format: R3 (wyde)

39 33	3231	30 22	21 17	16 12	11 7	6 0
38_7	Op_2	Imm_9	Rb_5	Ra_5	Rt_5	2_7
38_7	Op_2	Imm_9	Vb_5	Va_5	Vt_5	38_7
38_7	Op_2	Imm_9	Rb_5	Va_5	Vt_5	39_7

Op2	Mask Operation
0	a
1	a & imm
2	a | imm
3	a ^ imm

Operation:

Rt = Index of (Rb in Ra)

Execution Units: All Integer ALU's

Exceptions: none

Notes:

BMAP – BYTE MAP

Description:

First the target register is cleared, then bytes are mapped from the 16-byte source Ra into bytes in the target register. This instruction may be used to permute the bytes in register Ra and store the result in Rt. This instruction may also pack bytes, wydes or tetras. The map is determined by the low order 64-bits of register Rb. Bytes which are not mapped will end up as zero in the target register. Each nybble of the 64-bit value indicates the target byte in the target register.

Instruction Format: R3

BMAP Rt, Ra, Rb

39 33	3231	30 22	21 17	16 12	11 7	6 0
35_7	Prc_2	$\sim_9$	Rb_5	Ra_5	Rt_5	2_7
35_7	Prc_2	$\sim_9$	Vb_5	Va_5	Vt_5	38_7
35_7	Prc_2	$\sim_9$	Rb_5	Va_5	Vt_5	39_7

Operation:

Vector Operation

Execution Units: First Integer ALU

Exceptions: none

Notes:

CMOVNZ – CONDITIONAL MOVE IF NON-ZERO

CMOVNZ Rt, Ra, Rb, Rc

Description:

If Ra is non-zero then the target register is set to Rb, otherwise the target register is to Rc.

Instruction Format: R3

39 33	32 31	30 27	26 22	21 17	16 12	11 7	6 0
12_7	$\sim_2$	$\sim_4$	Rc_5	Rb_5	Ra_5	Rt_5	2_7
12_7	$\sim_2$	$\sim_4$	Vc_5	Vb_5	Va_5	Vt_5	38_7
12_7	$\sim_2$	$\sim_4$	Vc_5	Rb_5	Va_5	Vt_5	39_7

Clock Cycles: 1

Execution Units: ALU #0 only

Operation:

```
If Ra then
        Rt = Rb
else
        Rt = Rc
```

Exceptions: none

CMOVZ – CONDITIONAL MOVE IF ZERO

CMOVZ Rt, Ra, Rb, Rc

Description:

If Ra is zero then the target register is set to Rb, otherwise the target register is to Rc.

Instruction Format: R3

39 33	32 31	30 27	26 22	21 17	16 12	11 7	6 0
11_7	$\sim_2$	$\sim_4$	Rc_5	Rb_5	Ra_5	Rt_5	2_7
11_7	$\sim_2$	$\sim_4$	Vc_5	Vb_5	Va_5	Vt_5	38_7
11_7	$\sim_2$	$\sim_4$	Vc_5	Rb_5	Va_5	Vt_5	39_7

Clock Cycles: 1

Execution Units: ALU #0 only

Operation:

```
If Ra = 0 then
        Rt = Rb
else
        Rt = Rc
```

Exceptions: none

MAX3 – MAXIMUM SIGNED VALUE

Description:

Determines the maximum of three values in registers Ra, Rb and Rc and places the result in the target register Rt.

Instruction Format: R3

39 33	32 31	30 27	26 22	21 17	16 12	11 7	6 0
18_7	Prc_2	1_4	Rc_5	Rb_5	Ra_5	Rt_5	2_7
18_7	Prc_2	1_4	Vc_5	Vb_5	Va_5	Vt_5	38_7
18_7	Prc_2	1_4	Vc_5	Rb_5	Va_5	Vt_5	39_7

Execution Units: ALU #0 only

Operation:

```
IF (Ra > Rb and Ra > Rc)
        Rt = Ra
Else if (Rb > Rc)
        Rt = Rb
Else
        Rt = Rc
```

MAXU3 – MAXIMUM UNSIGNED VALUE

Description:

Determines the maximum of three values in registers Ra, Rb and Rc and places the result in the target register Rt. Values are unsigned integers.

Instruction Format: R3

39 33	32 31	30 27	26 22	21 17	16 12	11 7	6 0
18_7	Prc_2	5_4	Rc_5	Rb_5	Ra_5	Rt_5	2_7
18_7	Prc_2	5_4	Vc_5	Vb_5	Va_5	Vt_5	38_7
18_7	Prc_2	5_4	Vc_5	Rb_5	Va_5	Vt_5	39_7

Execution Units: ALU #0 only

Operation:

```
IF (Ra > Rb and Ra > Rc)
        Rt = Ra
Else if (Rb > Rc)
        Rt = Rb
Else
        Rt = Rc
```

MID3 – MIDDLE VALUE

Description:

Determines the middle value of three values in registers Ra, Rb and Rc and places the result in the target register Rt.

Instruction Format: R3

39 33	32 31	30 27	26 22	21 17	16 12	11 7	6 0
18_7	Prc_2	2_4	Rc_5	Rb_5	Ra_5	Rt_5	2_7
18_7	Prc_2	2_4	Vc_5	Vb_5	Va_5	Vt_5	38_7
18_7	Prc_2	2_4	Vc_5	Rb_5	Va_5	Vt_5	39_7

Execution Units: ALU #0 only

Operation:

```
IF (Ra > Rb and Ra < Rc)
        Rt = Ra
Else if (Rb > Ra and Rb < Rc)
        Rt = Rb
Else
        Rt = Rc
```

MIDU3 – MIDDLE UNSIGNED VALUE

Description:

Determines the middle value of three values in registers Ra, Rb and Rc and places the result in the target register Rt.

Instruction Format: R3

39 33	32 31	30 27	26 22	21 17	16 12	11 7	6 0
18_7	Prc_2	6_4	Rc_5	Rb_5	Ra_5	Rt_5	2_7
18_7	Prc_2	6_4	Vc_5	Vb_5	Va_5	Vt_5	38_7
18_7	Prc_2	6_4	Vc_5	Rb_5	Va_5	Vt_5	39_7

Execution Units: ALU #0 only

Operation:

```
IF (Ra > Rb and Ra < Rc)
        Rt = Ra
Else if (Rb > Ra and Rb < Rc)
        Rt = Rb
Else
        Rt = Rc
```

MIN3 – MINIMUM VALUE

Description:

Determines the minimum of three values in registers Ra, Rb and Rc and places the result in the target register Rt.

Instruction Format: R3

39 33	32 31	30 27	26 22	21 17	16 12	11 7	6 0
18_7	Prc_2	0_4	Rc_5	Rb_5	Ra_5	Rt_5	2_7
18_7	Prc_2	0_4	Vc_5	Vb_5	Va_5	Vt_5	38_7
18_7	Prc_2	0_4	Vc_5	Rb_5	Va_5	Vt_5	39_7

Execution Units: ALU #0 only

Operation:

```
IF (Ra < Rb and Ra < Rc)
        Rt = Ra
Else if (Rb < Rc)
        Rt = Rb
Else
        Rt = Rc
```

MINU3 – MINIMUM UNSIGNED VALUE

Description:

Determines the minimum of three values in registers Ra, Rb and Rc and places the result in the target register Rt.

Instruction Format: R3

39 33	32 31	30 27	26 22	21 17	16 12	11 7	6 0
18_7	Prc_2	4_4	Rc_5	Rb_5	Ra_5	Rt_5	2_7
18_7	Prc_2	4_4	Vc_5	Vb_5	Va_5	Vt_5	38_7
18_7	Prc_2	4_4	Vc_5	Rb_5	Va_5	Vt_5	39_7

Execution Units: ALU #0 only

Operation:

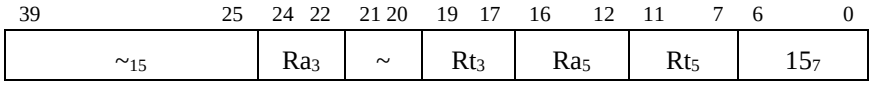
```
IF (Ra < Rb and Ra < Rc)
        Rt = Ra
Else if (Rb < Rc)
        Rt = Rb
Else
        Rt = Rc
```

MOV – MOVE REGISTER TO REGISTER

Description:

Move register-to-register. This instruction may move between different types of registers. Raw binary data is moved. No data conversions are applied. This instruction may move between vector elements and scalar registers. Some registers are accessible only in specific operating modes. Some registers are read-only.

Instruction Format: R1



Operation: R2

Rt = Ra

Clock Cycles: 1

Execution Units: All Integer ALU's

Exceptions: none

Notes:

Ra_8 / Rt_8	Register file	Mode Access	RW
0 to 31	General purpose registers 0 to 31	USHM	RW
32	Machine stack pointer	M	RW
33 to 39	Interrupt stack pointers IPL1 to IPL7	M	RW
40	User stack pointer	USHM	RW
41	Supervisor stack pointer	SHM	RW
42	Hypervisor stack pointer	HM	RW
48 to 51	Micro-code temporaries #0 to #3		RW
54	Vector restart mask		R
55	Vector exception record		R
63	Card table address		RW
52	Micro-code link register		RW
64 to 255	Vector register #8 to #31		RW

MOVSXB – MOVE, SIGN EXTEND BYTE

MOVSXB Rt, Ra

Description:

A byte is extracted from the source operand, sign extended, and the result placed in the target register.

Instruction Format: R3

39 33	3231	30 29	28	27 26 22	21 17	16 12	11 7	6 0
90_7	Prc_2	3	0	7_6	0_5	Ra_5	Rt_5	2_7
90_7	Prc_2	3	0	7_6	0_5	Va_5	Vt_5	38_7

Operation:

Clock Cycles:

Execution Units: All Integer ALU's

Exceptions: none

Notes:

MOVSXT – MOVE, SIGN EXTEND TETRA

MOVSXT Rt, Ra

Description:

A tetra is extracted from the source operand, sign extended, and the result placed in the target register.

Instruction Format: R3

39 33	3231	30 29	28	27 26 22	21 17	16 12	11 7	6 0
90_7	Prc_2	3	0	31_6	0_5	Ra_5	Rt_5	2_7
90_7	Prc_2	3	0	31_6	0_5	Va_5	Vt_5	38_7

Operation:

Clock Cycles:

Execution Units: All Integer ALU's

Exceptions: none

Notes:

MOVSXW – MOVE, SIGN EXTEND WYDE

MOVSXW Rt, Ra

Description:

A wyde is extracted from the source operand, sign extended, and the result placed in the target register.

Instruction Format: R3

39 33	3231	30 29	28	27 26 22	21 17	16 12	11 7	6 0
90_7	Prc_2	3	0	15_6	0_5	Ra_5	Rt_5	2_7
90_7	Prc_2	3	0	15_6	0_5	Va_5	Vt_5	38_7

Operation:

Clock Cycles:

Execution Units: All Integer ALU's

Exceptions: none

Notes:

MUX – MULTIPLEX

MUX Rt, Ra, Rb, Rc

Description:

If a bit in Ra is set then the bit of the target register is set to the corresponding bit in Rb, otherwise the bit in the target register is set to the corresponding bit in Rc.

Instruction Format: R3

39 33	32 31	30 27	26 22	21 17	16 12	11 7	6 0
33_7	Prc_2	$\sim_4$	Rc_5	Rb_5	Ra_5	Rt_5	2_7
33_7	Prc_2	$\sim_4$	Vc_5	Vb_5	Va_5	Vt_5	38_7
33_7	Prc_2	$\sim_4$	Vc_5	Rb_5	Va_5	Vt_5	39_7

Clock Cycles: 1

Execution Units: ALU #0 only

Operation:

For n = 0 to 63

If $Ra_{[n]}$ is set then

$Rt_{[n]} = Rb_{[n]}$

else

$Rt_{[n]} = Rc_{[n]}$

Exceptions: none

LOGICAL OPERATIONS

AND – BITWISE AND

Description:

Bitwise ‘and’ two registers with the complement of a third and place the result in the target register. All registers are treated as integer registers.

Instruction Format: R3

39 33	32 31	30 27	26 22	21 17	16 12	11 7	6 0
0_7	Prc_2	Op_4	Rc_5	Rb_5	Ra_5	Rt_5	2_7
0_7	Prc_2	Op_4	Vc_5	Vb_5	Va_5	Vt_5	38_7
0_7	Prc_2	Op_4	Vc_5	Rb_5	Va_5	Vt_5	39_7

OP_4		Mnemonic
0	Rt = (Ra & Rb) & Rc	AND_AND
1	Rt = (Ra & Rb) & ~Rc	AND_ANDC
2	Rt = (Ra & Rb) | Rc	AND_OR
3	Rt = (Ra & Rb) | ~Rc	AND_ORC
4	Rt = (Ra & Rb) ^ Rc	AND_EOR
5	Rt = (Ra & Rb) ^ ~Rc	AND_EORC
6 to 15	Reserved	

Operation: R3

Rt = Ra & Rb & ~Rc

Clock Cycles: 1

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

ANDI – BITWISE ‘AND’ IMMEDIATE

Description:

Bitwise ‘And’ a register and immediate value and place the result in the target register. The immediate is one extended to the machine width.

Instruction Format: RI

39 19	1817	16 12	11 7	6 0
$Immediate_{20..0}$	Prc_2	Ra_5	Rt_5	8_7
$Immediate_{20..0}$	Prc_2	Va_5	Vt_5	24_7

Clock Cycles: 1

Execution Units: All ALU’s

Operation:

Rt = Ra + immediate

Exceptions:

Notes:

ANDSI – BITWISE ‘AND’ SHIFTED IMMEDIATE

Description:

Bitwise ‘And’ an immediate value to a target register. The immediate is shifted left a multiple of 20 bits and one extended to both the left and right for the machine width. Note the shift is a multiple of only 20 bits while the constant may provide up to 23 bits. The extra three bits may be set to zero or sign extended when building a constant.

Instruction Format: RIS

39 17	16 15	14 12	11 7	6 0
$Immediate_{22..0}$	Prc_2	Sc_3	Rt_5	50_7
$Immediate_{22..0}$	Prc_2	Sc_3	Vt_5	56_7

Clock Cycles: 1

Execution Units: All ALU’s

Operation:

Rt = Ra & immediate << Sc * 20

Exceptions:

Notes:

ENOR – BITWISE EXCLUSIVE NOR

Description:

Bitwise exclusively nor three registers and place the result in the target register. All registers are integer registers.

Instruction Format: R3

39 33	32 31	30 27	26 22	21 17	16 12	11 7	6 0
10_7	Prc_2	$\sim_4$	Rc_5	Rb_5	Ra_5	Rt_5	2_7
10_7	Prc_2	$\sim_4$	Vc_5	Vb_5	Va_5	Vt_5	38_7
10_7	Prc_2	$\sim_4$	Vc_5	Rb_5	Va_5	Vt_5	39_7

Operation:

Rt = ~(Ra ^ Rb ^ Rc)

Clock Cycles: 1

Execution Units: All Integer ALU's

Exceptions: none

Notes:

EOR – BITWISE EXCLUSIVE OR

Description:

Bitwise exclusively or three registers and place the result in the target register. All registers are integer registers.

Instruction Format: R3

39 33	32 31	30 27	26 22	21 17	16 12	11 7	6 0
2_7	Prc_2	$\sim_4$	Rc_5	Rb_5	Ra_5	Rt_5	2_7
2_7	Prc_2	$\sim_4$	Vc_5	Vb_5	Va_5	Vt_5	38_7
2_7	Prc_2	$\sim_4$	Vc_5	Rb_5	Va_5	Vt_5	39_7

OP_4		Mnemonic
0	Rt = (Ra ^ Rb) & Rc	EOR_AND
1	Rt = (Ra ^ Rb) & ~Rc	EOR_ANDC
2	Rt = (Ra ^ Rb) | Rc	EOR_OR
3	Rt = (Ra ^ Rb) | ~Rc	EOR_ORC
4	Rt = (Ra ^ Rb) ^ Rc	EOR_EOR
5	Rt = (Ra ^ Rb) ^ ~Rc	EOR_EORC
6 to 14	Reserved	
15	Rt = (^Ra) ^ (^Rb) ^ (^Rc)	PAR (triple parity)

Operation: R3

Rt = Ra ^ Rb ^ Rc

Clock Cycles: 1

Execution Units: All Integer ALU's

Exceptions: none

Notes:

EORI – EXCLUSIVE OR IMMEDIATE

Description:

Exclusive Or a register and immediate value and place the sum in the target register. The immediate is zero extended to the machine width.

Instruction Format: RI

39 19	1817	16 12	11 7	6 0
$Immediate_{20..0}$	Prc_2	Ra_5	Rt_5	10_7
$Immediate_{20..0}$	Prc_2	Va_5	Vt_5	26_7

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = Ra ^ immediate

Exceptions:

Notes:

EORSI – BITWISE EXCLUSIVE ‘OR’ SHIFTED IMMEDIATE

Description:

Bitwise exclusive ‘or’ a register and immediate value and place the result in the target register. The immediate is shifted left in multiples of 20 bits and zero extended to the machine width. This instruction may be used to build a large constant in a register. Note the shift is a multiple of only 20 bits while the constant may provide up to 23 bits. The extra three bits may be set to zero when building a constant.

The 20-bit increment was chosen to closely match the size supported by other immediate operation instructions like ORI. It is also straightforward to implement in hardware. Note also that Rt is both a source register and a target register. This provides more bits for the immediate constant. It is envisioned that the vast majority of the time this instruction will follow one which has separate source and target operands.

Instruction Format: RIS

39 17	16 15	14 12	11 7	6 0
$Immediate_{22..0}$	Prc_2	Sc_3	Rt_5	59_7
$Immediate_{22..0}$	Prc_2	Sc_3	Vt_5	61_7

Clock Cycles: 1

Execution Units: All ALU’s

Operation:

Rt = Ra ^ (immediate << Sc * 16)

Exceptions:

Notes:

NAND – BITWISE NAND

Description:

Bitwise nand two registers and place the result in the target register. All registers are integer registers.

Instruction Format: R3

39 33	32 31	30 27	26 22	21 17	16 12	11 7	6 0
8_7	Prc_2	Op_4	Rc_5	Rb_5	Ra_5	Rt_5	2_7
8_7	Prc_2	Op_4	Vc_5	Vb_5	Va_5	Vt_5	38_7
8_7	Prc_2	Op_4	Vc_5	Rb_5	Va_5	Vt_5	39_7

OP_4		Mnemonic
0	Rt = ~(Ra & Rb) & Rc	NAND_AND
1	Rt = ~(Ra & Rb) & ~Rc	NAND_ANDC
2	Rt = ~(Ra & Rb) | Rc	NAND_OR
3	Rt = ~(Ra & Rb) | ~Rc	NAND_ORC
4	Rt = ~(Ra & Rb) ^ Rc	NAND_EOR
5	Rt = ~(Ra & Rb) ^ ~Rc	NAND_EORC
6 to 15	Reserved	

Operation: R2

Rt = ~(Ra & Rb & ~Rc)

Clock Cycles: 1

Execution Units: All Integer ALU's

Exceptions: none

Notes:

NOR – BITWISE OR

Description:

Bitwise nor two registers and place the result in the target register. All registers are integer registers.

Instruction Format: R3

39 33	32 31	30 27	26 22	21 17	16 12	11 7	6 0
9_7	Prc_2	Op_4	Rc_5	Rb_5	Ra_5	Rt_5	2_7
9_7	Prc_2	Op_4	Vc_5	Vb_5	Va_5	Vt_5	38_7
9_7	Prc_2	Op_4	Vc_5	Rb_5	Va_5	Vt_5	39_7

OP_4		Mnemonic
0	Rt = ~(Ra | Rb) & Rc	NOR_AND
1	Rt = ~(Ra | Rb) & ~Rc	NOR_ANDC
2	Rt = ~(Ra | Rb) | Rc	NOR_OR
3	Rt = ~(Ra | Rb) | ~Rc	NOR_ORC
4	Rt = ~(Ra | Rb) ^ Rc	NOR_EOR
5	Rt = ~(Ra | Rb) ^ ~Rc	NOR_EORC
6 to 15	Reserved	

Operation: R2

Rt = ~(Ra | Rb | Rc)

Clock Cycles: 1

Execution Units: All Integer ALU's

Exceptions: none

Notes:

OR – BITWISE OR

Description:

Bitwise or three registers and place the result in the target register. All registers are integer registers.

Instruction Format: R3

39 33	32 31	30 27	26 22	21 17	16 12	11 7	6 0
1_7	Prc_2	Op_4	Rc_5	Rb_5	Ra_5	Rt_5	2_7
1_7	Prc_2	Op_4	Vc_5	Vb_5	Va_5	Vt_5	38_7
1_7	Prc_2	Op_4	Vc_5	Rb_5	Va_5	Vt_5	39_7

OP_4		Mnemonic
0	Rt = (Ra | Rb) & Rc	OR_AND
1	Rt = (Ra | Rb) & ~Rc	OR_ANDC
2	Rt = (Ra | Rb) | Rc	OR_OR
3	Rt = (Ra | Rb) | ~Rc	OR_ORC
4	Rt = (Ra | Rb) ^ Rc	OR_EOR
5	Rt = (Ra | Rb) ^ ~Rc	OR_EORC
6 to 14	Reserved	
15	Rt = (Ra & Rb) | (Ra & Rc) | (Rb & Rc)	MAJ

Operation: R2

Rt = Ra | Rb | Rc

Clock Cycles: 1

Execution Units: All Integer ALU's

Exceptions: none

Notes:

ORC – BITWISE OR COMPLEMENT

Description:

Bitwise ‘or’ a source register and the complement of a second source register and place the result in the target register. All registers are integer registers.

Instruction Format: R3

39 33	32 31	30 27	26 22	21 17	16 12	11 7	6 0
1_7	Prc_2	3_4	Rc_5	Rb_5	Ra_5	Rt_5	2_7
1_7	Prc_2	3_4	Vc_5	Vb_5	Va_5	Vt_5	38_7
1_7	Prc_2	3_4	Vc_5	Rb_5	Va_5	Vt_5	39_7

Operation: R2

Rt = Ra | Rb | ~Rc

Clock Cycles: 1

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

ORI - OR IMMEDIATE

Description:

Or a register and immediate value and place the sum in the target register. The immediate is zero extended to the machine width.

Instruction Format: RI

39 19	1817	16 12	11 7	6 0
$Immediate_{20..0}$	Prc_2	Ra_5	Rt_5	9_7
$Immediate_{20..0}$	Prc_2	Va_5	Vt_5	25_7

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = Ra + immediate

Exceptions:

Notes:

ORSI – SHIFT AND BITWISE 'OR' IMMEDIATE

Description:

Bitwise 'or' the register and shifted immediate value and place the result in the target register. The immediate is shifted left in multiples of 20 bits and zero extended to the machine width. This instruction may be used to build a large constant in a register.

Note that Rt is both a source register and a target register. This provides more bits for the immediate constant. It is envisioned that the vast majority of the time this instruction will follow one which has separate source and target operands.

Instruction Format: RIS

39 17	16 15	14 12	11 7	6 0
$Immediate_{22..0}$	Prc_2	Sc_3	Rt_5	51_7
$Immediate_{22..0}$	Prc_2	Sc_3	Vt_5	60_7

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = Rt | (immediate << Sc * 20)

Exceptions:

Notes:

COMPARISON OPERATIONS

OVERVIEW

There are two basic types of comparison operators. The first type, compare, returns a bit vector indicating the relationship between the operands, the second type, set, returns a false or a constant depending on the result of the comparison.

CMP - COMPARISON

Description:

Compare two source operands and place the result in the target register. The result is a bit vector identifying the relationship between the two source operands as signed integers. The compare may be cumulative by or'ing the result of previous comparisons with the current one. This may be used to test for the presence or absence of data in an array.

Instruction Format: R3

39 33	32 31	30 27	26 22	21 17	16 12	11 7	6 0
3_7	Prc_2	Op_4	Rc_5	Rb_5	Ra_5	Rt_5	2_7
3_7	Prc_2	Op_4	Vc_5	Vb_5	Va_5	Vt_5	38_7
3_7	Prc_2	Op_4	Vc_5	Rb_5	Va_5	Vt_5	39_7

OP_4		Mnemonic
0	Rt = (Ra ? Rb) & Rc	CMP_AND
1	Rt = (Ra ? Rb) & ~Rc	CMP_ANDC
2	Rt = (Ra ? Rb) | Rc	CMP_OR
3	Rt = (Ra ? Rb) | ~Rc	CMP_ORC
4	Rt = (Ra ? Rb) ^ Rc	CMP_EOR
5	Rt = (Ra ? Rb) ^ ~Rc	CMP_EORC
6 to 7	Reserved	
8	Range Check	CMP_RNG
9 to 15	reserved	

Operation:

Rt = Ra ? Rb

Rt = (Ra ? Rb) | Rc ; cumulative

Clock Cycles: 1

Execution Units: All Integer ALU's

Exceptions: none

Notes:

Rt Bit	Mnem.	Meaning	Test
		Integer Compare Results	
0	EQ	= equal	a == b
1	NE	< > not equal	a <> b
2	LT	< less than	a < b
3	LE	<= less than or equal	a <= b
4	GE	>= greater than or equal	a >= b
5	GT	> greater than	a > b
6	BC	Bit clear	!a[b]
7	BS	Bit set	a[b]

Range Check:

Rt Bit	Mnem.	Meaning	Test
		Integer Compare Results	
0	GEL		a >=b and a < c
1	GELE		a >= b and a <= c
2	GL		a > b and a < c
3	GLE		a > b and a <= c
4	NGEL		Not (a >= b and a < c)
5	NGELE		Not (a >= b and a <= c)
6	NGL		Not (a > b and a < c)
7	NGLE		Not (a > b and a <= c)

CMPI – COMPARE IMMEDIATE

Description:

Compare two source operands and place the result in the target register. The result is a vector identifying the relationship between the two source operands as signed and unsigned integers.

Operation:

Rt = Ra ? Imm

Clock Cycles: 1

Execution Units: All Integer ALU's

Exceptions: none

Notes:

Instruction Format: RI

39 19	18 17	16 12	11 7	6 0
$Immediate_{20..0}$	Prc_2	Ra_5	Rt_5	11_7
$Immediate_{20..0}$	Prc_2	Va_5	Vt_5	27_7

Rt Bit	Mnem.	Meaning	Test
		Integer Compare Results	
0	EQ	= equal	a == b
1	NE	< > not equal	a <> b
2	LT	< less than	a < b
3	LE	<= less than or equal	a <= b
4	GE	>= greater than or equal	a >= b
5	GT	> greater than	a > b
6	BC	Bit clear	!a[b]
7	BS	Bit set	a[b]

CMPU – UNSIGNED COMPARISON

Description:

Compare two source operands and place the result in the target register. The result is a bit vector identifying the relationship between the two source operands as unsigned integers.

Instruction Format: R3

39 33	32 31	30 27	26 22	21 17	16 12	11 7	6 0
6_7	Prc_2	Op_4	Rc_5	Rb_5	Ra_5	Rt_5	2_7
6_7	Prc_2	Op_4	Vc_5	Vb_5	Va_5	Vt_5	38_7
6_7	Prc_2	Op_4	Vc_5	Rb_5	Va_5	Vt_5	39_7

Op_4		Mnemonic
0	Rt = (Ra ? Rb) & Rc	CMPU_AND
1	Rt = (Ra ? Rb) & ~Rc	CMPU_ANDC
2	Rt = (Ra ? Rb) | Rc	CMPU_OR
3	Rt = (Ra ? Rb) | ~Rc	CMPU_ORC
4	Rt = (Ra ? Rb) ^ Rc	CMPU_EOR
5	Rt = (Ra ? Rb) ^ ~Rc	CMPU_EORC
6 to 7	Reserved	
8	Range Check	CMPU_RNG
9 to 15	reserved	

Operation:

Rt = Ra ? Rb

Clock Cycles: 1

Execution Units: All Integer ALU's

Exceptions: none

Notes:

Rt Bit	Mnem.	Meaning	Test
		Integer Compare Results	
0	EQ	= equal	a = b
1	NE	Not equal	a <> b
2	LTU	< less than	a < b

3	LEU	<= less than or equal	a <= b
4	GEU	>= greater than or equal	a >= b
5	GTU	> greater than	a > b
6			
7			

CMPUI – COMPARE UNSIGNED IMMEDIATE

Description:

Compare two source operands and place the result in the target register. The result is a vector identifying the relationship between the two source operands as signed and unsigned integers.

Operation:

Rt = Ra ? Rb or Rt = Ra ? Imm or Rt = Imm ? Ra

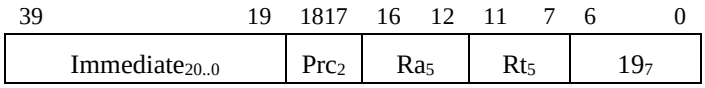
Clock Cycles: 1

Execution Units: All Integer ALU's

Exceptions: none

Notes:

Instruction Format: RI



Rt Bit	Mnem.	Meaning	Test
		Integer Compare Results	
0			
1			
2	LTU	< less than	a < b
3	LEU	<= less than or equal	a <= b
4	GEU	>= greater than or equal	a >= b
5	GTU	> greater than	a > b
6			
7			

SEQ – SET IF EQUAL

Description:

Compare two source operands for equality and if they are equal place the result in the target predicate register. The result is a eight-bit signed immediate value or the value of register Rc. Note that if the source operands are not equal the target register is not affected.

Instruction Format: R3

SEQ Rt, Ra, Rb, imm_6

39 33	32 31	30	29 22	21 17	16 12	11 7	6 0
80_7	Prc_2	~	Imm_8	Rb_5	Ra_5	Rt_5	2_7
80_7	Prc_2	~	Imm_8	Vb_5	Va_5	Vt_5	38_7
80_7	Prc_2	~	Imm_8	Rb_5	Va_5	Vt_5	39_7

SEQ Rt, Ra, Rb, Rc

39 33	32 31	30 27	26 22	21 17	16 12	11 7	6 0
96_7	Prc_2	$\sim_4$	Rc_5	Rb_5	Ra_5	Rt_5	2_7
96_7	Prc_2	$\sim_4$	Vc_5	Vb_5	Va_5	Vt_5	38_7
96_7	Prc_2	$\sim_4$	Vc_5	Rb_5	Va_5	Vt_5	39_7

Operation: R2

$Rt = Ra = Rb \ ? \ Imm_8 : Rt$

Clock Cycles: 1 for scalar, 5 for vector

Execution Units: All Integer ALU's

Exceptions: none

Notes:

SEQI –SET IF EQUAL

Description:

Compare two source operands for equality and place the result in the target predicate register. The result is a Boolean value of one. The first operand is in a register, the second operand is a 16-bit immediate constant.

Instruction Format: R3

SEQ Rt, Ra, imm_{16}

39 35	34 33	32 17	16 12	11 7	6 0
16_5	Prc_2	Imm_{16}	Ra_5	Rt_5	1_7
16_5	Prc_2	Imm_{16}	Va_5	Vt_5	20_7

Operation: R3

Rt = Ra == Imm_{16} ? 1 : Rt

Clock Cycles: 1 for scalar, 10 for vector

Execution Units: All Integer ALU's

Exceptions: none

Notes:

SLE – SET IF LESS THAN OR EQUAL

Description:

Compare two source operands for signed less than or equal and place the result in the target register if the comparison is true. The result is an eight-bit sign extended immediate or the contents of register Rc. This instruction may also test for greater than or equal by swapping operands.

Instruction Format: R3

SLE Rt, Ra, Rb, imm_6

39 33	32 31	30	28 22	21 17	16 12	11 7	6 0
83_7	Prc_2	~	Imm_8	Rb_5	Ra_5	Rt_5	2_7
83_7	Prc_2	~	Imm_8	Vb_5	Va_5	Vt_5	38_7
83_7	Prc_2	~	Imm_8	Rb_5	Va_5	Vt_5	39_7

SLE Rt, Ra, Rb, Rc

39 33	32 31	30 27	26 22	21 17	16 12	11 7	6 0
99_7	Prc_2	$\sim_4$	Rc_5	Rb_5	Ra_5	Rt_5	2_7
99_7	Prc_2	$\sim_4$	Vc_5	Vb_5	Va_5	Vt_5	38_7
99_7	Prc_2	$\sim_4$	Vc_5	Rb_5	Va_5	Vt_5	39_7

Operation:

$Rt = Ra <= Rb \ ? \ Imm_6 : Rt$

Clock Cycles: 1 for scalar, 5 for vector

Execution Units: All Integer ALU's

Exceptions: none

Notes:

SLEI –SET IF LESS THAN OR EQUAL IMMEDIATE

Description:

Compare two source operands for less than or equal and place the result in the target register. The result is a Boolean value of one. The first operand is in a register, the second operand is a 16-bit immediate constant.

Instruction Format: R3

SLE Rt, Ra, imm_{16}

39 35	34 33	32 17	16 12	11 7	6 0
19_5	Prc_2	Imm_{16}	Ra_5	Rt_5	1_7
19_5	Prc_2	Imm_{16}	Va_5	Vt_5	20_7

Operation: R3

Rt = Ra <= Imm_{16} ? 1 : Rt

Clock Cycles: 1 for scalar, 10 for vector

Execution Units: All Integer ALU's

Exceptions: none

Notes:

SLEU – SET IF UNSIGNED LESS THAN OR EQUAL

Description:

Compare two source operands for unsigned less than or equal and place the result in the target register if condition is true. The result is an eight-bit immediate value or the contents of register Rc. This instruction may also test for greater than or equal by swapping operands.

Instruction Format: R3

SLEU Rt, Ra, Rb, imm_6

39 33	32 31	30	29 22	21 17	16 12	11 7	6 0
85_7	Prc_2	~	Imm_8	Rb_5	Ra_5	Rt_5	2_7
85_7	Prc_2	~	Imm_8	Vb_5	Va_5	Vt_5	38_7
85_7	Prc_2	~	Imm_8	Rb_5	Va_5	Vt_5	39_7

SLEU Rt, Ra, Rb, Rc

39 33	32 31	30 27	26 22	21 17	16 12	11 7	6 0
101_7	Prc_2	$~_4$	Rc_5	Rb_5	Ra_5	Rt_5	2_7
101_7	Prc_2	$~_4$	Vc_5	Vb_5	Va_5	Vt_5	38_7
101_7	Prc_2	$~_4$	Vc_5	Rb_5	Va_5	Vt_5	39_7

Operation:

$Rt = Ra <= Rb \ ? \ Imm_8 : Rt$

Clock Cycles: 1, 5 for vector

Execution Units: All Integer ALU's

Exceptions: none

Notes:

SLEUI –SET IF LESS THAN OR EQUAL UNSIGNED IMMEDIATE

Description:

Compare two source operands for unsigned less than or equal and place the result in the target register. The result is a Boolean value of one. The first operand is in a register, the second operand is a 16-bit immediate constant.

Instruction Format: R3

SLE Rt, Ra, imm_{16}

39 35	34 33	32 17	16 12	11 7	6 0
21_5	Prc_2	Imm_{16}	Ra_5	Rt_5	1_7
21_5	Prc_2	Imm_{16}	Va_5	Vt_5	20_7

Operation: R3

Rt = Ra <= Imm_{16} ? 1 : Rt

Clock Cycles: 1 for scalar, 10 for vector

Execution Units: All Integer ALU's

Exceptions: none

Notes:

SLT – SET IF LESS THAN

Description:

Compare two source operands for signed less than and place the result in the target predicate register. If Ra is less than Rb then the result is set to the sign extended immediate value, otherwise the result is set to zero. This instruction may also test for greater than by swapping operands.

Instruction Format: R3

SLT Rt, Ra, Rb, imm_6

39 33	32 31	30	29 22	21 17	16 12	11 7	6 0
82_7	Prc_2	~	Imm_8	Rb_5	Ra_5	Rt_5	2_7
82_7	Prc_2	~	Imm_8	Vb_5	Va_5	Vt_5	38_7
82_7	Prc_2	~	Imm_8	Rb_5	Va_5	Vt_5	39_7

SLT Rt, Ra, Rb, Rc

39 33	32 31	30 27	26 22	21 17	16 12	11 7	6 0
98_7	Prc_2	$\sim_4$	Rc_5	Rb_5	Ra_5	Rt_5	2_7
98_7	Prc_2	$\sim_4$	Vc_5	Vb_5	Va_5	Vt_5	38_7
98_7	Prc_2	$\sim_4$	Vc_5	Rb_5	Va_5	Vt_5	39_7

Assembler Default Format:

The default assembler format places a one or a zero in the target register.

SLT Rt, Ra, Rb

Operation:

Prt = Ra < Rb ? Imm_8 : Rt

Clock Cycles: 1 for scalar, 5 for vector

Execution Units: All Integer ALU's

Exceptions: none

Notes:

SLTI –SET IF LESS THAN IMMEDIATE

Description:

Compare two source operands for less than and place the result in the target register. The result is a Boolean value of one. The first operand is in a register, the second operand is a 16-bit immediate constant.

Instruction Format: R3

SLT Rt, Ra, imm_{16}

39 35	34 33	32 17	16 12	11 7	6 0
18_5	Prc_2	Imm_{16}	Ra_5	Rt_5	1_7
18_5	Prc_2	Imm_{16}	Va_5	Vt_5	20_7

Operation: R3

$Rt = Ra < Imm_{16} ? 1 : Rt$

Clock Cycles: 1 for scalar, 10 for vector

Execution Units: All Integer ALU's

Exceptions: none

Notes:

SLTU – SET IF UNSIGNED LESS THAN

Description:

Compare two source operands for unsigned less than and place the result in the target register. The result is an eight-bit immediate value or the contents of register Rc if the condition is true, otherwise the target register is not affected. This instruction may also test for greater than by swapping operands.

Instruction Format: R3

SLTU Rt, Ra, Rb, imm_6

39 33	32 31	30	29 22	21 17	16 12	11 7	6 0
84_7	Prc_2	~	Imm_8	Rb_5	Ra_5	Rt_5	2_7
84_7	Prc_2	~	Imm_8	Vb_5	Va_5	Vt_5	38_7
84_7	Prc_2	~	Imm_8	Rb_5	Va_5	Vt_5	39_7

SLTU Rt, Ra, Rb, Rc

39 33	32 31	30 27	26 22	21 17	16 12	11 7	6 0
100_7	Prc_2	$~_4$	Rc_5	Rb_5	Ra_5	Rt_5	2_7
100_7	Prc_2	$~_4$	Vc_5	Vb_5	Va_5	Vt_5	38_7
100_7	Prc_2	$~_4$	Vc_5	Rb_5	Va_5	Vt_5	39_7

Operation:

Rt = Ra < Rb ? Imm6 : Rt

Clock Cycles: 1

Execution Units: All Integer ALU's

Exceptions: none

Notes:

SLTUI –SET IF LESS THAN UNSIGNED IMMEDIATE

Description:

Compare two source operands for unsigned less than and place the result in the target register. The result is a Boolean value of one. The first operand is in a register, the second operand is a 16-bit immediate constant.

Instruction Format: R3

SLTU Rt, Ra, imm_{16}

39 35	34 33	32 17	16 12	11 7	6 0
20_5	Prc_2	Imm_{16}	Ra_5	Rt_5	1_7
20_5	Prc_2	Imm_{16}	Va_5	Vt_5	20_7

Operation: R3

$Rt = Ra < Imm_{16} ? 1 : Rt$

Clock Cycles: 1 for scalar, 10 for vector

Execution Units: All Integer ALU's

Exceptions: none

Notes:

SNE – SET IF NOT EQUAL

Description:

Compare two source operands for inequality and place the result in the target register if the comparison is true. The result is an eight-bit immediate or the contents of register Rc. IF the comparison is false the target register is not affected.

Instruction Format: R3

SNE Rt, Ra, Rb, imm_6

39 33	32 31	30	29 22	21 17	16 12	11 7	6 0
81_7	Prc_2	~	Imm_8	Rb_5	Ra_5	Rt_5	2_7
81_7	Prc_2	~	Imm_8	Vb_5	Va_5	Vt_5	38_7
81_7	Prc_2	~	Imm_8	Rb_5	Va_5	Vt_5	39_7

SNE Rt, Ra, Rb, Rc

39 33	32 31	30 27	26 22	21 17	16 12	11 7	6 0
97_7	Prc_2	$\sim_4$	Rc_5	Rb_5	Ra_5	Rt_5	2_7
97_7	Prc_2	$\sim_4$	Vc_5	Vb_5	Va_5	Vt_5	38_7
97_7	Prc_2	$\sim_4$	Vc_5	Rb_5	Va_5	Vt_5	39_7

Operation:

Rt = Ra != Rb ? Imm6 : Rt

Clock Cycles: 1

Execution Units: All Integer ALU's

Exceptions: none

Notes:

SNEQI –SET IF NOT EQUAL

Description:

Compare two source operands for inequality and place the result in the target predicate register. The result is a Boolean value of one. The first operand is in a register, the second operand is a 16-bit immediate constant.

Instruction Format: R3

SEQ Rt, Ra, imm_{16}

39 35	34 33	32 17	16 12	11 7	6 0
17_5	Prc_2	Imm_{16}	Ra_5	Rt_5	1_7
17_5	Prc_2	Imm_{16}	Va_5	Vt_5	20_7

Operation: R3

Rt = Ra != Imm_{16} ? 1 : Rt

Clock Cycles: 1 for scalar, 10 for vector

Execution Units: All Integer ALU's

Exceptions: none

Notes:

ZSEQ – ZERO OR SET IF EQUAL

Description:

Compare two source operands for equality and place the result in the target predicate register. The result is an eight-bit signed immediate value or zero.

Instruction Format: R3

ZSEQ Rt, Ra, Rb, imm_8

39 33	32 31	30	29 22	21 17	16 12	11 7	6 0
112_7	Prc_2	~	Imm_8	Rb_5	Ra_5	Rt_5	2_7
112_7	Prc_2	~	Imm_8	Vb_5	Va_5	Vt_5	38_7
112_7	Prc_2	~	Imm_8	Rb_5	Va_5	Vt_5	39_7

ZSEQ Rt, Ra, Rb, Rc

39 33	32 31	30 27	26 22	21 17	16 12	11 7	6 0
120_7	Prc_2	$~_4$	Rc_5	Rb_5	Ra_5	Rt_5	2_7
120_7	Prc_2	$~_4$	Vc_5	Vb_5	Va_5	Vt_5	38_7
120_7	Prc_2	$~_4$	Vc_5	Rb_5	Va_5	Vt_5	39_7

Operation: R3

$Rt = Ra = Rb \ ? \ Imm_8 : 0$

Clock Cycles: 1 for scalar, 10 for vector

Execution Units: All Integer ALU's

Exceptions: none

Notes:

ZSEQI – ZERO OR SET IF EQUAL

Description:

Compare two source operands for equality and place the result in the target predicate register. The result is a Boolean value of one or zero. The first operand is in a register, the second operand is a 16-bit immediate constant.

Instruction Format: R3

ZSEQ Rt, Ra, imm_{16}

39 35	34 33	32 17	16 12	11 7	6 0
0_5	Prc_2	Imm_{16}	Ra_5	Rt_5	1_7
0_5	Prc_2	Imm_{16}	Va_5	Vt_5	20_7

Operation: R3

Rt = Ra == Imm_{16} ? 1 : 0

Clock Cycles: 1 for scalar, 10 for vector

Execution Units: All Integer ALU's

Exceptions: none

Notes:

ZSGEI – ZERO OR SET IF GREATER THAN OR EQUAL IMMEDIATE

Description:

Compare two source operands for signed greater than or equal and place the result in the target predicate register. The result is a Boolean true or false.

Instruction Format: R3

ZSGT Rt, Ra, imm_{16}

39 35	34 33	32 17	16 12	11 7	6 0
11_5	Prc_2	Imm_{16}	Ra_5	Rt_5	1_7
11_5	Prc_2	Imm_{16}	Va_5	Vt_5	20_7

Operation:

Rt = Ra >= Imm ? 1 : 0

Clock Cycles: 1

Execution Units: All Integer ALU's

Exceptions: none

Notes:

ZSGEUI – ZERO OR SET IF GREATER THAN OR EQUAL UNSIGNED IMMEDIATE

Description:

Compare two source operands for unsigned greater than or equal and place the result in the target predicate register. The result is a Boolean true or false.

Instruction Format: R3

ZSGEU Rt, Ra, imm_{16}

39 35	34 33	32 17	16 12	11 7	6 0
13_5	Prc_2	Imm_{16}	Ra_5	Rt_5	1_7
13_5	Prc_2	Imm_{16}	Va_5	Vt_5	20_7

Operation:

Rt = Ra >= Imm ? 1 : 0

Clock Cycles: 1

Execution Units: All Integer ALU's

Exceptions: none

Notes:

ZSGTI – ZERO OR SET IF GREATER THAN IMMEDIATE

Description:

Compare two source operands for signed greater than and place the result in the target predicate register. The result is a Boolean true or false.

Instruction Format: R3

ZSGT Rt, Ra, imm_{16}

39 35	34 33	32 17	16 12	11 7	6 0
10_5	Prc_2	Imm_{16}	Ra_5	Rt_5	1_7
10_5	Prc_2	Imm_{16}	Va_5	Vt_5	20_7

Operation:

Rt = Ra > Imm ? 1 : 0

Clock Cycles: 1

Execution Units: All Integer ALU's

Exceptions: none

Notes:

ZSGTUI – ZERO OR SET IF GREATER THAN UNSIGNED IMMEDIATE

Description:

Compare two source operands for unsigned greater than and place the result in the target predicate register. The result is a Boolean true or false.

Instruction Format: R3

ZSGTU Rt, Ra, imm_{16}

39 35	34 33	32 17	16 12	11 7	6 0
12_5	Prc_2	Imm_{16}	Ra_5	Rt_5	1_7
12_5	Prc_2	Imm_{16}	Va_5	Vt_5	20_7

Operation:

Rt = Ra > Imm ? 1 : 0

Clock Cycles: 1

Execution Units: All Integer ALU's

Exceptions: none

Notes:

ZSLE – ZERO OR SET IF LESS THAN OR EQUAL

Description:

Compare two source operands for signed less than or equal and place the result in the target register. The result is a six-bit sign extended immediate or the contents of register Rc if the comparison is true, otherwise the target register is set to zero. This instruction may also test for greater than or equal by swapping operands.

Instruction Format: R3

ZSLE Rt, Ra, Rb, imm_6

39 33	32 31	30	29 22	21 17	16 12	11 7	6 0
115_7	Prc_2	~	Imm_8	Rb_5	Ra_5	Rt_5	2_7
115_7	Prc_2	~	Imm_8	Vb_5	Va_5	Vt_5	38_7
115_7	Prc_2	~	Imm_8	Rb_5	Va_5	Vt_5	39_7

ZSLE Rt, Ra, Rb, Rc

39 33	32 31	30 27	26 22	21 17	16 12	11 7	6 0
123_7	Prc_2	$\sim_4$	Rc_5	Rb_5	Ra_5	Rt_5	2_7
123_7	Prc_2	$\sim_4$	Vc_5	Vb_5	Va_5	Vt_5	38_7
123_7	Prc_2	$\sim_4$	Vc_5	Rb_5	Va_5	Vt_5	39_7

Operation:

Rt = Ra <= Rb ? Imm_8 : 0

Clock Cycles: 1 for scalar, 5 for vector

Execution Units: All Integer ALU's

Exceptions: none

Notes:

ZSLEI – ZERO OR SET IF LESS THAN OR EQUAL

Description:

Compare two source operands for signed less than or equal and place the result in the target register. The result is a Boolean value of one or zero. The first operand is in a register, the second operand is a 16-bit immediate constant.

Instruction Format: R3

ZSLE Rt, Ra, imm_{16}

39 35	34 33	32 17	16 12	11 7	6 0
3_5	Prc_2	Imm_{16}	Ra_5	Rt_5	1_7
3_5	Prc_2	Imm_{16}	Va_5	Vt_5	20_7

Operation:

$Rt = Ra <= Imm_{16} ? 1 : 0$

Clock Cycles: 1 for scalar, 5 for vector

Execution Units: All Integer ALU's

Exceptions: none

Notes:

ZSLEU – ZERO OR SET IF UNSIGNED LESS THAN OR EQUAL

Description:

Compare two source operands for unsigned less than or equal and place the result in the target register. The result is an eight-bit immediate value or the contents of register Rc if the condition is true, otherwise the target register is set to zero. This instruction may also test for greater than or equal by swapping operands.

Instruction Format: R3

ZSLEU Rt, Ra, Rb, imm_6

39 33	32 31	30	29 22	21 17	16 12	11 7	6 0
117_7	Prc_2	~	Imm_8	Rb_5	Ra_5	Rt_5	2_7
117_7	Prc_2	~	Imm_8	Vb_5	Va_5	Vt_5	38_7
117_7	Prc_2	~	Imm_8	Rb_5	Va_5	Vt_5	39_7

ZSLEU Rt, Ra, Rb, Rc

39 33	32 31	30 27	26 22	21 17	16 12	11 7	6 0
125_7	Prc_2	$~_4$	Rc_5	Rb_5	Ra_5	Rt_5	2_7
125_7	Prc_2	$~_4$	Vc_5	Vb_5	Va_5	Vt_5	38_7
125_7	Prc_2	$~_4$	Vc_5	Rb_5	Va_5	Vt_5	39_7

Operation:

Rt = Ra <= Rb ? Imm8 : 0

Clock Cycles: 1

Execution Units: All Integer ALU's

Exceptions: none

Notes:

ZSLEUI – ZERO OR SET IF UNSIGNED LESS THAN OR EQUAL

Description:

Compare two source operands for signed less than or equal and place the result in the target register. The result is a Boolean value of one or zero. The first operand is in a register, the second operand is a 16-bit immediate constant.

Instruction Format: R3

ZSLE Rt, Ra, imm_{16}

39 35	34 33	32 17	16 12	11 7	6 0
5_5	Prc_2	Imm_{16}	Ra_5	Rt_5	1_7
5_5	Prc_2	Imm_{16}	Va_5	Vt_5	20_7

Operation:

$Rt = Ra <= Imm_{16} ? 1 : 0$

Clock Cycles: 1 for scalar, 5 for vector

Execution Units: All Integer ALU's

Exceptions: none

Notes:

ZSLT – ZERO OR SET IF LESS THAN

Description:

Compare two source operands for signed less than and place the result in the target predicate register. If Ra is less than Rb then the result is set to the sign extended immediate value or the contents of register Rc, otherwise the result is set to zero. This instruction may also test for greater than by swapping operands.

Instruction Format: R3

ZSLT Rt, Ra, Rb, imm_8

39 33	32 31	30	29 22	21 17	16 12	11 7	6 0
114_7	Prc_2	~	Imm_8	Rb_5	Ra_5	Rt_5	2_7
114_7	Prc_2	~	Imm_8	Vb_5	Va_5	Vt_5	38_7
114_7	Prc_2	~	Imm_8	Rb_5	Va_5	Vt_5	39_7

ZSLT Rt, Ra, Rb, Rc

39 33	32 31	30 27	26 22	21 17	16 12	11 7	6 0
122_7	Prc_2	$~_4$	Rc_5	Rb_5	Ra_5	Rt_5	2_7
122_7	Prc_2	$~_4$	Vc_5	Vb_5	Va_5	Vt_5	38_7
122_7	Prc_2	$~_4$	Vc_5	Rb_5	Va_5	Vt_5	39_7

Assembler Default Format:

The default assembler format places a one or a zero in the target register.

ZSLT Rt, Ra, Rb

Operation:

Prt = Ra < Rb ? Imm_8 : 0

Clock Cycles: 1 for scalar, 5 for vector

Execution Units: All Integer ALU's

Exceptions: none

Notes:

ZSLTI – ZERO OR SET IF LESS THAN IMMEDIATE

Description:

Compare two source operands for signed less than and place the result in the target predicate register. The result is a Boolean true or false.

Instruction Format: R3

ZSLTI Rt, Ra, imm_{16}

39 35	34 33	32 17	16 12	11 7	6 0
2_5	Prc_2	Imm_{16}	Ra_5	Rt_5	1_7
2_5	Prc_2	Imm_{16}	Va_5	Vt_5	20_7

Operation:

Rt = Ra < Imm ? 1 : 0

Clock Cycles: 1

Execution Units: All Integer ALU's

Exceptions: none

Notes:

ZSLTU – ZERO OR SET IF UNSIGNED LESS THAN

Description:

Compare two source operands for unsigned less than and place the result in the target register. The result is an eight-bit immediate value or the contents of register Rc if the condition is true, otherwise the target register is set to zero. This instruction may also test for greater than by swapping operands.

Instruction Format: R3

ZSLTU Rt, Ra, Rb, imm_8

39 33	32 31	30	29 22	21 17	16 12	11 7	6 0
116_7	Prc_2	~	Imm_8	Rb_5	Ra_5	Rt_5	2_7
116_7	Prc_2	~	Imm_8	Vb_5	Va_5	Vt_5	38_7
116_7	Prc_2	~	Imm_8	Rb_5	Va_5	Vt_5	39_7

ZSLTU Rt, Ra, Rb, Rc

39 33	32 31	30 27	26 22	21 17	16 12	11 7	6 0
124_7	Prc_2	$~_4$	Rc_5	Rb_5	Ra_5	Rt_5	2_7
124_7	Prc_2	$~_4$	Vc_5	Vb_5	Va_5	Vt_5	38_7
124_7	Prc_2	$~_4$	Vc_5	Rb_5	Va_5	Vt_5	39_7

Operation:

Rt = Ra < Rb ? Imm_8 : 0

Clock Cycles: 1

Execution Units: All Integer ALU's

Exceptions: none

Notes:

ZSLTUI – ZERO OR SET IF UNSIGNED LESS THAN

Description:

Compare two source operands for signed less than and place the result in the target register. The result is a Boolean value of one or zero. The first operand is in a register, the second operand is a 16-bit immediate constant.

Instruction Format: R3

ZSLE Rt, Ra, imm_{16}

39 35	34 33	32 17	16 12	11 7	6 0
4_5	Prc_2	Imm_{16}	Ra_5	Rt_5	1_7
4_5	Prc_2	Imm_{16}	Va_5	Vt_5	20_7

Operation:

$Rt = Ra <= Imm_{16} ? 1 : 0$

Clock Cycles: 1 for scalar, 5 for vector

Execution Units: All Integer ALU's

Exceptions: none

Notes:

ZSNE – ZERO OR SET IF NOT EQUAL

Description:

Compare two source operands for inequality and place the result in the target register if the comparison is true. The result is an eight-bit immediate or the contents of register Rc. IF the comparison is false the target register is set to zero.

Instruction Format: R3

ZSNE Rt, Ra, Rb, imm_8

39 33	32 31	30	29 22	21 17	16 12	11 7	6 0
113_7	Prc_2	~	Imm_8	Rb_5	Ra_5	Rt_5	2_7
113_7	Prc_2	~	Imm_8	Vb_5	Va_5	Vt_5	38_7
113_7	Prc_2	~	Imm_8	Rb_5	Va_5	Vt_5	39_7

ZSNE Rt, Ra, Rb, Rc

39 33	32 31	30 27	26 22	21 17	16 12	11 7	6 0
121_7	Prc_2	$~_4$	Rc_5	Rb_5	Ra_5	Rt_5	2_7
121_7	Prc_2	$~_4$	Vc_5	Vb_5	Va_5	Vt_5	38_7
121_7	Prc_2	$~_4$	Vc_5	Rb_5	Va_5	Vt_5	39_7

Operation:

Rt = Ra != Rb ? Imm_8 : 0

Clock Cycles: 1

Execution Units: All Integer ALU's

Exceptions: none

Notes:

ZSNEI – ZERO OR SET IF NOT EQUAL

Description:

Compare two source operands for inequality and place the result in the target predicate register. The result is a Boolean value of one or zero. The first operand is in a register, the second operand is a 16-bit immediate constant.

Instruction Format: R3

ZSNE Rt, Ra, imm_{16}

39 35	34 33	32 17	16 12	11 7	6 0
1_5	Prc_2	Imm_{16}	Ra_5	Rt_5	1_7
1_5	Prc_2	Imm_{16}	Va_5	Vt_5	20_7

Operation: R3

Rt = Ra != Imm_{16} ? 1 : 0

Clock Cycles: 1 for scalar, 10 for vector

Execution Units: All Integer ALU's

Exceptions: none

Notes:

SHIFT AND ROTATE OPERATIONS

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 and Thor use 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.

Thor2024 has a full complement of shift instructions including rotates.

ASL –ARITHMETIC SHIFT LEFT

Description:

This is an alternate mnemonic for the ALSP (pair shift) instruction. Left shift an operand value by an operand value and place the upper bits of the result in the target register. The 'C' field of the instruction indicates to complement the Ra operand, which is zero. The first operand must be in a register specified by the Rb. The second operand may be either a register specified by the Rc field of the instruction, or an immediate value.

Instruction Format: SHIFT

39 36	3534	33	32	31 27	26 22	21 16	16 12	11 7	6 0
0_4	Prc_2	C	0	~$_5$	Rc_5	Rb_5	0_5	Rt_5	88_7
0_4	Prc_2	C	0	~$_5$	Vc_5	Vb_5	0_5	Vt_5	90_7

Operation:

Rt = {Rb,Ra} << Rc

Operation Size: .o

Execution Units: integer ALU

Exceptions: none

Example:

ASLP –ARITHMETIC SHIFT LEFT PAIR

Description:

Left shift a pair of operand values by an operand value and place the upper bits of the result in the target register. The ‘C’ field of the instruction indicates to invert the Ra operand of the pair while shifting. The pair of registers shifted is specified by Ra (lower bits), Rb (upper bits). The third operand may be either a register specified by the Rc field of the instruction, or an immediate value.

This instruction may be used to perform a rotate operation by specifying the same register for Ra and Rb. It may also be used to implement a ring counter by inverting Ra during the shift.

Instruction Format: SHIFT

39 36	3534	33	32	31 27	26 22	21 16	16 12	11 7	6 0
0_4	Prc_2	C	0	$\sim_5$	Rc_5	Rb_5	Ra_5	Rt_5	88_7
0_4	Prc_2	C	0	$\sim_5$	Vc_5	Vb_5	Va_5	Vt_5	90_7

Operation:

Rt = {Rb,Ra} << Rc

Operation Size: .o

Execution Units: integer ALU

Exceptions: none

Example:

ASLI –ARITHMETIC SHIFT LEFT

Description:

Left shift an operand value by an operand value and place the result in the target register. The first operand must be in a register specified by the Rb. The second operand is an immediate value.

Instruction Format: SHIFT

39 36	3534	33	32	31 28	27 22	21 16	16 12	11 7	6 0
0_4	Prc_2	C	1	$\sim_4$	Imm_6	Rb_5	0_5	Rt_5	88_7
0_4	Prc_2	C	1	$\sim_4$	Imm_6	Vb_5	0_5	Vt_5	90_7

Operation:

Rt = {Rb,Ra} << Imm

Operation Size: .o

Execution Units: integer ALU

Exceptions: none

Example:

ASLPI –ARITHMETIC SHIFT LEFT PAIR BY IMMEDIATE

Description:

Left shift a pair of operand values by an operand value and place the result in the target register. The ‘C’ field of the instruction indicates to invert the Ra operand. The operand pair must be in registers Rb (upper bits) and Ra (lower bits). The third operand is an immediate value.

Instruction Format: SHIFT

39 36	3534	33	32	31 28	27 22	21 16	16 12	11 7	6 0
0_4	Prc_2	C	1	$\sim_4$	Imm_6	Rb_5	Ra_5	Rt_5	88_7
0_4	Prc_2	C	1	$\sim_4$	Imm_6	Vb_5	Va_5	Vt_5	90_7

Operation:

Rt = {Rb,Ra} << Imm

Operation Size: .o

Execution Units: integer ALU

Exceptions: none

Example:

ASR –ARITHMETIC SHIFT RIGHT

Description:

Right shift an operand value by an operand value and place the result in the target register. The sign bit is shifted into the most significant bits. The first operand must be in a register specified by the Ra. The second operand may be either a register specified by the Rc field of the instruction, or an immediate value.

Instruction Format: SHIFT

39 36	3534	33	32	3130	29 27	26 22	21 16	16 12	11 7	6 0
2_4	Prc_2	C	0	Rm_2	~$_3$	Rc_5	~$_5$	Ra_5	Rt_5	88_7
2_4	Prc_2	C	0	Rm_2	~$_3$	Vc_5	~$_5$	Va_5	Vt_5	90_7

Rm_2		
0	Truncate	Discards bits
1	Round towards zero	If result is negative, then it is rounded up
2	Round up	If there was a carry out of the LSB, add one

Operation:

Rt = round(Ra >> Rc)

Operation Size: .o

Execution Units: integer ALU

Exceptions: none

Example:

ASRI –ARITHMETIC SHIFT RIGHT

Description:

Right shift an operand value by an operand value and place the result in the target register. The sign bit is shifted into the most significant bits. The first operand must be in a register specified by the Ra. The second operand is an immediate value.

Instruction Format: SHIFT

39 36	3534	33	32	3130	29 28	27 22	21 16	16 12	11 7	6 0
2_4	Prc_2	C	1	Rm_2	$\sim_2$	Imm_6	$\sim_5$	Ra_5	Rt_5	88_7
2_4	Prc_2	C	1	Rm_2	$\sim_2$	Imm_6	$\sim_5$	Va_5	Vt_5	90_7

Rm_2		
0	Truncate	Discards bits
1	Round towards zero	If result is negative, then it is rounded up
2	Round up	If there was a carry out of the LSB, add one

Operation:

Rt = round(Ra >> Imm)

Operation Size: .o

Execution Units: integer ALU

Exceptions: none

Example:

LSR –LOGIC SHIFT RIGHT

Description:

This is an alternate mnemonic for the LSRP instruction where Rb is assumed to be r0. Right shift an operand value by an operand value and place the result in the target register. The ‘C’ field of indicates to shift a zero or a one into the most significant bits. The first operand must be in a register specified by the Ra. The second operand may be either a register specified by the Rc field of the instruction, or an immediate value.

Instruction Format: SHIFT

39 36	3534	33	32	31 27	26 22	21 16	16 12	11 7	6 0
1_4	Prc_2	C	0	$\sim_5$	Rc_5	0_5	Ra_5	Rt_5	88_7
1_4	Prc_2	C	0	$\sim_5$	Vc_5	0_5	Va_5	Vt_5	90_7

Operation:

Rt = Ra >> Rc

Operation Size: .o

Execution Units: integer ALU

Exceptions: none

Example:

LSRP –LOGIC SHIFT RIGHT PAIR

Description:

Right shift a pair of operand values by an operand value and place the lower bits of the result in the target register. The ‘C’ field of the instruction indicates to complement the Rb register during the shift. The pair of operands are specified by Ra and Rb. The third operand may be either a register specified by the Rc field of the instruction, or an immediate value.

This instruction may be used to perform a right rotate operation by specifying the same register for Ra and Rb.

Instruction Format: SHIFT

39 36	3534	33	32	31 27	26 22	21 16	16 12	11 7	6 0
1_4	Prc_2	C	0	$\sim_5$	Rc_5	Rb_5	Ra_5	Rt_5	88_7
1_4	Prc_2	C	0	$\sim_5$	Vc_5	Vb_5	Va_5	Vt_5	90_7

Operation:

Rt = {Rb,Ra} >> Rc

Operation Size: .o

Execution Units: integer ALU

Exceptions: none

Example:

LSRI –LOGICAL SHIFT RIGHT

Description:

Right shift an operand value by an operand value and place the result in the target register. The ‘C’ field of the instruction indicates to shift a zero or a one into the most significant bits. The first operand must be in a register specified by the Ra. The second operand is an immediate value.

Instruction Format: SHIFT

39 36	3534	33	32	31 28	27 22	21 16	16 12	11 7	6 0
1_4	Prc_2	C	1	$\sim_4$	Imm_6	0_5	Ra_5	Rt_5	88_7
1_4	Prc_2	C	1	$\sim_4$	Imm_6	0_5	Va_5	Vt_5	90_7

Operation:

Rt = Ra >> Imm

Operation Size: .o

Execution Units: integer ALU

Exceptions: none

Example:

LSRPI –LOGICAL SHIFT RIGHT PAIR BY IMMEDIATE

Description:

Right shift a pair of operand values by an operand value and place the lower bits of the result in the target register. The ‘C’ field of the instruction indicates to complement the Rb operand during the shift. The operand pair must be in a register specified by Ra and Rb. The third operand is an immediate value.

This instruction may be used to perform a right rotate operation.

Instruction Format: SHIFT

39 36	3534	33	32	31 28	27 22	21 16	16 12	11 7	6 0
1_4	Prc_2	C	1	$\sim_4$	Imm_6	Rb_5	Ra_5	Rt_5	88_7
1_4	Prc_2	C	1	$\sim_4$	Imm_6	Vb_5	Va_5	Vt_5	90_7

Operation:

Rt = {Rb,Ra} >> Imm

Operation Size: .o

Execution Units: integer ALU

Exceptions: none

Example:

ROL –ROTATE LEFT

Description:

This is an alternate mnemonic for the ASLP instruction. Rotate left an operand value by an operand value and place the result in the target register. The most significant bits are shifted into the least significant bits. The first operand must be in a register specified by Ra and Rb. The second operand may be either a register specified by the Rc field of the instruction, or an immediate value.

Instruction Format: SHIFT

39 36	3534	33	32	31 27	26 22	21 16	16 12	11 7	6 0
0_4	Prc_2	C	0	$\sim_5$	Rc_5	Rb_5	Ra_5	Rt_5	88_7
0_4	Prc_2	C	0	$\sim_5$	Vc_5	Vb_5	Va_5	Vt_5	90_7

Operation:

Rt = {Rb,Ra} << Rc

Operation Size: .o

Execution Units: integer ALU

Exceptions: none

Example:

ROLI –ROTATE LEFT BY IMMEDIATE

Description:

Rotate left shift an operand value by an operand value and place the result in the target register. The most significant bits are shifted into the least significant bits. The first operand must be in a register specified by the Ra. The second operand is an immediate value.

Instruction Format: SHIFT

39 36	3534	33	32	31 28	27 22	21 16	16 12	11 7	6 0
0_4	Prc_2	C	1	$\sim_4$	Imm_6	Rb_5	Ra_5	Rt_5	88_7
0_4	Prc_2	C	1	$\sim_4$	Imm_6	Vb_5	Va_5	Vt_5	90_7

Operation:

Rt = {Rb,Ra} << Imm

Operation Size: .o

Execution Units: integer ALU

Exceptions: none

Example:

ROR –ROTATE RIGHT

Description:

Rotate right an operand value by an operand value and place the result in the target register. The least significant bits are shifted into the most significant bits. The first operand must be in a register specified by Ra and Rb. The second operand may be either a register specified by the Rc field of the instruction, or an immediate value.

Instruction Format: SHIFT

39 36	3534	33	32	31 27	26 22	21 16	16 12	11 7	6 0
1_4	Prc_2	C	0	$\sim_5$	Rc_5	Rb_5	Ra_5	Rt_5	88_7
1_4	Prc_2	C	0	$\sim_5$	Vc_5	Vb_5	Va_5	Vt_5	90_7

Operation:

Rt = {Rb,Ra} >> Rc

Operation Size: .o

Execution Units: integer ALU

Exceptions: none

Example:

RORI –ROTATE RIGHT BY IMMEDIATE

Description:

Rotate right an operand value by an operand value and place the result in the target register. The least significant bits are shifted into the most significant bits. The first operand must be in a register specified by Ra and Rb. The second operand is an immediate value.

Instruction Format: SHIFT

39 36	3534	33	32	31 28	27 22	21 16	16 12	11 7	6 0
1_4	Prc_2	C	1	$\sim_4$	Imm_6	Rb_5	Ra_5	Rt_5	88_7
1_4	Prc_2	C	1	$\sim_4$	Imm_6	Vb_5	Va_5	Vt_5	90_7

Operation:

Rt = {Rb,Ra} >> Imm

Operation Size: .o

Execution Units: integer ALU

Exceptions: none

Example:

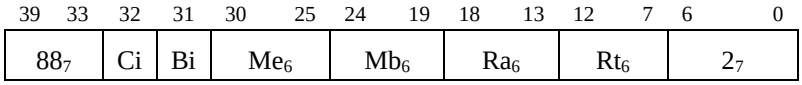
BIT-FIELD MANIPULATION OPERATIONS

Many CPUs do not have direct support for bit-field manipulation. Instead, they rely on ordinary logical and shift operations. The benefit of having bit-field operations is that they are more code dense then performing the operations using other ALU ops.

The beginning and end of a bitfield may be specified as either a pair of immediate constants or in a pair of registers.

GENERAL FORMAT OF BITFIELD INSTRUCTIONS

CLR Rt, Ra, Rb, Rc



Mb_6 may be either a register spec or a six-bit unsigned immediate constant specifying the start position of the bitfield.

Me_6 may be either a register spec or a six-bit unsigned immediate constant specifying the end position of the bitfield.

The Ci field indicates (1) to use either an immediate constant, or (0) to use a register for the third source operand.

The Bi field indicates (1) to use either an immediate constant, or (0) to use a register for the second source operand.

CLR – CLEAR BIT FIELD

Description:

A bit field in the source operand is cleared and the result placed in the target register. Rb specifies the first bit of the bitfield, Rc specifies the last bit of the bitfield. Immediate constants may be substituted for Rb and Rc.

Instruction Format: R3

CLR Rt, Ra, Rb, Rc

39 33	3231	30 29	28	27	26 22	21 17	16 12	11 7	6 0
88_7	Prc_2	0	~	~	Rc_5	Rb_5	Ra_5	Rt_5	2_7
88_7	Prc_2	1	I	~	Rc_5	$Uimm_5$	Ra_5	Rt_5	2_7
88_7	Prc_2	2	~	$Uimm_6$		Rb_5	Ra_5	Rt_5	2_7
88_7	Prc_2	3	I	$Uimm_6$		$Uimm_5$	Ra_5	Rt_5	2_7
88_7	Prc_2	0	~	~	Vc_5	Vb_5	Va_5	Vt_5	38_7
88_7	Prc_2	1	I	~	Vc_5	$Uimm_5$	Va_5	Vt_5	38_7
88_7	Prc_2	2	~	$Uimm_6$		Vb_5	Va_5	Vt_5	38_7
88_7	Prc_2	3	I	$Uimm_6$		$Uimm_5$	Va_5	Vt_5	38_7

Operation:

Rt = Ra

Rt[ME:MB] = 0

Clock Cycles:

Execution Units: All Integer ALU's

Exceptions: none

Notes:

COM – COMPLEMENT BIT FIELD

Description:

A bit field in the source operand is one's complemented and the result placed in the target register. Rb specifies the first bit of the bitfield, Rc specifies the last bit of the bitfield. Immediate constants may be substituted for Rb and Rc.

Operation:

Instruction Format: BITFLD

COM Rt, Ra, Rb, Rc

39 33	3231	30 29	28	27	26 22	21 17	16 12	11 7	6 0
93_7	Prc_2	0	~	~	Rc_5	Rb_5	Ra_5	Rt_5	2_7
93_7	Prc_2	1	I	~	Rc_5	$Uimm_5$	Ra_5	Rt_5	2_7
93_7	Prc_2	2	~	$Uimm_6$		Rb_5	Ra_5	Rt_5	2_7
93_7	Prc_2	3	I	$Uimm_6$		$Uimm_5$	Ra_5	Rt_5	2_7
93_7	Prc_2	0	~	~	Vc_5	Vb_5	Va_5	Vt_5	38_7
93_7	Prc_2	1	I	~	Vc_5	$Uimm_5$	Va_5	Vt_5	38_7
93_7	Prc_2	2	~	$Uimm_6$		Vb_5	Va_5	Vt_5	38_7
93_7	Prc_2	3	I	$Uimm_6$		$Uimm_5$	Va_5	Vt_5	38_7

Clock Cycles:

Execution Units: All Integer ALU's

Exceptions: none

Notes:

DEP – DEPOSIT BIT FIELD

Description:

A source operand is transferred to a bitfield in the target register. Rb specifies the first bit of the bitfield, Rc specifies the last bit of the bitfield. Immediate constants may be substituted for Rb and Rc.

Instruction Formats:

DEP Rt, Ra, Rb, Rc

39 33	3231	30 29	28	27	26 22	21 17	16 12	11 7	6 0
92_7	Prc_2	0	~	~	Rc_5	Rb_5	Ra_5	Rt_5	2_7
92_7	Prc_2	1	I	~	Rc_5	$Uimm_5$	Ra_5	Rt_5	2_7
92_7	Prc_2	2	~	$Uimm_6$		Rb_5	Ra_5	Rt_5	2_7
92_7	Prc_2	3	I	$Uimm_6$		$Uimm_5$	Ra_5	Rt_5	2_7
92_7	Prc_2	0	~	~	Vc_5	Vb_5	Va_5	Vt_5	38_7
92_7	Prc_2	1	I	~	Vc_5	$Uimm_5$	Va_5	Vt_5	38_7
92_7	Prc_2	2	~	$Uimm_6$		Vb_5	Va_5	Vt_5	38_7
92_7	Prc_2	3	I	$Uimm_6$		$Uimm_5$	Va_5	Vt_5	38_7

Operation:

MB = Rb or Imm

ME = Rc or Imm

Rt[ME:MB] = Ra

Clock Cycles: 1

Execution Units: All Integer ALU's

Exceptions: none

Notes:

EXT – EXTRACT BIT FIELD

Description:

A bit field is extracted from the source operand, sign extended, and the result placed in the target register. Rb specifies the first bit of the bitfield, Rc specifies the last bit of the bitfield. Immediate constants may be substituted for Rb and Rc.

Instruction Format: BITFLD

EXT Rt, Ra, Rb, Rc

Instruction Format: R3

39 33	3231	30 29	28	27	26 22	21 17	16 12	11 7	6 0
90_7	Prc_2	0	~	~	Rc_5	Rb_5	Ra_5	Rt_5	2_7
90_7	Prc_2	1	I	~	Rc_5	$Uimm_5$	Ra_5	Rt_5	2_7
90_7	Prc_2	2	~	$Uimm_6$		Rb_5	Ra_5	Rt_5	2_7
90_7	Prc_2	3	I	$Uimm_6$		$Uimm_5$	Ra_5	Rt_5	2_7
90_7	Prc_2	0	~	~	Vc_5	Vb_5	Va_5	Vt_5	38_7
90_7	Prc_2	1	I	~	Vc_5	$Uimm_5$	Va_5	Vt_5	38_7
90_7	Prc_2	2	~	$Uimm_6$		Vb_5	Va_5	Vt_5	38_7
90_7	Prc_2	3	I	$Uimm_6$		$Uimm_5$	Va_5	Vt_5	38_7

Operation:

Rt = sign extend(Ra[ME:MB])

Clock Cycles:

Execution Units: All Integer ALU's

Exceptions: none

Notes:

EXTU – EXTRACT UNSIGNED BIT FIELD

Description:

A bit field is extracted from the source operand, zero extended, and the result placed in the target register. Rb specifies the first bit of the bitfield, Rc specifies the last bit of the bitfield. Immediate constants may be substituted for Rb and Rc.

Instruction Format: BITFLD

EXTU Rt, Ra, Rb, Rc

Instruction Format: R3

39 33	3231	30 29	28	27	26 22	21 17	16 12	11 7	6 0
91_7	Prc_2	0	~	~	Rc_5	Rb_5	Ra_5	Rt_5	2_7
91_7	Prc_2	1	I	~	Rc_5	$Uimm_5$	Ra_5	Rt_5	2_7
91_7	Prc_2	2	~	$Uimm_6$		Rb_5	Ra_5	Rt_5	2_7
91_7	Prc_2	3	I	$Uimm_6$		$Uimm_5$	Ra_5	Rt_5	2_7
91_7	Prc_2	0	~	~	Vc_5	Vb_5	Va_5	Vt_5	38_7
91_7	Prc_2	1	I	~	Vc_5	$Uimm_5$	Va_5	Vt_5	38_7
91_7	Prc_2	2	~	$Uimm_6$		Vb_5	Va_5	Vt_5	38_7
91_7	Prc_2	3	I	$Uimm_6$		$Uimm_5$	Va_5	Vt_5	38_7

Operation:

Rt = zero extend(Ra[ME:MB])

Clock Cycles:

Execution Units: All Integer ALU's

Exceptions: none

Notes:

SET – SET BIT FIELD

Description:

A bit field in the source operand is set to all ones and the result placed in the target register. Rb specifies the first bit of the bitfield, Rc specifies the last bit of the bitfield. Immediate constants may be substituted for Rb and Rc.

Instruction Format: BITFLD

SET Rt, Ra, Rb, Rc

39 33	3231	30 29	28	27	26 22	21 17	16 12	11 7	6 0
89_7	Prc_2	0	~	~	Rc_5	Rb_5	Ra_5	Rt_5	2_7
89_7	Prc_2	1	I	~	Rc_5	$Uimm_5$	Ra_5	Rt_5	2_7
89_7	Prc_2	2	~	$Uimm_6$		Rb_5	Ra_5	Rt_5	2_7
89_7	Prc_2	3	I	$Uimm_6$		$Uimm_5$	Ra_5	Rt_5	2_7
89_7	Prc_2	0	~	~	Vc_5	Vb_5	Va_5	Vt_5	38_7
89_7	Prc_2	1	I	~	Vc_5	$Uimm_5$	Va_5	Vt_5	38_7
89_7	Prc_2	2	~	$Uimm_6$		Vb_5	Va_5	Vt_5	38_7
89_7	Prc_2	3	I	$Uimm_6$		$Uimm_5$	Va_5	Vt_5	38_7

Operation:

Rt = Ra

Rt[ME:MB] = 111…

Clock Cycles:

Execution Units: All Integer ALU's

Exceptions: none

Notes:

AES64DS – FINAL ROUND DECRYPTION

Description:

Perform the final round of decryption for the AES standard. Register Ra represents the entire AES state.

Instruction Format: R3

39 33	33	3231	30 27	26 22	21 17	16 12	11 7	6 0
26_7	~	2_2	$\sim_4$	18_5	$\sim_5$	Ra_5	Rt_5	2_7
26_7	~	2_2	$\sim_4$	18_5	$\sim_5$	Va_5	Vt_5	38_7

Operation:

Exceptions: none

AES64DSM – MIDDLE ROUND DECRYPTION

Description:

Perform a middle round of decryption for the AES standard. Register Ra represents the entire AES state.

Instruction Format: R3

39 33	33	3231	30 27	26 22	21 17	16 12	11 7	6 0
26_7	~	2_2	$\sim_4$	19_5	$\sim_5$	Ra_5	Rt_5	2_7
26_7	~	2_2	$\sim_4$	19_5	$\sim_5$	Va_5	Vt_5	38_7

Operation:

Exceptions: none

AES64ES – FINAL ROUND ENCRYPTION

Description:

Perform the final round of encryption for the AES standard. Register Ra represents the entire AES state.

Instruction Format: R3

39 33	33	3231	30 27	26 22	21 17	16 12	11 7	6 0
26_7	~	2_2	$\sim_4$	20_5	$\sim_5$	Ra_5	Rt_5	2_7
26_7	~	2_2	$\sim_4$	20_5	$\sim_5$	Va_5	Vt_5	38_7

Operation:

Exceptions: none

AES64ESM – MIDDLE ROUND ENCRYPTION

Description:

Perform a middle round of encryption for the AES standard. Register Ra represents the entire AES state.

Instruction Format: R3

39 33	33	3231	30 27	26 22	21 17	16 12	11 7	6 0
26_7	~	2_2	$\sim_4$	21_5	$\sim_5$	Ra_5	Rt_5	2_7
26_7	~	2_2	$\sim_4$	21_5	$\sim_5$	Va_5	Vt_5	38_7

Operation:

Exceptions: none

SHA256SIG0

Description:

Implements the Sigma0 transformation function used in the SHA2-256 and SHA2-224 hash function. Only the low order 32 bits of Ra are operated on. The 32-bit result is sign extended to the machine width.

Instruction Format: R3

39 33	33	3231	30 27	26 22	21 17	16 12	11 7	6 0
26_7	~	2_2	$\sim_4$	24_5	$\sim_5$	Ra_5	Rt_5	2_7
26_7	~	2_2	$\sim_4$	24_5	$\sim_5$	Va_5	Vt_5	38_7

Operation:

Rt = sign extend(ror32(Ra,7) ^ ror32(Ra,18) ^ (Ra_{32} >> 3))

Execution Units: ALU #0

Exceptions: none

SHA256SIG1

Description:

Implements the Sigma1 transformation function used in the SHA2-256 and SHA2-224 hash function. Only the low order 32 bits of Ra are operated on. The 32-bit result is sign extended to the machine width.

Instruction Format: R3

39 33	33	3231	30 27	26 22	21 17	16 12	11 7	6 0
26_7	~	2_2	$\sim_4$	25_5	$\sim_5$	Ra_5	Rt_5	2_7
26_7	~	2_2	$\sim_4$	25_5	$\sim_5$	Va_5	Vt_5	38_7

Clock Cycles: 1

Operation:

Rt = sign extend(ror32(Ra,17) ^ ror32(Ra,19) ^ (Ra_{32} >> 10))

Execution Units: ALU #0

Exceptions: none

SHA256SUM0

Description:

Implements the Sum0 transformation function used in the SHA2-256 and SHA2-224 hash function. Only the low order 32 bits of Ra are operated on. The 32-bit result is sign extended to the machine width.

Instruction Format: R3

39 33	33	3231	30 27	26 22	21 17	16 12	11 7	6 0
26_7	~	2_2	$\sim_4$	26_5	$\sim_5$	Ra_5	Rt_5	2_7
26_7	~	2_2	$\sim_4$	26_5	$\sim_5$	Va_5	Vt_5	38_7

Operation:

Rt = sign extend(ror32(Ra,2) ^ ror32(Ra,13) ^ ror32(Ra, 22))

Execution Units: ALU #0

Exceptions: none

SHA256SUM1

Description:

Implements the Sum1 transformation function used in the SHA2-256 and SHA2-224 hash function. Only the low order 32 bits of Ra are operated on. The 32-bit result is sign extended to the machine width.

Instruction Format: R3

39 33	33	3231	30 27	26 22	21 17	16 12	11 7	6 0
26_7	~	2_2	$\sim_4$	27_5	$\sim_5$	Ra_5	Rt_5	2_7
26_7	~	2_2	$\sim_4$	27_5	$\sim_5$	Va_5	Vt_5	38_7

Operation:

Rt = sign extend(ror32(Ra,6) ^ ror32(Ra,11) ^ ror32(Ra, 25))

Execution Units: ALU #0

Exceptions: none

SHA512SIG0

Description:

Implements the Sigma0 transformation function used in the SHA2-512 hash function.

Instruction Format: R3

39 33	33	3231	30 27	26 22	21 17	16 12	11 7	6 0
26_7	~	2_2	$\sim_4$	28_5	$\sim_5$	Ra_5	Rt_5	2_7
26_7	~	2_2	$\sim_4$	28_5	$\sim_5$	Va_5	Vt_5	38_7

Operation:

Rt = ror64(Ra,1) ^ ror64(Ra, 8) ^ (Ra >> 7)

Execution Units: ALU #0

Exceptions: none

SHA512SIG1

Description:

Implements the Sigma1 transformation function used in the SHA2-512 hash function.

Instruction Format: R3

39 33	33	3231	30 27	26 22	21 17	16 12	11 7	6 0
26_7	~	2_2	$\sim_4$	29_5	$\sim_5$	Ra_5	Rt_5	2_7
26_7	~	2_2	$\sim_4$	29_5	$\sim_5$	Va_5	Vt_5	38_7

Operation:

Rt = ror64(Ra,19) ^ ror64(Ra, 61) ^ (Ra >> 6)

Execution Units: ALU #0

Exceptions: none

SHA512SUM0

Description:

Instruction Format: R3

39 33	33	3231	30 27	26 22	21 17	16 12	11 7	6 0
26_7	~	2_2	$\sim_4$	30_5	$\sim_5$	Ra_5	Rt_5	2_7
26_7	~	2_2	$\sim_4$	30_5	$\sim_5$	Va_5	Vt_5	38_7

SHA512SUM1

Description:

Instruction Format: R3

39 33	33	3231	30 27	26 22	21 17	16 12	11 7	6 0
26_7	~	2_2	$\sim_4$	31_5	$\sim_5$	Ra_5	Rt_5	2_7
26_7	~	2_2	$\sim_4$	31_5	$\sim_5$	Va_5	Vt_5	38_7

SM3P0

Description:

P0 transform of SM3 hash function.

Instruction Format: R3

39 33	33	3231	30 27	26 22	21 17	16 12	11 7	6 0
26_7	~	2_2	$\sim_4$	14_5	$\sim_5$	Ra_5	Rt_5	2_7
26_7	~	2_2	$\sim_4$	14_5	$\sim_5$	Va_5	Vt_5	38_7

Operation

Rt = Ra ^ rol(Ra,9) ^ rol(Ra,17)

SM3P1

Description:

P1 transform of SM3 hash function.

Instruction Format: R3

39 33	33	3231	30 27	26 22	21 17	16 12	11 7	6 0
26_7	~	2_2	$\sim_4$	15_5	$\sim_5$	Ra_5	Rt_5	2_7
26_7	~	2_2	$\sim_4$	15_5	$\sim_5$	Va_5	Vt_5	38_7

Operation

Rt = Ra ^ rol(Ra,15) ^ rol(Ra,23)

VADD – ADD VECTOR REGISTER-REGISTER

Description:

Add three registers and place the sum in the target register. All register values are integers. All registers are vector registers. Each element is added independently.

Instruction Format: RV3

39 33	32 31	30 25	24 19	18 13	12 7	6 0
4_7	Op_2	Rc_6	Rb_6	Ra_6	Rt_6	38_7

Operation: R3

Op_2	
0	Rt = Ra + Rb + Rc
1	reserved
2	Rt = Ra + Rb + Rc + 1
3	Rt = Ra + Rb + Rc – 1

Operation:

Vt = Va + Vb

Clock Cycles: 8

Execution Units: All Integer ALU's

Exceptions: none

Notes:

VADDS – ADD VECTOR AND SCALAR REGISTER-REGISTER

Description:

Add three registers and place the sum in the target register. All register values are integers. If Vc is not used, it is assumed to be zero. All registers are vector registers except Rb which is a scalar register.

Instruction Format: RV3

39 33	32 31	30 25	24 19	18 13	12 7	6 0
4_7	Op_2	Vc_6	Rb_6	Va_6	Vt_6	39_7

Operation: R3

Op_2	
0	Rt = Ra + Rb + Rc
1	reserved
2	Rt = Ra + Rb + Rc + 1
3	Rt = Ra + Rb + Rc – 1

Clock Cycles: 1

Execution Units: All Integer ALU's

Exceptions: none

Notes:

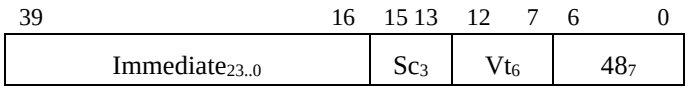
VADDSI – ADD SHIFTED IMMEDIATE

Description:

Add a vector register and immediate value and place the result in the target vector register. The immediate is shifted left in multiples of 20 bits and zero extended to the machine width. This instruction may be used to build a large constant in a register. Note the shift is a multiple of only 20 bits while the constant may provide up to 24 bits. The extra four bits may be set to zero when building a constant. Each element of the vector is added independently.

The 20-bit increment was chosen to match the size supported by other immediate operation instructions like ORI. It has been rounded down to a size that is easily readable in assembly language as hex numbers. Note also that Rt is both a source register and a target register. This provides more bits for the immediate constant. It is envisioned that the vast majority of the time this instruction will follow one which has separate source and target operands.

Instruction Format: RIS



Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = Ra | immediate << Sc * 20

Exceptions:

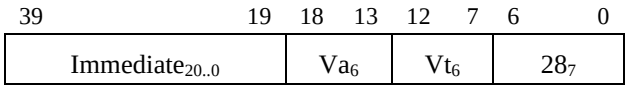
Notes:

VADDI - ADD IMMEDIATE

Description:

Add a vector register and immediate value and place the sum in the target vector register. The immediate is sign extended to the machine width. This instruction may also be used to calculate a virtual address. It has the same number of displacement bits as a load or store instruction. Each element of the vector register has the constant added to it.

Instruction Format: RI



Clock Cycles: 1

Execution Units: All ALU's

Operation:

Vt = Va + immediate

Exceptions:

Notes:

VAND – BITWISE ‘AND’ VECTOR REGISTER-REGISTER

Description:

Bitwise ‘And’ two registers with the complement of a third register and place the result in the target register. All register values are integers. If Vc is not used, it is assumed to be zero. All registers are vector registers. Each element is anded independently.

Instruction Format: RV3

39 33	32 31	30 25	24 19	18 13	12 7	6 0
0_7	Op_2	Vc_6	Vb_6	Va_6	Vt_6	38_7

Operation: R3

Op_2	
0	Vt = Va & Vb & ~Vc
1	reserved
2	Vt = Va & Vb & Vc
3	reserved

Clock Cycles: 1

Execution Units: All Integer ALU’s

Exceptions: none

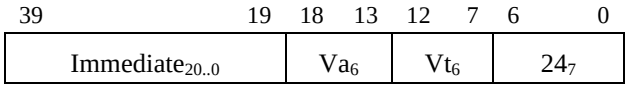
Notes:

VANDI – BITWISE ‘AND’ IMMEDIATE

Description:

Bitwise ‘And’ a vector register, and immediate value and place the result in the target vector register. The immediate is one extended to the machine width. Each element of the vector is bitwise ‘anded’ with the immediate value.

Instruction Format: RI



Clock Cycles: 1

Execution Units: All ALU’s

Operation:

Vt = Va & immediate

Exceptions:

Notes:

VANDS – BITWISE ‘AND’ VECTOR AND SCALAR REGISTER-REGISTER

Description:

Bitwise ‘And’ three registers and place the result in the target register. All register values are integers. If Vc is not used, it is assumed to be zero. All registers are vector registers, except Rb which is a scalar register. Each element is anded independently.

Instruction Format: RV3

39 33	32 31	30 25	24 19	18 13	12 7	6 0
0_7	Op_2	Vc_6	Rb_6	Va_6	Vt_6	38_7

Operation: R3

Op_2	
0	Vt = Va & Rb & Vc
1	reserved
2	Vt = Va & Rb & ~Vc
3	reserved

Clock Cycles: 1

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

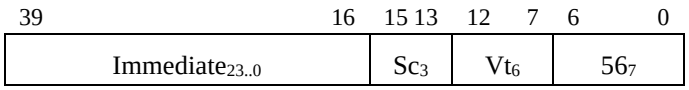
VANDSI – BITWISE ‘AND’ SHIFTED IMMEDIATE

Description:

Bitwise ‘and’ a vector register and immediate value and place the result in the target vector register. The immediate is shifted left in multiples of 20 bits and zero extended to the machine width. This instruction may be used to build a large constant in a register. Note the shift is a multiple of only 20 bits while the constant may provide up to 24 bits. The extra four bits may be set to zero when building a constant. Each element of the vector register is anded independently.

The 20-bit increment was chosen to match the size supported by other immediate operation instructions like ORI. It has been rounded down to a size that is easily readable in assembly language as hex numbers. Note also that Rt is both a source register and a target register. This provides more bits for the immediate constant. It is envisioned that the vast majority of the time this instruction will follow one which has separate source and target operands.

Instruction Format: RIS



Clock Cycles: 1

Execution Units: All ALU’s

Operation:

Rt = Ra | immediate << Sc * 20

Exceptions:

Notes:

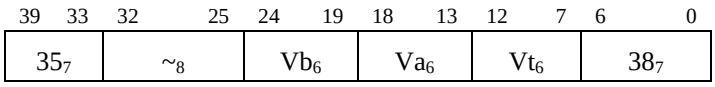
VBMAP – BYTE MAP

Description:

First the target register is cleared, then bytes are mapped from the 16-byte source Ra into bytes in the target register. This instruction may be used to permute the bytes in register Ra and store the result in Rt. This instruction may also pack bytes, wydes or tetras. The map is determined by the low order 64-bits of register Rb. Bytes which are not mapped will end up as zero in the target register. Each nybble of the 64-bit value indicates the target byte in the target register.

Instruction Format: R2

VBMAP Vt, Va, Vb



Operation:

Vector Operation

Execution Units: First Integer ALU

Exceptions: none

Notes:

VCMP - COMPARISON

Description:

Compare two source operands and place the result in the target register. The result is a bit vector identifying the relationship between the two source operands as signed integers.

Operation:

Rt = Ra ? Rb

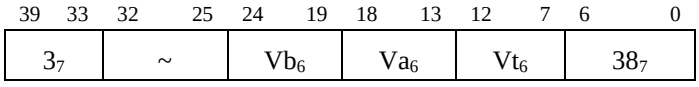
Clock Cycles: 1

Execution Units: All Integer ALU's

Exceptions: none

Notes:

Instruction Format: R3V



Rt Bit	Mnem.	Meaning	Test
		Integer Compare Results	
0	EQ	= equal	a == b
1	NE	< > not equal	a <> b
2	LT	< less than	a < b
3	LE	<= less than or equal	a <= b
4	GE	>= greater than or equal	a >= b
5	GT	> greater than	a > b
6	BC	Bit clear	!a[b]
7	BS	Bit set	a[b]

VCMPI – COMPARE IMMEDIATE

Description:

Compare two source operands and place the result in the target register. The result is a vector identifying the relationship between the two source operands as signed integers.

Operation:

Rt = Ra ? Imm

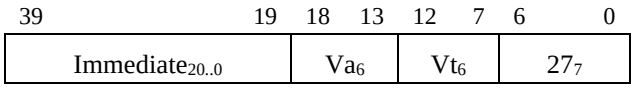
Clock Cycles: 1

Execution Units: All Integer ALU's

Exceptions: none

Notes:

Instruction Format: RI



Rt Bit	Mnem.	Meaning	Test
		Integer Compare Results	
0	EQ	= equal	a == b
1	NE	< > not equal	a <> b
2	LT	< less than	a < b
3	LE	<= less than or equal	a <= b
4	GE	>= greater than or equal	a >= b
5	GT	> greater than	a > b
6	BC	Bit clear	!a[b]
7	BS	Bit set	a[b]

VCMPS – COMPARISON TO SCALAR

Description:

Compare two source operands and place the result in the target register. The result is a bit vector identifying the relationship between the two source operands as signed integers. The first source operand, Va, is a vector register. The second source operand, Rb, is a scalar register. The target register is a vector register.

Operation:

Rt = Ra ? Rb

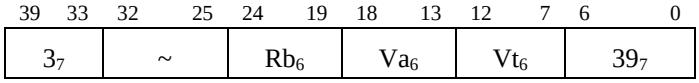
Clock Cycles: 1

Execution Units: All Integer ALU's

Exceptions: none

Notes:

Instruction Format: R3VS



Rt Bit	Mnem.	Meaning	Test
		Integer Compare Results	
0	EQ	= equal	a == b
1	NE	< > not equal	a <> b
2	LT	< less than	a < b
3	LE	<= less than or equal	a <= b
4	GE	>= greater than or equal	a >= b
5	GT	> greater than	a > b
6	BC	Bit clear	!a[b]
7	BS	Bit set	a[b]

VDIV – SIGNED DIVISION

Description:

Divide source dividend operand by divisor operand and place the quotient in the target register. All registers are integer registers. Arithmetic is signed twos-complement values.

Operation:

Vt = Va / Vb

Instruction Format: R3V

39 33	32 25	24 19	18 13	12 7	6 0
17_7	~	Vb_6	Va_6	Vt_6	38_7

Size	Clocks
Octa-byte	280

Execution Units: All Integer ALU's

Exceptions: DBZ

Notes:

VDIVS – SIGNED DIVISION

Description:

Divide source dividend operand by divisor operand and place the quotient in the target register. All registers are integer registers. Arithmetic is signed twos-complement values.

Operation:

Vt = Va / Rb

Instruction Format: R3V

39 33	32 25	24 19	18 13	12 7	6 0
17_7	~	Rb_6	Va_6	Vt_6	39_7

Size	Clocks
Octa-byte	280

Execution Units: All Integer ALU's

Exceptions: DBZ

Notes:

VEOR – BITWISE EXCLUSIVE OR

Description:

Bitwise exclusively or three registers and place the result in the target register. All registers are integer registers.

Instruction Format: R3

39 33	32 31	30 35	24 19	18 13	12 7	6 0
2_7	Op_2	Vc_6	Vb_6	Va_6	Vt_6	38_7

Operation: R3

Op_2	
0	Vt = Va ^ Vb ^ Vc
1	Vt = Va ^ Vb ^ ~Vc
2	Vt = Va ^ Vb ^ -Vc
3	Vt = Va ^ Vb ^ (Vc ^ 8000000000000000h)

Clock Cycles: 1

Execution Units: All Integer ALU's

Exceptions: none

Notes:

VEORS – BITWISE EXCLUSIVE OR WITH SCALAR

Description:

Bitwise exclusively or three registers and place the result in the target register. All registers are integer registers.

Instruction Format: R3

39 33	32 31	30 35	24 19	18 13	12 7	6 0
2_7	Op_2	Vc_6	Rb_6	Va_6	Vt_6	38_7

Operation: R3

Op_2	
0	Vt = Va ^ Rb ^ Vc
1	Vt = Va ^ Rb ^ ~Vc
2	Vt = Va ^ Rb ^ -Vc
3	Vt = Va ^ Rb ^ (Vc ^ 8000000000000000h)

Clock Cycles: 1

Execution Units: All Integer ALU's

Exceptions: none

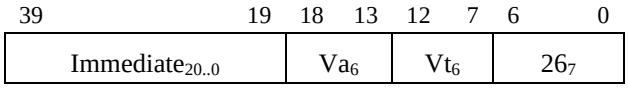
Notes:

VEORI – BITWISE EXCLUSIVE ‘OR’ IMMEDIATE

Description:

Bitwise exclusive ‘Or’ a vector register, and immediate value and place the result in the target vector register. The immediate is one extended to the machine width. Each element of the vector is bitwise exclusive ‘ord’ with the immediate value.

Instruction Format: RI



Clock Cycles: 1

Execution Units: All ALU’s

Operation:

Vt = Va ^ immediate

Exceptions:

Notes:

VMUL – MULTIPLY REGISTER-REGISTER

Description:

Multiply two registers and place the product in the target register. All registers are integer registers. Values are treated as signed integers.

Instruction Format: R3

39 33	3231	30 25	24 19	18 13	12 7	6 0
16_7	~	Vc_6	Vb_6	Va_6	Vt_6	38_7

Size	Clocks
Octa-byte	32

Operation: R2

Rt = Ra * Rb + Rc

Execution Units: All Integer ALU's

Exceptions: none

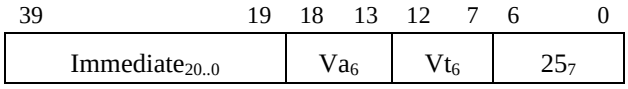
Notes:

VORI – BITWISE ‘OR’ IMMEDIATE

Description:

Bitwise ‘Or’ a vector register, and immediate value and place the result in the target vector register. The immediate is one extended to the machine width. Each element of the vector is bitwise ‘ord’ with the immediate value.

Instruction Format: RI



Clock Cycles: 1

Execution Units: All ALU’s

Operation:

Vt = Va | immediate

Exceptions:

Notes:

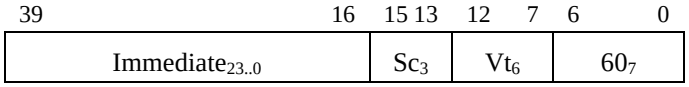
VORSI – BITWISE ‘OR’ SHIFTED IMMEDIATE

Description:

Bitwise ‘or’ a vector register and immediate value and place the result in the target vector register. The immediate is shifted left in multiples of 20 bits and zero extended to the machine width. This instruction may be used to build a large constant in a register. Note the shift is a multiple of only 20 bits while the constant may provide up to 24 bits. The extra four bits may be set to zero when building a constant.

The 20-bit increment was chosen to match the size supported by other immediate operation instructions like ORI. It has been rounded down to a size that is easily readable in assembly language as hex numbers. Note also that Rt is both a source register and a target register. This provides more bits for the immediate constant. It is envisioned that the vast majority of the time this instruction will follow one which has separate source and target operands.

Instruction Format: RIS



Clock Cycles: 1

Execution Units: All ALU’s

Operation:

Rt = Ra | immediate << Sc * 20

Exceptions:

Notes:

VSEQ – SET IF EQUAL

Description:

Compare two source operands for equality and place the result in the target predicate register. The result is a Boolean true or false.

Operation:

Prt = Va == Vb

Clock Cycles: 1

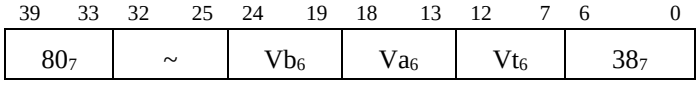
Execution Units: All Integer ALU's

Exceptions: none

Notes:

Instruction Format: R3V

VSEQ Vt, Va, Vb

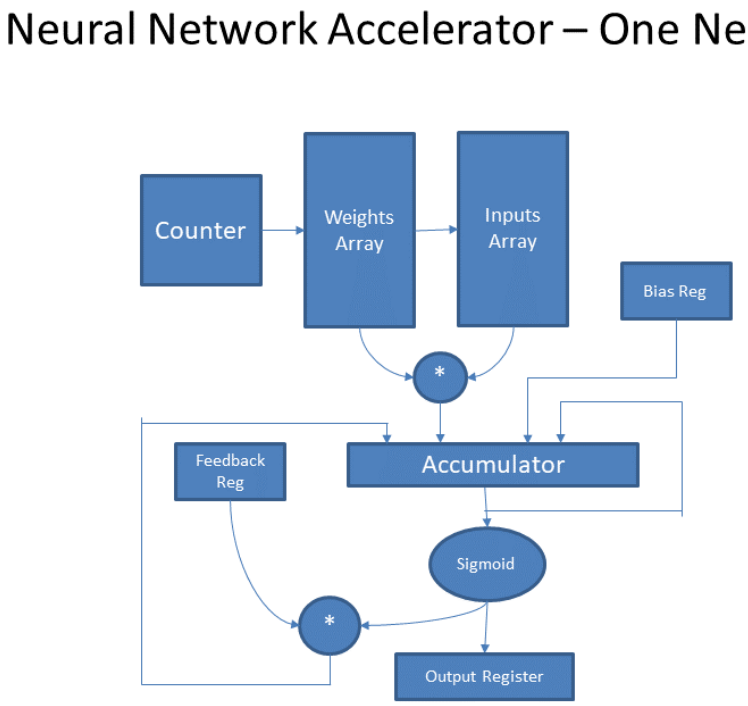


NEURAL NETWORK ACCELERATOR INSTRUCTIONS

OVERVIEW

Included in the ISA are instructions for neural network acceleration. Each neuron is composed of an accumulator that sums the product of weights and inputs and an output activation function. Neurons may be biased with a bias value and may also have feedback from output to input via a feedback constant. The neurons are implemented using 16.16 fixed-point arithmetic. There are 8 neurons in a single layer which may calculate simultaneously. Following is a sketch of the NNA organization. Note that multi-layer networks and additional neurons may be implemented by appropriate software modification of the NNA. The weights and input arrays have a depth of 1024 entries. Not all entries need be used. The number of entries in use is configurable programmatically with the base count and maximum count register using the NNA_MTBC and NNA_MTMC instruction.

Several of the NNA instructions allow multiple neurons to be updated at the same time by representing the neuron update list as a bitmask.



NNA_MFACT – MOVE FROM OUTPUT ACTIVATION

Description:

Move from activation output register. Move a value from the neuron's activation register output to the target register Rt. Bits 0 to 3 of Ra specify the neuron.

Instruction Format: R3

39 33	33	3231	30 27	26 22	21 17	16 12	11 7	6 0
26_7	~	2_2	$\sim_4$	10_5	$\sim_5$	Ra_5	Rt_5	2_7
26_7	~	2_2	$\sim_4$	10_5	$\sim_5$	Va_5	Vt_5	38_7

Clock Cycles: 1

Execution Units: NNA

Notes:

NNA_MTBC – MOVE TO BASE COUNT

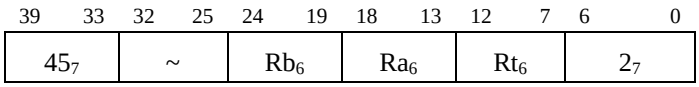
Description:

Move to base count register. Move the value in Ra to the base count register for the neurons identified with a bitmask in Rb. Each bit of Rb represents a neuron. Multiple neurons may be initialized at the same time. Ra contains the base count value.

The neuron calculates the activation output using weight and input array entries between the base count and maximum count inclusive.

Manipulating the base count and maximum count registers ease the implementation of multi-layer networks that do not require the use of all array entries.

Instruction Format: R3



Clock Cycles: 1

Execution Units: NNA

Notes:

NNA_MTBIAS – MOVE TO BIAS

Description:

Move to bias value. Move the value in Ra to the bias register for the neurons identified with a bitmask in Rb. Each bit of Rb represents a neuron. Multiple neurons may be initialized at the same time. Ra contains the bias value.

Instruction Format: R3

39 33	32 25	24 19	18 13	12 7	6 0
42_7	~	Rb_6	Ra_6	Rt_6	2_7

Clock Cycles: 1

Execution Units: NNA

Notes:

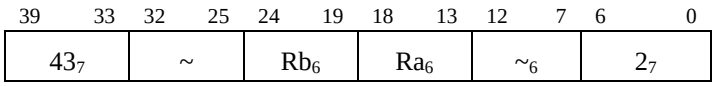
NNA_MTFB – MOVE TO FEEDBACK

Description:

Move to feedback constant. Move the value in Ra to the feedback constant for the neurons identified with a bitmask in Rb. Each bit of Rb represents a neuron. Multiple neurons may be initialized at the same time. Ra contains the feedback constant.

The feedback constant acts to create feedback in the neuron by multiplying the output activation level by the feedback constant and using the result as an input. If no feedback is desired then this constant should be set to zero.

Instruction Format: R2



Clock Cycles: 1

Execution Units: NNA

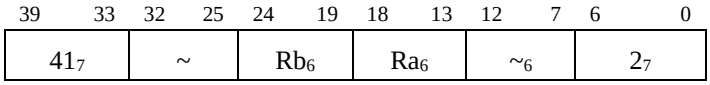
Notes:

NNA_MTIN – MOVE TO INPUT

Description:

Move to input array. Move the value in Ra to the input memory cell identified with Rb. Bits 0 to 15 of Rb specify the memory cell address, bits 32 to 63 of Rb are a bit mask specifying the neurons to update. Bits 0 to 15 of Rb are incremented and stored in Rt.

Instruction Format: R2



Clock Cycles: 1

Execution Units: NNA

Notes:

Multiple neurons may have their inputs updated at the same time with the same value. All the neurons may have the same inputs but the weights for the individual neurons would be different so that a pattern may be recognized.

NNA_MTMC – MOVE TO MAX COUNT

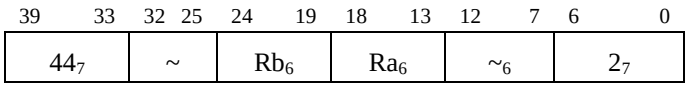
Description:

Move to maximum count register. Move the value in Ra to the maximum count register for the neurons identified with a bitmask in Rb. Each bit of Rb represents a neuron. Multiple neurons may be initialized at the same time. Ra contains the maximum count value.

The maximum count is the upper limit of inputs and weights to use in the calculation of the activation function. The maximum count should not exceed the hardware table size. The table size is 1024 entries.

The neuron calculates the activation output using weight and input array entries between the base count and maximum count inclusive.

Instruction Format: R2



Clock Cycles: 1

Execution Units: NNA

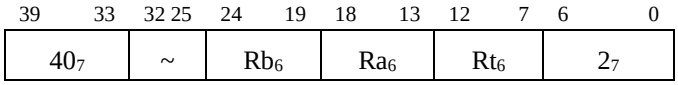
Notes:

NNA_MTWT – MOVE TO WEIGHTS

Description:

Move to weights array. Move the value in Ra to the weight memory cell identified with Rb. Bits 0 to 15 or Rb specify the memory cell address, bits 32 to 63 of Rb are a bit mask specifying the neurons to update. Bits 0 to 15 of Rb are incremented and stored in Rt.

Instruction Format: R2



Clock Cycles: 1

Execution Units: NNA

Notes:

NNA_STAT – GET STATUS

Description:

This instruction gets the status of the neurons. There is a bit in Rt for each neuron. A bit will be set if the neuron is finished performing the calculation of the activation function, otherwise the bit will be clear.

Instruction Format: R3

39 33	33	3231	30 27	26 22	21 17	16 12	11 7	6 0
26_7	~	2_2	$\sim_4$	9_5	$\sim_5$	Ra_5	Rt_5	2_7
26_7	~	2_2	$\sim_4$	9_5	$\sim_5$	Va_5	Vt_5	38_7

Clock Cycles: 1

Execution Units: NNA

Notes:

NNA_TRIG – TRIGGER CALC

Description:

This instruction triggers an NNA cycle for the neurons identified in the bit mask. The bit mask is contained in register Ra.

Instruction Format: R3

39 33	33	3231	30 27	26 22	21 17	16 12	11 7	6 0
26_7	~	2_2	$\sim_4$	8_5	$\sim_5$	Ra_5	Rt_5	2_7
26_7	~	2_2	$\sim_4$	8_5	$\sim_5$	Va_5	Vt_5	38_7

Clock Cycles: 1

Execution Units: NNA

Notes:

FLOATING-POINT OPERATIONS

PRECISION

Three storage formats are supported for binary floats: 64-bit double precision and 32-bit single precision.

Pr_2	Qualifier	Precision
0	H	Half precision
1	S	Single precision
2	D	Double precision
3	Q	Quad precision

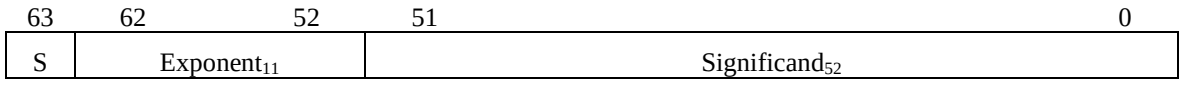
REPRESENTATIONS

BINARY FLOATS

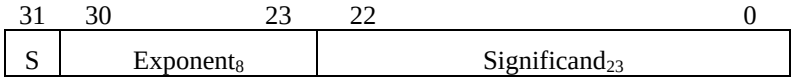
Double Precision, Float:64

The core uses a 64-bit double precision binary floating-point representation.

Double Precision

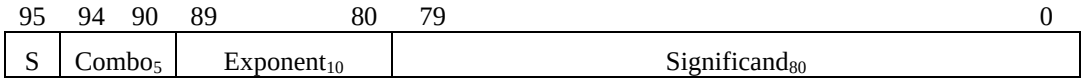


Single Precision, float



DECIMAL FLOATS

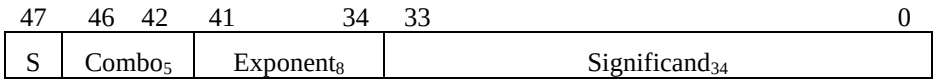
The core uses a 96-bit densely packed decimal double precision floating-point representation.



The significand stores 25 densely packed decimal digits. One whole digit before the decimal point.

The exponent is a power of ten as a binary number with an offset of 1535. Range is 10^{-1535} to 10^{1536}

48-bit single precision decimal floating point:

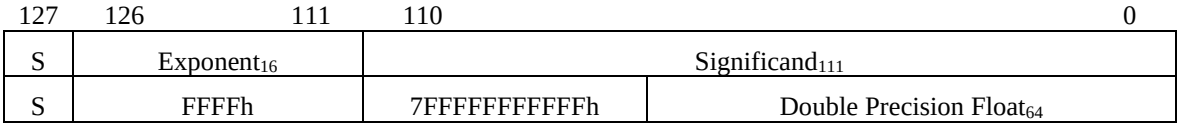


The significand stores 11 DPD digits. One whole digit before the decimal point.

NAN BOXING

Lower precision values are 'NaN boxed' meaning all the bits needed to extend the value to the width of the register are filled with ones. The sign bit of the number is preserved. Thus, lower precision values encountered in calculations are treated as NaNs.

Example: NaN boxed double precision value.



ROUNDING MODES

BINARY FLOAT ROUNDING MODES

Rm3	Rounding Mode
000	Round to nearest ties to even
001	Round to zero (truncate)
010	Round towards plus infinity
011	Round towards minus infinity
100	Round to nearest ties away from zero
101	Reserved
110	Reserved
111	Use rounding mode in float control register

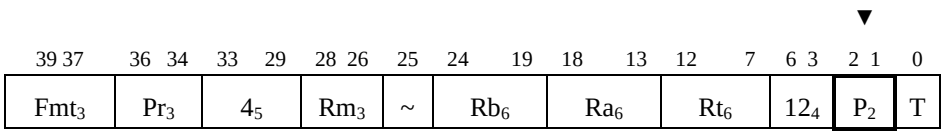
DECIMAL FLOAT ROUNDING MODES

Rm3	Rounding Mode
000	Round ceiling
001	Round floor
010	Round half up
011	Round half even
100	Round down
101	Reserved
110	Reserved
111	Use rounding mode in float control register

GENERAL INSTRUCTION FORMAT

PRECISION:

A two-bit field, P, in the instruction determines the precision of the calculations.



P_2	Precision
0	Quad
1	Double
2	Single
3	Half

FABS – ABSOLUTE VALUE

Description:

This instruction computes the absolute value of the contents of the source operand and places the result in Rt. The sign bit of the value is cleared. No rounding occurs.

Integer Instruction Format: FLT1

FABS Rt, Ra

39 36	35 34	33 31	30 27	26 22	21 17	16 12	11 7	6 0
8_4	Pr_2	~$_3$	~$_4$	1_5	0_5	Ra_5	Rt_5	16_7
12_4	Pr_2	~$_3$	~$_4$	1_5	0_5	Va_5	Vt_5	16_7

Operation:

Ft = Abs(Fa)

Execution Units: All FPUs

Clock Cycles: 1

Exceptions: none

Notes:

Pr_2		**Precision**	**Clocks**
0	H	Half precision	1
1	S	Single precision	1
2	D	Double precision	1
3	Q	Quad precision	1

FADD –FLOAT ADDITION

Description:

Add two source operands and place the sum in the target register. Values are treated as floating-point values.

Supported Operand Sizes:

Operation:

Rt = Ra + Rb

Execution Units: All FPU's

Exceptions: none

Notes:

Instruction Format: FLT2

FADD Ft, Fa, Fb

39 36	35 34	33 31	30 27	26 22	21 17	16 12	11 7	6 0
8_4	Pr_2	Rm_3	$\sim_4$	4_5	Rb_5	Ra_5	Rt_5	16_7
12_4	Pr_2	Rm_3	$\sim_4$	4_5	Vb_5	Va_5	Vt_5	16_7
13_4	Pr_2	Rm_3	$\sim_4$	4_5	Rb_5	Va_5	Vt_5	16_7

Pr_2		**Precision**	**Clocks**
0	H	Half precision	8
1	S	Single precision	8
2	D	Double precision	8
3	Q	reserved	

FCMP - COMPARISON

Description:

Compare two source operands and place the result in the target register. The result is a vector identifying the relationship between the two source operands as floating-point values. This instruction may compare against lower precision immediate values to conserve code space. The source operands are floating-point values, the target operand is an integer. No rounding occurs.

Operation:

Rt = Fa ? Fb or Rt = Fa ? Imm

Clock Cycles: 1

Execution Units: All Integer ALU's

Exceptions: none

Instruction Format: FLT2

FCMP Rt, Ra, Rb

39 36	35 34	33 31	30 35	24 19	18 13	12 7	6 0
8_4	Pr_2	$\sim_3$	13_6	Rb_6	Ra_6	Rt_6	16_7
12_4	Pr_2	$\sim_3$	13_6	Vb_6	Va_6	Vt_6	16_7
13_4	Pr_2	$\sim_3$	13_6	Rb_6	Va_6	Vt_6	16_7

Rt bit	Mnem.	Meaning	Test
		Float Compare Results	
0	EQ	equal	!nan & eq
1	NE	not equal	!eq
2	GT	greater than	!nan & !eq & !lt & !inf
3	UGT	Unordered or greater than	Nan || (!eq & !lt & !inf)
4	GE	greater than or equal	Eq || (!nan & !lt & !inf)
5	UGE	Unordered or greater than or equal	Nan || (!lt || eq)
6	LT	Less than	Lt & (!nan & !inf & !eq)
7	ULT	Unordered or less than	Nan | (!eq & lt)
8	LE	Less than or equal	Eq | (lt & !nan)
9	ULE	unordered less than or equal	Nan | (eq | lt)
10	GL	Greater than or less than	!nan & (!eq & !inf)
11	UGL	Unordered or greater than or less than	Nan | !eq
12	ORD	Greater than less than or equal / ordered	!nan
13	UN	Unordered	Nan

14		Reserved	
15		reserved	

FCONST – LOAD FLOAT CONSTANT

Description:

This instruction loads a constant from the constant ROM and places the value in Rt.

Integer Instruction Format: R1

FCONST Rt, N

39 36	35 34	33 29	28 26	25	24 19	18 13	12 7	6 0
~$_4$	2_2	1_5	~$_3$	~	4_6	N_6	Rt_6	16_7

Clock Cycles: 1

Operation:

Ft = FConst[N]

Execution Units: FPU #0

Clock Cycles: 1

Exceptions: none

Notes:

N_6	Binary64	Decimal	
0	3fe0000000000000	0.5	
1	3ff0000000000000	1.0	
2	4000000000000000	2.0	
3	3ff8000000000000	1.5	
4	0x5FE6EB50C7B537A9		Lomont reciprocal square root magic
21			
22			
23			
57	7FF0000000000000		infinity
58	7FF0000000000001		Nan – infinity - infinity
59	7FF0000000000002		Nan – infinity / infinity
60	7FF0000000000003		Nan – zero / zero
61	7FF0000000000004		Nan – infinity * zero
62	7FF0000000000005		Nan – square root of infinity
63	7FF0000000000006		Nan – square root of negative

FCOS – FLOAT COSINE

Description:

This instruction computes an approximation of the co-sine value of the contents of the source operand and places the result in Rt.

Integer Instruction Format: R1

FCOS Rt, Ra

39 36	35 34	33 31	30 27	26 22	21 17	16 12	11 7	6 0
8_4	Pr_2	Rm_3	$\sim_4$	12_5	1_5	Ra_5	Rt_5	16_7
12_4	Pr_2	Rm_3	$\sim_4$	12_5	1_5	Va_5	Vt_5	16_7

Operation:

Ft = cos(Fa)

Execution Units: FPU #0

Exceptions: none

Notes:

Pr_2		Precision	Clocks
0	H	Half precision	24
1	S	Single precision	
2	D	Double precision	42
3	Q	Quad precision	

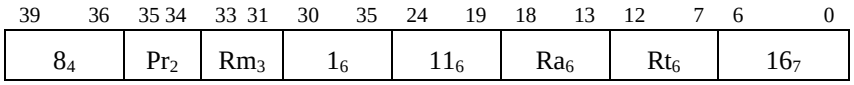
FCVTD2Q – CONVERT DOUBLE TO QUAD PRECISION

Description:

This instruction converts the contents of the source operand to the equivalent of a quad precision value and places the result in Rt.

Integer Instruction Format: R1

FCVTD2Q Rt, Ra



Clock Cycles: 1

Operation:

Rt = Cvt(Ra, double)

Execution Units: FPU #0

Clock Cycles: 1

Exceptions: none

Notes:

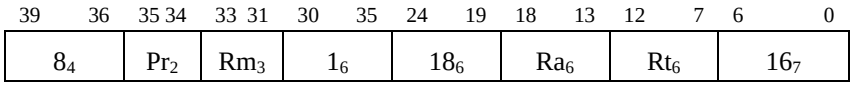
FCVTQ2D – ROUND QUAD TO DOUBLE PRECISION

Description:

This instruction rounds the contents of the source operand to the equivalent of a double precision value and places the result in Rt. This instruction may be used in preparation for a store.

Integer Instruction Format: R1

FCVTQ2D Rt, Ra



Clock Cycles: 2

Operation:

Rt = Round(Ra, double)

Execution Units: FPU #0

Clock Cycles: 2

Exceptions: none

Notes:

FCVTQ2H – ROUND QUAD TO HALF PRECISION

Description:

This instruction rounds the contents of the source operand to the equivalent of a half precision value and places the result in Rt. Note the register continues to contain a quad precision value. This instruction may be used in preparation for a store.

Integer Instruction Format: R1

FCVTQ2H Rt, Ra

39 36	35 34	33 31	30 35	24 19	18 13	12 7	6 0
8_4	Pr_2	Rm_3	1_6	16_6	Ra_6	Rt_6	16_7

Clock Cycles: 1

Operation:

Rt = Round(Ra, half)

Execution Units: FPU #0

Clock Cycles: 1

Exceptions: none

Notes:

FCVTQ2S – ROUND QUAD TO SINGLE PRECISION

Description:

This instruction rounds the contents of the source operand to the equivalent of a single precision value and places the result in Rt. Note the register continues to contain a quad precision value. This instruction may be used in preparation for a store.

Integer Instruction Format: R1

FCVTQ2S Rt, Ra

39 36	35 34	33 31	30 35	24 19	18 13	12 7	6 0
8_4	Pr_2	Rm_3	1_6	17_6	Ra_6	Rt_6	16_7

Clock Cycles: 1

Operation:

Rt = Round(Ra, single)

Execution Units: FPU #0

Clock Cycles: 1

Exceptions: none

Notes:

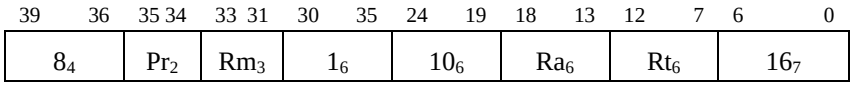
FCVTS2Q – CONVERT SINGLE TO QUAD PRECISION

Description:

This instruction converts the contents of the source operand to the equivalent of a quad precision value and places the result in Rt.

Integer Instruction Format: R1

FCVTD2Q Rt, Ra



Clock Cycles: 1

Operation:

Rt = Cvt(Ra, single)

Execution Units: FPU #0

Clock Cycles: 1

Exceptions: none

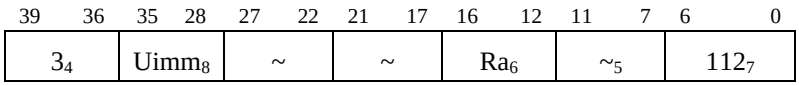
Notes:

FCX – CLEAR FLOATING-POINT EXCEPTIONS

Description:

This instruction clears floating point exceptions. The Exceptions to clear are identified as the bits set in the union of register Ra and an immediate field in the instruction. Either the immediate or Ra should be zero.

Instruction Format: EX



Execution Units: All Floating Point

Operation:

Exceptions:

Bit	Exception Enabled
0	global invalid operation clears the following: - division of infinities - zero divided by zero - subtraction of infinities - infinity times zero - NaN comparison - division by zero
1	overflow
2	underflow
3	divide by zero
4	inexact operation
5	summary exception

FDIV –FLOAT DIVISION

Description:

Divide two source operands and place the quotient in the target register. All registers values are treated as floating-point values.

Supported Operand Sizes:

Operation:

Rt = Ra / Rb or Rt = Ra / Imm or Rt = Imm / Rb

Clock Cycles:

Execution Units: All Integer ALU's

Exceptions: none

Notes:

This instruction is currently implemented as a macro instruction.

Instruction Format: FLT2

FDIV Rt, Ra, Rb

39 36	35 34	33 31	30 35	24 19	18 13	12 7	6 0
8_4	Pr_2	Rm_3	7_6	Rb_6	Ra_6	Rt_6	16_7
12_4	Pr_2	Rm_3	7_6	Vb_6	Va_6	Vt_6	16_7
13_4	Pr_2	Rm_3	7_6	Rb_6	Va_6	Vt_6	16_7

Pr_2		Precision	Clocks
0	H	Half precision	10
1	S	Single precision	30
2	D	Double precision	46
3	Q	reserved	

FDP –FLOAT DOT PRODUCT

Description:

Multiply two pairs of source operands, add the products and place the result in the target register. All register values are treated as quad precision floating-point values.

Note this instruction uses the target register as a source operand and will overwrite the value in that register.

Instruction Format: FLT3

FDP Rt, Ra, Rb, Rc

39 37	36 34	33 31	30 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	7_3	Rc_6	Rb_6	Ra_6	Rt_6	16_7

Operation:

Rt = (Rt * Ra) + (Rb * Rc)

Clock Cycles: 8

Execution Units: All Integer ALU's

Exceptions: none

Notes:

FDX – DISABLE FLOATING POINT EXCEPTIONS

Description:

This instruction disables floating point exceptions. The Exceptions disabled are identified as the bits set in the union of register Ra and an immediate field in the instruction. Either the immediate or Ra should be zero.

Instruction Format: EX

39 36	35 28	27 22	21 17	16 12	11 7	6 0
4_4	$Uimm_8$	~	~	Ra_6	$\sim_5$	112_7

Execution Units: All Floating Point

Operation:

Exceptions:

Bit	Exception Disabled
0	invalid operation
1	overflow
2	underflow
3	divide by zero
4	inexact operation
5	reserved

FEX – ENABLE FLOATING POINT EXCEPTIONS

Description:

This instruction enables floating point exceptions. The Exceptions enabled are identified as the bits set in the union of register Ra and an immediate field in the instruction. Either the immediate or Ra should be zero.

Instruction Format: EX

39 36	35 28	27 22	21 17	16 12	11 7	6 0
5_4	$Uimm_8$	~	~	Ra_6	$\sim_5$	112_7

Execution Units: All Floating Point

Operation:

Exceptions:

Bit	Exception Enabled
0	invalid operation
1	overflow
2	underflow
3	divide by zero
4	inexact operation
5	reserved

FMA –FLOAT MULTIPLY AND ADD

Description:

Multiply two source operands, add a third operand and place the result in the target register. All register values are treated as floating-point values.

Instruction Format: FLT3

FMA Rt, Ra, Rb, Rc

39 36	35 34	33 31	30 35	24 19	18 13	12 7	6 0
0_4	Pr_2	Rm_3	Rc_6	Rb_6	Ra_6	Rt_6	16_7
4_4	Pr_2	Rm_3	Vc_6	Vb_6	Va_6	Vt_6	16_7

Operation:

Rt = Ra * Rb + Rc

Clock Cycles: 8

Execution Units: All Integer ALU's

Exceptions: none

Notes:

FMS –FLOAT MULTIPLY AND SUBTRACT

Description:

Multiply two source operands, subtract a third operand and place the result in the target register. All register values are treated as quad precision floating-point values.

Instruction Format: FLT3

FMS Rt, Ra, Rb, Rc

39 36	35 34	33 31	30 35	24 19	18 13	12 7	6 0
1_4	Pr_2	Rm_3	Rc_6	Rb_6	Ra_6	Rt_6	16_7
5_4	Pr_2	Rm_3	Vc_6	Vb_6	Va_6	Vt_6	16_7

Operation:

Rt = Ra * Rb - Rc

Clock Cycles: 8

Execution Units: All Integer ALU's

Exceptions: none

Notes:

FMUL –FLOAT MULTIPLICATION

Description:

Multiply two source operands and place the product in the target register. All registers values are treated as quad precision floating-point values. An immediate value is converted to quad precision value from single, or double precision.

Operation:

Rt = Ra * Rb

Clock Cycles: 8

Execution Units: All Integer ALU's

Exceptions: none

Notes:

Instruction Format: FLT2

FMUL Rt, Ra, Rb

39 36	35 34	33 31	30 35	24 19	18 13	12 7	6 0
8_4	Pr_2	Rm_3	6_6	Rb_6	Ra_6	Rt_6	16_7
12_4	Pr_2	Rm_3	6_6	Vb_6	Va_6	Vt_6	16_7
13_4	Pr_2	Rm_3	6_6	Rb_6	Va_6	Vt_6	16_7

Pr_2		Precision	Clocks
0	H	Half precision	8
1	S	Single precision	8
2	D	Double precision	8
3		reserved	

FNMUL –FLOAT NEGATE MULTIPLY

Description:

Multiply two source operands and place the product in the target register. All registers values are treated as quad precision floating-point values. An immediate value is converted to quad precision value from single, or double precision.

Operation:

Rt = -(Ra * Rb)

Clock Cycles: 8

Execution Units: All Integer ALU's

Exceptions: none

Notes:

Instruction Format: FLT2

FNMUL Rt, Ra, Rb

39 36	35 34	33 31	30 35	24 19	18 13	12 7	6 0
2_4	Pr_2	Rm_3	30_6	Rb_6	Ra_6	Rt_6	16_7
6_4	Pr_2	Rm_3	30_6	Vb_6	Va_6	Vt_6	16_7

$Opcode_7$		**Precision**	**Clocks**
96	H	Half precision	8
97	S	Single precision	8
98	D	Double precision	8
99		reserved	

FNMA –FLOAT NEGATE MULTIPLY AND ADD

Description:

Multiply two source operands, add a third operand and place the negative of the result in the target register. All register values are treated as floating-point values.

Instruction Format: FLT3

FNMA Rt, Ra, Rb, Rc

39 36	35 34	33 31	30 35	24 19	18 13	12 7	6 0
2_4	Pr_2	Rm_3	Rc_6	Rb_6	Ra_6	Rt_6	16_7
6_4	Pr_2	Rm_3	Vc_6	Vb_6	Va_6	Vt_6	16_7

Operation:

Rt = -(Ra * Rb + Rc)

Clock Cycles: 8

Execution Units: All Integer ALU's

Exceptions: none

Notes:

FNMS –FLOAT NEGATE MULTIPLY AND SUBTRACT

Description:

Multiply two source operands, subtract a third operand and place the negative of the result in the target register. All register values are treated as quad precision floating-point values.

Instruction Format: FLT3

FNMS Rt, Ra, Rb, Rc

39 36	35 34	33 31	30 35	24 19	18 13	12 7	6 0
3_4	Pr_2	Rm_3	Rc_6	Rb_6	Ra_6	Rt_6	16_7
7_4	Pr_2	Rm_3	Vc_6	Vb_6	Va_6	Vt_6	16_7

Operation:

Rt = -(Ra * Rb – Rc)

Clock Cycles: 8

Execution Units: All Integer ALU's

Exceptions: none

Notes:

FRES – FLOATING POINT RECIPROCAL ESTIMATE

Description:

Estimates the reciprocal of the floating-point number in register Ra and place the result into target register Rt.

Instruction Format: FLT1

39 36	35 34	33 31	30 35	24 19	18 13	12 7	6 0
8_4	Pr_2	$\sim_3$	1_6	23_6	Ra_6	Rt_6	16_7
12_4	Pr_2	$\sim_3$	1_6	23_6	Va_6	Vt_6	16_7

Est_2	Bits	Clocks
0	8	2
1	16	22
2	32	38
3	53	54

Operation:

Rt = fres (Ra)

Execution Units: Floating Point

Notes:

This function is currently micro-coded.

FRSQRTE – FLOAT RECIPROCAL SQUARE ROOT ESTIMATE

Description:

Estimate the reciprocal of the square root of the number in register Ra and place the result into target register Rt.

Instruction Format: FLT1

39 36	35 34	33 31	30 35	24 19	18 13	12 7	6 0
8_4	Pr_2	Rm_3	1_6	22_6	Ra_6	Rt_6	16_7
12_4	Pr_2	Rm_3	1_6	22_6	Va_6	Vt_6	16_7

Est_2	Bits	Clocks
0	9	46
1	17	70
2	34	94
3	68	119

Execution Units: Floating Point

Notes:

Taking the reciprocal square root of a negative number or of infinity results in a Nan output.

This function is currently micro-coded.

FSCALEB –SCALE EXPONENT

Description:

Add the source operand to the exponent. The second source operand is an integer value. No rounding occurs.

Instruction Formats:

FSCALEB Rt, Ra, Rb

39 36	35 34	33 31	30 35	24 19	18 13	12 7	6 0
8_4	Pr_2	Rm_3	0_6	Rb_6	Ra_6	Rt_6	16_7
12_4	Pr_2	Rm_3	0_6	Vb_6	Va_6	Vt_6	16_7
13_4	Pr_2	Rm_3	0_6	Rb_6	Va_6	Vt_6	16_7

Operation:

Clock Cycles:

Execution Units: All Integer ALU's

Exceptions: none

Notes:

FSEQ – FLOAT SET IF EQUAL

Description:

Compares two source operands for equality and places the result in the target register. The result is a Boolean true or false. Positive and negative zero are considered equal. For FSEQ if either operand is a NaN zero the result is false. No rounding occurs.

Operation:

Rt = Ra == Rb

Clock Cycles: 1

Execution Units: All FPU's

Exceptions: none

Notes:

Instruction Formats:

FSEQ Rt, Ra, Rb

39 36	35 34	33 31	30 35	24 19	18 13	12 7	6 0
8_4	Pr_2	$\sim_3$	8_6	Rb_6	Ra_6	Rt_6	16_7
12_4	Pr_2	$\sim_3$	8_6	Vb_6	Va_6	Vt_6	16_7
13_4	Pr_2	$\sim_3$	8_6	Rb_6	Va_6	Vt_6	16_7
14_4	Pr_2	$\sim_3$	8_6	Vb_6	Va_6	Rt_6	16_7

Pr_2		Precision	Clocks
0	H	Half precision	1
1	S	Single precision	1
2	D	Double precision	1
3	Q	Quad precision	1

FSGNJ – FLOAT SIGN INJECT

Description:

Copy the sign of Ra and the exponent and significand of Rb into the target register Rt. No rounding occurs.

Operation:

Rt = {Ra.sign, Rb.exp, Rb.sig}

Clock Cycles: 1

Execution Units: All FPU's

Exceptions: none

Notes:

Instruction Formats:

FSGNJ Rt, Ra, Rb

39 36	35 34	33 31	30 35	24 19	18 13	12 7	6 0
8_4	Pr_2	$\sim_3$	16_6	Rb_6	Ra_6	Rt_6	16_7
12_4	Pr_2	$\sim_3$	16_6	Vb_6	Va_6	Vt_6	16_7
13_4	Pr_2	$\sim_3$	16_6	Rb_6	Va_6	Vt_6	16_7

Pr_2		Precision	Clocks
0	H	Half precision	1
1	S	Single precision	1
2	D	Double precision	1
3	Q	reserved	

FSGNJN – FLOAT NEGATIVE SIGN INJECT

Description:

Copy the negative of the sign of Ra and the exponent and significand of Rb into the target register Rt. No rounding occurs.

Operation:

Rt = {~Ra.sign, Rb.exp, Rb.sig}

Clock Cycles: 1

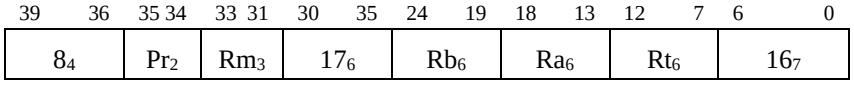
Execution Units: All FPU's

Exceptions: none

Notes:

Instruction Formats:

FSGNJN Rt, Ra, Rb



Pr_2		Precision	Clocks
0	H	Half precision	1
1	S	Single precision	1
2	D	Double precision	1
3		reserved	

FSGNJX – FLOAT SIGN INJECT XOR

Description:

Copy the xor of the sign of Ra and Rb and the exponent and significand of Rb into the target register Rt. No rounding occurs.

Operation:

Rt = {Ra.sign ^ Rb.sign, Rb.exp, Rb.sig}

Clock Cycles: 1

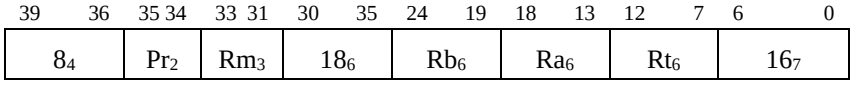
Execution Units: All FPU's

Exceptions: none

Notes:

Instruction Formats:

FSGNJX Rt, Ra, Rb



Pr_2		Precision	Clocks
0	H	Half precision	1
1	S	Single precision	1
2	D	Double precision	1
3	Q	reserved	

FSIGMOID – SIGMOID APPROXIMATE

Description:

This function uses a 1024 entry 32-bit precision lookup table with linear interpolation to approximate the logistic sigmoid function in the range -8.0 to +8.0. Outside of this range 0.0 or +1.0 is returned. The sigmoid output is between 0.0 and +1.0. The value of the sigmoid for register Ra is returned in register Rt as a 64-bit double precision floating-point value.

Instruction Format: FLT1

39 36	35 34	33 31	30 27	26 22	21 17	16 12	11 7	6 0
8_4	Pr_2	Rm_3	$\sim_4$	12_5	16_5	Ra_5	Rt_5	16_7
12_4	Pr_2	Rm_3	$\sim_4$	12_5	16_5	Va_5	Vt_5	16_7

Clock Cycles: 5

Execution Units: Floating Point

FSIGN – SIGN OF NUMBER

Description:

This instruction provides the sign of a double precision floating point number contained in a general-purpose register as a floating-point double result. The result is +1.0 if the number is positive, 0.0 if the number is zero, and -1.0 if the number is negative.

Instruction Format: FLT1

39 36	35 34	33 31	30 27	26 22	21 17	16 12	11 7	6 0
8_4	Pr_2	Rm_3	$\sim_4$	1_5	6_5	Ra_5	Rt_5	16_7
12_4	Pr_2	Rm_3	$\sim_4$	1_5	6_5	Va_5	Vt_5	16_7

Clock Cycles: 1

Execution Units: All Floating Point

Operation:

Rt = sign of (Ra)

FSIN – FLOAT SINE

Description:

This instruction computes an approximation of the sine value of the contents of the source operand and places the result in Rt.

Integer Instruction Format: R1

FSIN Rt, Ra

39 36	35 34	33 31	30 27	26 22	21 17	16 12	11 7	6 0
8_4	Pr_2	Rm_3	$\sim_4$	12_5	0_5	Ra_5	Rt_5	16_7
12_4	Pr_2	Rm_3	$\sim_4$	12_5	0_5	Va_5	Vt_5	16_7

Operation:

Ft = sin(Fa)

Execution Units: FPU #0

Clock Cycles: 125

Exceptions: none

Notes:

Pr_2		Precision	Clocks
0	H	Half precision	24
1	S	Single precision	
2	D	Double precision	42
3		reserved	

FSLE – FLOAT SET IF LESS THAN OR EQUAL

Description:

Compares two source operands for less than or equal and places the result in the target register. The target register is a predicate register. The result is a Boolean true or false. Positive and negative zero are considered equal. For FSLE if either operand is a NaN zero the result is false. No rounding occurs. This instruction may also test for greater than or equal by swapping operands.

Instruction Format:

FSLE Prt, Ra, Rb

39 36	35 34	33 31	30 35	24 19	18 13	12 7	6 0
8_4	Pr_2	$\sim_3$	11_6	Rb_6	Ra_6	Rt_6	16_7
12_4	Pr_2	$\sim_3$	11_6	Vb_6	Va_6	Vt_6	16_7
13_4	Pr_2	$\sim_3$	11_6	Rb_6	Va_6	Vt_6	16_7

Operation:

Rt = Fa < Fb

Clock Cycles: 1

Execution Units: All FPU's

Exceptions: none

Notes:

FSLT – FLOAT SET IF LESS THAN

Description:

Compares two source operands for less than and places the result in the target register. The target register is a predicate register. The result is a Boolean true or false. Positive and negative zero are considered equal. For FSLT if either operand is a NaN zero the result is false. No rounding occurs. This instruction may also test for greater than by swapping operands.

Instruction Formats:

FSLT Rt, Ra, Rb

39 36	35 34	33 31	30 35	24 19	18 13	12 7	6 0
8_4	Pr_2	$\sim_3$	10_6	Rb_6	Ra_6	Rt_6	16_7
12_4	Pr_2	$\sim_3$	10_6	Vb_6	Va_6	Vt_6	16_7
13_4	Pr_2	$\sim_3$	10_6	Rb_6	Va_6	Vt_6	16_7

Operation:

Rt = Ra < Rb

Clock Cycles: 1

Execution Units: All FPU's

Exceptions: none

Notes:

FSNE – FLOAT SET IF NOT EQUAL

Description:

Compares two source operands for equality and places the result in the target predicate register. The result is a Boolean true or false. Positive and negative zero are considered equal. 16, 32, 64, and 128-bit immediates are supported. No rounding occurs.

Operation:

Prt = Fa != Fb or Prt = Fa != Imm

Clock Cycles: 1

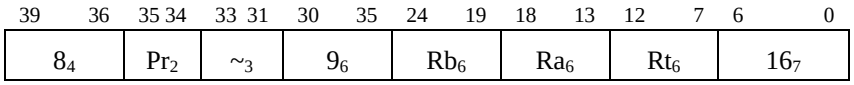
Execution Units: All FPU's

Exceptions: none

Notes:

Instruction Formats:

FSNE Ft, Fa, Fb

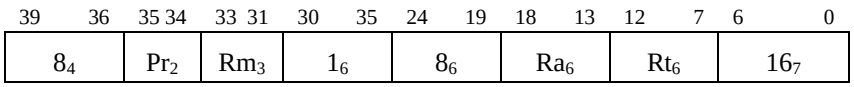


FSQRT – FLOATING POINT SQUARE ROOT

Description:

Take the square root of the floating-point number in register Ra and place the result into target register Rt. The sign bit (bit 63) of the register is set to zero. This instruction can generate NaNs.

Instruction Format: FLT1



Operation:

Rt = fsqrt (Ra)

Clock Cycles: 72

Execution Units: Floating Point

FSUB –FLOAT SUBTRACTION

Description:

Subtract two source operands and place the difference in the target register.

Supported Operand Sizes:

Instruction Format: FLT2

FSUB Ft, Fa, Fb

39 36	35 34	33 31	30 35	24 19	18 13	12 7	6 0
8_4	Pr_2	Rm_3	5_6	Rb_6	Ra_6	Rt_6	16_7
12_4	Pr_2	Rm_3	5_6	Vb_6	Va_6	Vt_6	16_7
13_4	Pr_2	Rm_3	5_6	Rb_6	Va_6	Vt_6	16_7

Operation:

Ft = Fa - Fb

Clock Cycles: 8

Execution Units: All Integer ALU's

Exceptions: none

Notes:

FTRUNC – TRUNCATE VALUE

Description:

The FTRUNC instruction truncates off the fractional portion of the number leaving only a whole value. For instance, ftrunc(1.5) equals 1.0. Ftrunc does not change the representation of the number. To convert a value to an integer in a fixed-point representation see the FTOI instruction.

Instruction Format: FLT1

39 36	35 34	33 31	30 35	24 19	18 13	12 7	6 0
8_4	Pr_2	Rm_3	1_6	21_6	Ra_6	Rt_6	16_7

Pr2:

3	Q	Quad precision
2	D	Double precision
1	S	Single precision
0	H	Half precision

Clock Cycles: 1

Execution Units: Floating Point

FTX – TRIGGER FLOATING POINT EXCEPTIONS

Description:

This instruction triggers floating point exceptions. The Exceptions to trigger are identified as the bits set in the union of register Ra and an immediate field in the instruction. Either the immediate or Ra should be zero.

Instruction Format: EX

39 36	35 28	27 22	21 17	16 12	11 7	6 0
2_4	$Uimm_8$	~	~	Ra_6	$\sim_5$	112_7

Execution Units: All Floating Point

Operation:

Exceptions:

Bit	Exception Enabled
0	global invalid operation
1	overflow
2	underflow
3	divide by zero
4	inexact operation
5	reserved

DFADD – ADD REGISTER-REGISTER

Description:

Add two registers and place the sum in the target register. The values are treated as quad precision decimal floating-point values.

Instruction Format: DFLT

39 36	35 34	33 31	30 35	24 19	18 13	12 7	6 0
8_4	Pr_2	Rm_3	4_6	Rb_6	Ra_6	Rt_6	17_7

Execution Units: All DFPU's

Operation:

Rt = Ra + Rb

Exceptions:

Notes:

DFMUL – MULTIPLY REGISTER-REGISTER

Description:

Multiply two registers and place the product in the target register. The values are treated as quad precision decimal floating-point values.

Instruction Format: DFLT

39 36	35 34	33 31	30 35	24 19	18 13	12 7	6 0
8_4	Pr_2	Rm_3	6_6	Rb_6	Ra_6	Rt_6	17_7

Execution Units: All ALU's

Operation:

Rt = Ra * Rb

Exceptions:

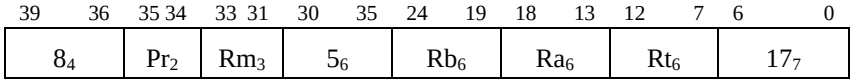
Notes:

DFSUB – ADD REGISTER-REGISTER

Description:

Subtract two registers and place the difference in the target register. If the instruction is a vector addition then Ra and Rt are vector registers. Rb may be either a vector or a scalar register. The values are treated as quad precision decimal floating-point values.

Instruction Format: DFLT



Execution Units: All DFPU's

Operation:

Rt = Ra - Rb

Exceptions:

Notes:

LOAD / STORE INSTRUCTIONS

OVERVIEW

ADDRESSING MODES

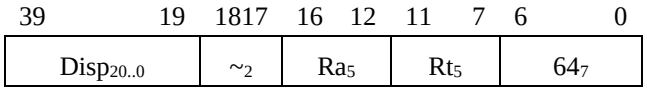
Load and store instructions have two addressing modes, register indirect with displacement and indexed addressing.

LOAD FORMATS

REGISTER INDIRECT WITH DISPLACEMENT FORMAT

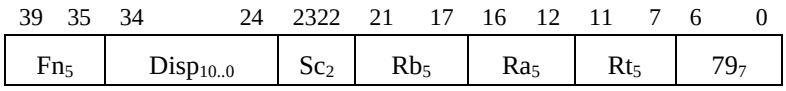
For register indirect with displacement addressing the load or store address is the sum of a register Ra and a displacement constant found in the instruction.

Instruction Format: d[Ra]



Scaled Indexed with Displacement Format

Instruction Format: d[Ra+Rb*]

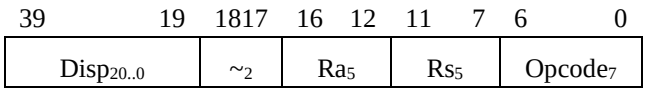


STORE FORMATS

REGISTER INDIRECT WITH DISPLACEMENT FORMAT

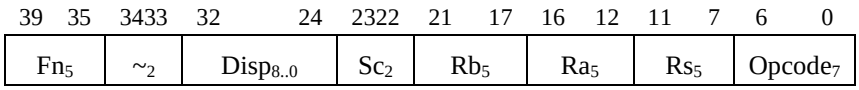
For register indirect with displacement addressing the load or store address is the sum of a register Ra and a displacement constant found in the instruction.

Instruction Format: d[Ra]



Scaled Indexed with Displacement Format

Instruction Format: d[Ra+Rb*]



AMOADD – AMO ADDITION

Description:

Atomically add source operand register Rc to value from memory and store the result back to memory. The original value of the memory cell is stored in register Rt. The memory address is the sum of Ra and Rb.

Supported Operand Sizes: .h

Instruction Formats: AMO

AMOADD Rt, Rc, d[Ra+Rb]

39 36	35 34	3327	26 22	21 17	16 12	11 7	6 0
0_4	Ar	$\sim_7$	Rc_5	Rb_5	Ra_5	Rt_5	92_7

Clock Cycles:

AMOAND – AMO BITWISE 'AND'

Description:

Atomically bitwise 'and' source operand register Ra to value from memory and store the result back to memory. The original value of the memory cell is stored in register Rt. The memory address is the sum of Rc and scaled index Rb.

Supported Operand Sizes: .h

Instruction Formats: AMO

AMOAND Rt, Ra, d[Rc+Rb*Sc]

39 36	35 34	3327	26 22	21 17	16 12	11 7	6 0
1_4	Ar	$\sim_7$	Rc_5	Rb_5	Ra_5	Rt_5	92_7

Clock Cycles:

AMOASL – AMO ARITHMETIC SHIFT LEFT

Description:

Atomically shift the contents of memory to the left by one bit and store the result back to memory. The original value of the memory cell is stored in register Rt. The memory address is the sum of Rc and scaled index Rb.

Supported Operand Sizes: .h

Instruction Formats: AMO

AMOASL Rt, d[Rc+Rb*Sc]

39 36	35 34	3327	26 22	21 17	16 12	11 7	6 0
8_4	Ar	$\sim_7$	Rc_5	Rb_5	Ra_5	Rt_5	92_7

Clock Cycles:

AMOEOR – AMO BITWISE EXCLUSIVELY 'OR'

Description:

Atomically bitwise exclusively 'or' source operand register Ra to value from memory and store the result back to memory. The original value of the memory cell is stored in register Rt. The memory address is the sum of Rc and scaled index Rb.

Supported Operand Sizes: .h

Instruction Formats: AMO

AMOEOR Rt, Ra, d[Rc+Rb*Sc]

39 36	35 34	3327	26 22	21 17	16 12	11 7	6 0
3_4	Ar	$\sim_7$	Rc_5	Rb_5	Ra_5	Rt_5	92_7

Clock Cycles:

AMOLSR – AMO LOGICAL SHIFT RIGHT

Description:

Atomically shift the contents of memory to the right by one bit and store the result back to memory. The original value of the memory cell is stored in register Rt. The memory address is the sum of Rc and scaled index Rb.

Supported Operand Sizes: .h

Instruction Formats: AMO

AMOASL Rt, d[Rc+Rb*Sc]

39 36	35 34	3327	26 22	21 17	16 12	11 7	6 0
9_4	Ar	$\sim_7$	Rc_5	Rb_5	Ra_5	Rt_5	92_7

Clock Cycles:

AMOMAX – AMO MAXIMUM

Description:

Atomically determine the maximum of source operand register Ra and a value from memory and store the result back to memory. The original value of the memory cell is stored in register Rt. The memory address is the sum of Rc and scaled index Rb.

Supported Operand Sizes: .h

Instruction Formats: AMO

AMOMAX Rt, Ra, d[Rc+Rb*Sc]

39 36	35 34	3327	26 22	21 17	16 12	11 7	6 0
5_4	Ar	$\sim_7$	Rc_5	Rb_5	Ra_5	Rt_5	92_7

AMOMAXU – AMO UNSIGNED MAXIMUM

Description:

Atomically determine the maximum of source operand register Ra and a value from memory and store the result back to memory. Values are treated as unsigned integers. The original value of the memory cell is stored in register Rt. The memory address is the sum of Rc and scaled index Rb.

Supported Operand Sizes: .h

Instruction Formats: AMO

AMOMAX Rt, Ra, d[Rc+Rb*Sc]

39 36	35 34	3327	26 22	21 17	16 12	11 7	6 0
13_4	Ar	$\sim_7$	Rc_5	Rb_5	Ra_5	Rt_5	92_7

AMOMIN – AMO MINIMUM

Description:

Atomically determine the minimum of source operand register Ra and a value from memory and store the result back to memory. The original value of the memory cell is stored in register Rt. The memory address is the sum of Rc and scaled index Rb.

Supported Operand Sizes: .h

Instruction Formats: AMO

AMOAND Rt, Ra, d[Rc+Rb*Sc]

39 36	35 34	3327	26 22	21 17	16 12	11 7	6 0

4_{4}	Ar	$\sim_{7}$	Rc_{5}	Rb_{5}	Ra_{5}	Rt_{5}	92_{7}

AMOMINU – AMO UNSIGNED MINIMUM

Description:

Atomically determine the minimum of source operand register Ra and a value from memory and store the result back to memory. Values are treated as unsigned integers. The original value of the memory cell is stored in register Rt. The memory address is the sum of Rc and scaled index Rb.

Supported Operand Sizes: .h

Instruction Formats: AMO

AMOAND Rt, Ra, d[Rc+Rb*Sc]

39 36	35 34	3327	26 22	21 17	16 12	11 7	6 0
12_4	Ar	$\sim_7$	Rc_5	Rb_5	Ra_5	Rt_5	92_7

AMOOR – AMO BITWISE ‘OR’

Description:

Atomically bitwise ‘and’ source operand register Ra to value from memory and store the result back to memory. The original value of the memory cell is stored in register Rt. The memory address is the sum of Rc and scaled index Rb.

Supported Operand Sizes: .h

Instruction Formats: AMO

AMOOR Rt, Ra, d[Rc+Rb*Sc]

39 36	35 34	3327	26 22	21 17	16 12	11 7	6 0
2_4	Ar	$\sim_7$	Rc_5	Rb_5	Ra_5	Rt_5	92_7

Clock Cycles:

AMOROL – AMO ROTATE LEFT

Description:

Atomically rotate the contents of memory to the left by one bit and store the result back to memory. The original value of the memory cell is stored in register Rt. The memory address is the sum of Rc and scaled index Rb.

Supported Operand Sizes: .h

Instruction Formats: AMO

AMOROL Rt, d[Rc+Rb*Sc]

39 36	35 34	3327	26 22	21 17	16 12	11 7	6 0
10_4	Ar	$\sim_7$	Rc_5	Rb_5	Ra_5	Rt_5	92_7

Clock Cycles:

AMOROR – AMO ROTATE RIGHT

Description:

Atomically rotate the contents of memory to the right by one bit and store the result back to memory. The original value of the memory cell is stored in register Rt. The memory address is the sum of Rc and scaled index Rb.

Supported Operand Sizes: .h

Instruction Formats: AMO

AMOROR Rt, d[Rc+Rb*Sc]

39 36	35 34	3327	26 22	21 17	16 12	11 7	6 0
11_4	Ar	$\sim_7$	Rc_5	Rb_5	Ra_5	Rt_5	92_7

Clock Cycles:

AMOSWAP – AMO SWAP

Description:

Atomically swap source operand register Ra with value from memory. The original value of the memory cell is stored in register Rt. The memory address is the sum of Rc and scaled index Rb.

Supported Operand Sizes: .h

Instruction Formats: AMO

AMOSWAP Rt, Ra, d[Rc+Rb]

39 36	35 34	3327	26 22	21 17	16 12	11 7	6 0
6_4	Ar	$\sim_7$	Rc_5	Rb_5	Ra_5	Rt_5	92_7

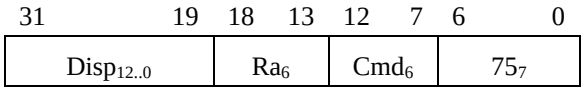
Clock Cycles:

CACHE <CMD>,<EA>

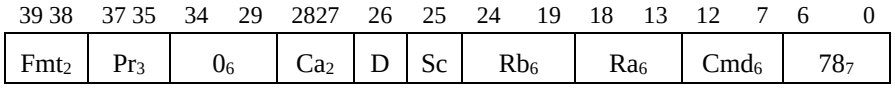
Description:

Issue command to cache controller.

Instruction Format: d[Rn]



Instruction Format: d[Ra+Rb*]



Notes:

Cmd6	Cache	
???000	Ins.	Invalidate cache
???001	Ins.	Invalidate line
???010	TLB	Invalidate TLB
???011	TLB	Invalidate TLB entry
000???	Data	Invalidate cache
001???	Data	Invalidate line
010???	Data	Turn cache off
011???	Data	Turn cache on

CAS – COMPARE AND SWAP

Description:

If the contents of the addressed memory cell equals the contents of Rb then a 64-bit value is stored to memory from the source register Rc. The original contents of the memory cell are loaded into register Rt. The memory address is contained in register Ra. If the operation was successful then Rt and Rb will be equal. The compare and swap operation is an atomic operation performed by the memory controller.

Instruction Format:

31	30 25	24 19	18 13	12 7	6 0
~	Rc_5	Rb_6	Ra_6	Rt_6	93_7

Operation:

```
Rt = memory[[Ra]]
if memory[[Ra]] = Rb
        memory[[Ra]] = Rc
```

Assembler:

```
CAS  Rt, Rb, Rc, [Ra]
```

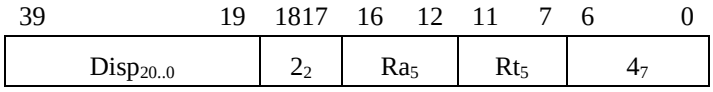
Note:

LDA RN,<EA> - LOAD ADDRESS

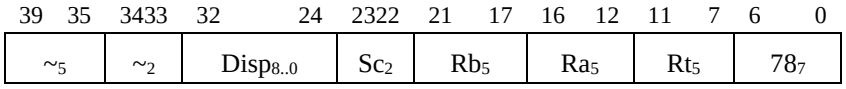
Description:

Load register Rt with the computed memory address.

Instruction Format: d[Rn]



Instruction Format: d[Ra+Rb*Sc]



Clock Cycles: 1

Execution Units: All Integer ALU's

Exceptions: none

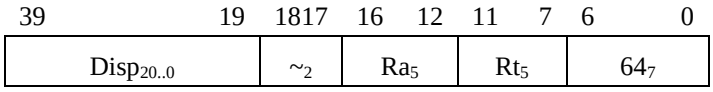
Notes:

LDB RN,<EA> - LOAD BYTE

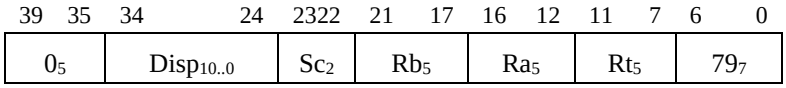
Description:

Load register Rt with a byte from source. The source value is sign extended to the machine width.

Instruction Format: d[Rn]



Instruction Format: d[Ra+Rb*Sc]



Execution Units: AGEN, MEM

Exceptions:

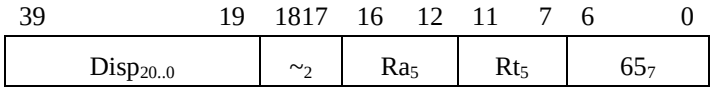
Notes:

LDBU RN,<EA> - LOAD UNSIGNED BYTE

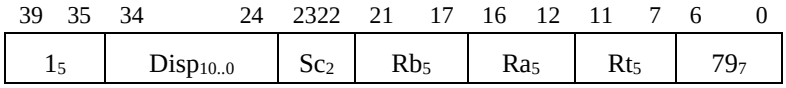
Description:

Load register Rt with a byte from source. The source value is zero extended to the machine width.

Instruction Format: d[Rn]



Instruction Format: d[Ra+Rb*Sc]



Execution Units: AGEN, MEM

Exceptions:

Notes:

LDCTX - LOAD CONTEXT

Description:

Load all registers from the context area of memory. The CTX special purpose register supplies the address.

Instruction Format: LSCTX

39 35	34 24	2322	21 17	16 12	11 7	6 0
15_5	~	~	$~_5$	$~_5$	$~_5$	79_7

Execution Units: AGEN, MEM

Exceptions:

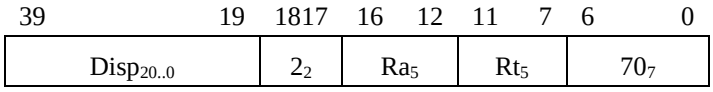
Notes:

LDO RN,<EA> - LOAD OCTA

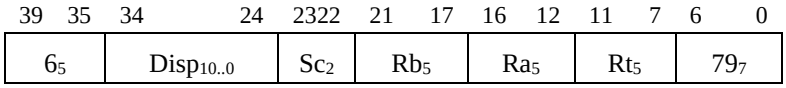
Description:

Load register Rt with an octa from memory. The memory value is sign extended to the machine width.

Instruction Format: d[Rn]



Instruction Format: d[Ra+Rb*Sc]



Execution Units: AGEN, MEM

Exceptions:

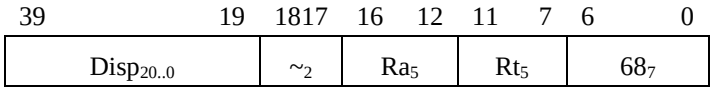
Notes:

LDT RN,<EA> - LOAD TETRA

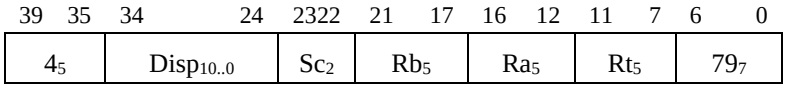
Description:

Load register Rt with a tetra from memory. The memory value is sign extended to the machine width.

Instruction Format: d[Rn]



Instruction Format: d[Ra+Rb*Sc]



Execution Units: AGEN, MEM

Exceptions:

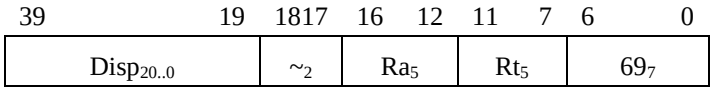
Notes:

LDTU RN,<EA> - LOAD UNSIGNED TETRA

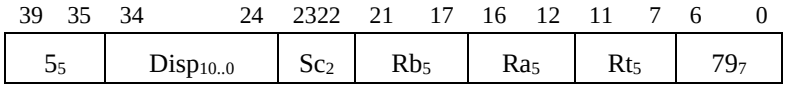
Description:

Load register Rt with a tetra from memory. The memory value is zero extended to the machine width.

Instruction Format: d[Rn]



Instruction Format: d[Ra+Rb*Sc]



Execution Units: AGEN, MEM

Exceptions:

Notes:

LDV VN, [RA], PN, TF - LOAD VECTOR REGISTER

Description:

Load vector register Vt from memory. The memory address is the value in register Ra. Vector elements are loaded depending on the contents of a predicate register.

Instruction Format: [Ra]

39 35	34 32	31 26	25	24 19	18 13	12 7	6 0
Fn_5	$\sim_3$	0_6	T	Rb_6	Ra_6	Vt_6	73_7

Operation:

IF (Rb[ele]=T) THEN Vt = memory[Ra]

Execution Units: AGEN, MEM

Exceptions:

Notes:

LDW RN,<EA> - LOAD WYDE

Description:

Load register Rt with a wyde from source. The source value is sign extended to the machine width.

Instruction Format: d[Rn]

39 19	18 17	16 12	11 7	6 0
$Disp_{20..0}$	$\sim_2$	Ra_5	Rt_5	66_7

Instruction Format: d[Ra+Rb*Sc]

39 35	34 24	23 22	21 17	16 12	11 7	6 0
2_5	$Disp_{10..0}$	Sc_2	Rb_5	Ra_5	Rt_5	79_7

Execution Units: AGEN, MEM

Exceptions:

Notes:

LDWU RN,<EA> - LOAD UNSIGNED WYDE

Description:

Load register Rt with a wyde from source. The source value is zero extended to the machine width.

Instruction Format: d[Rn]

39 19	18 17	16 12	11 7	6 0
$Disp_{20..0}$	$\sim_2$	Ra_5	Rt_5	67_7

Instruction Format: d[Ra+Rb*Sc]

39 35	34 24	23 22	21 17	16 12	11 7	6 0
3_5	$Disp_{10..0}$	Sc_2	Rb_5	Ra_5	Rt_5	79_7

Execution Units: AGEN, MEM

Exceptions:

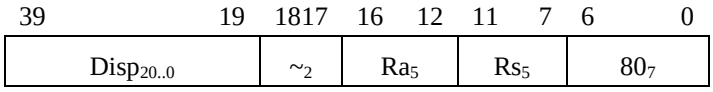
Notes:

STB RS,<EA> - STORE BYTE

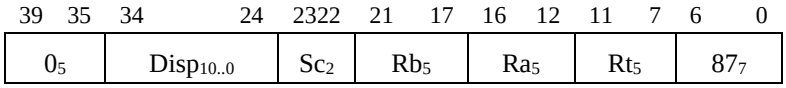
Description:

Store a byte from register Rs to memory.

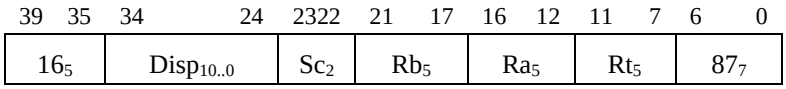
Instruction Format: d[Rn]



Instruction Format: d[Ra+Rb*Sc]



Instruction Format (Ra=r56 to r63): d[Ra+Rb*Sc]



STCTX - STORE CONTEXT TO MEMORY

Description:

Stores all the registers to memory. The memory address is specified by the CTX special purpose register.

Instruction Format: LSCTX

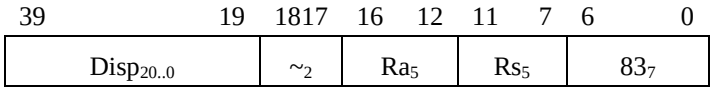
39 35	34 24	2322	21 17	16 12	11 7	6 0
7_5	$\sim_{11}$	$\sim_2$	$\sim_5$	$\sim_5$	$\sim_5$	87_7

STO RS,<EA> - STORE TO MEMORY

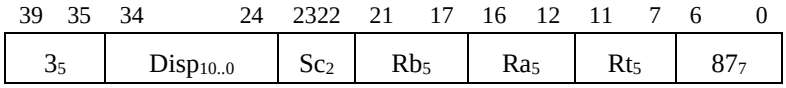
Description:

Store a value from register Rs to memory.

Instruction Format: d[Rn]



Instruction Format: d[Ra+Rb*Sc]

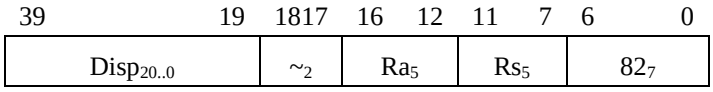


STT RS,<EA> - STORE TETRA

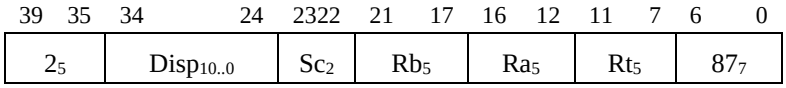
Description:

Store a tetra from register Rs to memory.

Instruction Format: d[Rn]



Instruction Format: d[Ra+Rb*Sc]

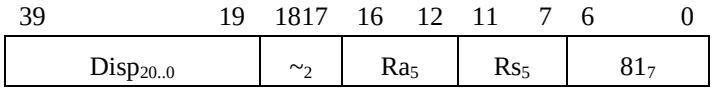


STW RS,<EA> - STORE WYDE

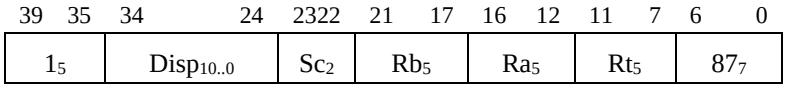
Description:

Store a wyde from register Rs to memory.

Instruction Format: d[Rn]



Instruction Format: d[Ra+Rb*Sc]



BCMP – BLOCK COMPARE

Description:

This instruction compares data from the memory location addressed by Ra to the memory location addressed by Rb until the loop counter LC reaches zero or until a mismatch occurs. Ra and Rb increment by the specified amount. This instruction is interruptible. A predicate register is set to true if the entire block is equal, otherwise it is set to false.

Instruction Format:

39 37	36 34	33 31	30 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	$\sim_3$	Sz_4	Im_2	Rb_6	Ra_6	Rt_6	109_7

Sz_4	Adjustment Amount
1	1
2	2
3	4
4	8
5	16
15	-1
14	-2
13	-4
12	-8
11	-16
others	reserved

Assembler Example

```
LDI LC,200
BCMP.O Pr1,[Ra]+,[Rb]+
SUBF LC,LC,200          ; get index of difference
```

Execution Units: Memory

Operation:

```
temp = 0
Prt = true
```

```
while LC <> 0 and mem[Rb] = mem[Ra]
        Ra = Ra + amt
        Rb = Rb + amt
        LC = LC – 1
        If mem[Rb] != mem[Ra]
                Prt = false
```

BFND – BLOCK FIND

Description:

This instruction compares data from the memory location in Rb to the data in register Ra. A target predicate register is set if the data is found.

Instruction Format:

39 37	36 34	33 31	30 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	$\sim_3$	Sz_4	Im_2	Rb_6	Ra_6	Rt_6	108_7

Sz_4	Adjustment Amount
1	1
2	2
3	4
4	8
5	16
15	-1
14	-2
13	-4
12	-8
11	-16
others	reserved

Execution Units: Memory

Operation:

Description:

This instruction moves a data from the memory location addressed by Ra to the memory location addressed by Rb until the loop counter LC reaches zero. Ra and Rb are adjusted by a specified amount after the move. This instruction is interruptible.

Instruction Format:

39 37	36 34	33 31	30 27	26 25	24 19	18 13	12 11	10 9	8 7	6 0
Fmt_3	Pr_3	$\sim_3$	Sz_4	Im_2	Rb_6	Ra_6	$\sim_2$	Bi_2	Ai_2	111_7

Sz_4	Adjustment Amount
0	0
1	1
2	2
3	4
4	8
15	-1
14	-2
13	-4
12	-8
others	reserved

Ai_2 / Bi_2	
0	No change
1	Increment
2	Decrement
3	reserved

Assembler Example

```
LDI LC,200
BMOV.B [Ra]+,[Rb]+
```

Execution Units: Memory

Operation:

```
temp = 0
while LC <> 0
        t0 = mem[Ra]
        mem[Rb] = t0
        Ra = Ra + amt
        Rb = Rb + amt
        LC = LC – 1
```

BSET – BLOCK SET

Description:

This instruction stores data contained in register Ra to consecutive memory locations beginning at the address in Ra until the loop counter reaches zero.

Instruction Format:

39 29	28 25	24 19	18 13	12 7	6 0
$\sim_{11}$	Sz_4	$\sim_6$	Ra_6	Rs_6	110_7

Sz_4	Adjustment Amount
1	1
2	2
3	4
4	8
15	-1
14	-2
13	-4
12	-8
others	reserved

Execution Units: Memory

Operation:

```
if LC <> 0
        mem[Ra] = Rb
        Ra = Ra + amt
        LC = LC – 1
```

Assembler Example

```
LDI LC,200
BSETB Rb,[Ra]+
```

VECTOR SPECIFIC INSTRUCTIONS

MFVL – MOVE FROM VECTOR LENGTH

Description:

This instruction moves the vector length register to a general-purpose register. This is an alternate mnemonic for the MOV instruction.

Instruction Format: MOV

23 21	2019	1817	16 12	11 7	6 0
~$_3$	1_2	Rt_2	20_5	Rt_5	15_7

Operation:

Rt = VL

Execution Units: ALU #1

MTVL – MOVE TO VECTOR LENGTH

Description:

This instruction moves a general-purpose register to the vector length register. This is an alternate mnemonic for the MOV instruction. Moving a value larger than the maximum vector length of the machine will result in setting the vector length to the maximum vector length.

Instruction Format: MOV

23 21	2019	1817	16 12	11 7	6 0
~$_3$	Ra_2	1_2	Ra_5	20_5	15_7

Operation:

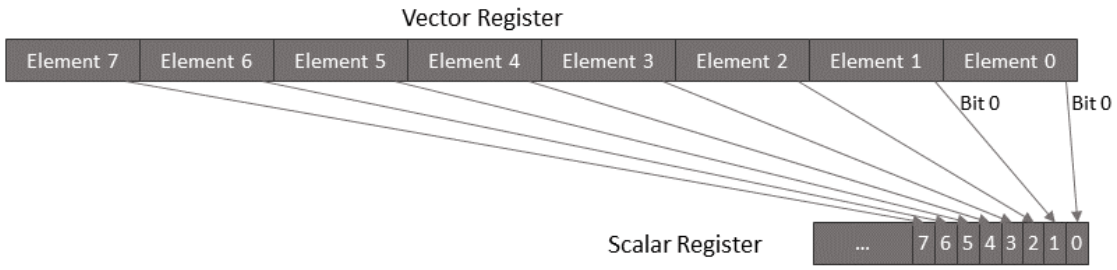
VL = min(Ra, maximum vector length)

V2BITS

Description

Convert Boolean vector to bits. A bit specified by Rb or an immediate of each vector element is copied to the bit corresponding to the vector element in the target register. The target register is a scalar register. Usually, Rb would be zero so that the least significant bit of the vector is copied.

A typical use is in moving the result of a vector set operation into a predicate register.



V2BITS Rt, Ra, Rb

Instruction Format: R3

39 33	3231	3028	27 22	21 17	16 12	11 7	6 0
48_7	Prc_2	$\sim_3$	$Uimm_6$	Rb_5	Va_5	Rt_5	2_7

Operation

For x = 0 to VL-1

Rt.bit[x] = Ra[x].bit[Rb|imm6]

Exceptions: none

Example:

```
cmp v1,v2,v3       ; compare vectors v2 and v3
v2bits pr1,v1,#8   ; move NE status to bits in m1
pred pr1,"TTTTIIII"
vadd v4,v5,v6      ; perform some masked vector operations
pred pr1,"TTTTIIII"
vmuls v7,v8,v9
pred pr1,"TTTTIIII"
vadd v7,v7,v4
```

VEINS / VMOVSV – VECTOR ELEMENT INSERT

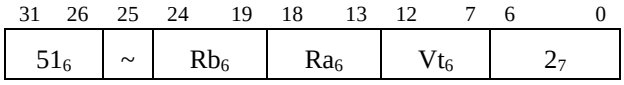
Synopsis

Vector element insert.

Description

A general-purpose register Ra is transferred into one element of a vector register Vt. The element to insert is identified by Rb.

Instruction Format: R2



Operation

Vt[Rb] = Ra

Exceptions: none

VEX / VMOVS – VECTOR ELEMENT EXTRACT

Synopsis

Vector element extract.

Description

A vector register element from Va is transferred into a general-purpose register Rt. The element to extract is identified by Rb. Rb and Rt are scalar registers.

Instruction Format: R2

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	50_7	$\sim_2$	Rb_6	Va_6	Rt_6	2_7

Operation

Rt = Va[Rb]

Exceptions: none

VGNDX – GENERATE INDEX

Description

A value in a register Ra is multiplied by the element number and added to a value in Rb and copied to elements of vector register Vt guided by a vector mask register. Ra is a scalar register. This operation may be used to compute memory addresses for a subsequent vector load or store operation. Only the low order 24-bits of Ra are involved in the multiply. The result of the multiply is a product less than 41 bits in size. The multiply is a fast 24x16 bit multiply.

Instruction Format: R2

31 26	25	24 19	18 13	12 7	6 0
52_6	~	Rb_6	Ra_6	Vt_6	2_7

Operation

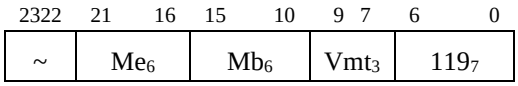
```
y = 0
for x = 0 to VL - 1
    if (Pr[x])
        Vt[y] = Ra * y + Rb
        y = y + 1
```

VMFILL – VECTOR MASK FILL

Description:

Fill the contents of a vector mask register with a mask of ones beginning at Mb_6 and ending at Me_6 inclusive. Fill the remainder of the register with zeros.

Instruction Format:



1 clock cycle

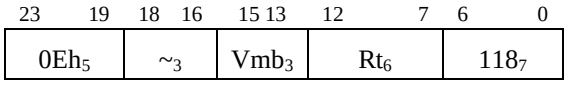
Exceptions: none

VMFIRST – FIND FIRST SET BIT

Description

The position of the first bit set in the mask register is copied to the target register. If no bits are set the value is -1. The search begins at the least significant bit and proceeds to the most significant bit.

Instruction Format: R1



Operation

Rt = first set bit number of (Vm)

Exceptions: none

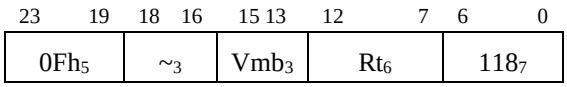
Execution Units: ALU #2

VMLAST – FIND LAST SET BIT

Description

The position of the last bit set in the mask register is copied to the target register. If no bits are set the value is -1. The search begins at the most significant bit of the mask register and proceeds to the least significant bit.

Instruction Format: VMR2



Operation

Rt = last set bit number of (Vm)

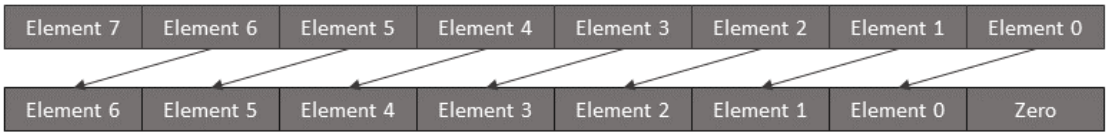
Exceptions: none

Execution Units: ALU #2

VSHLV – SHIFT VECTOR LEFT

Description

Elements of the vector are transferred upwards to the next element position. The first is loaded with the value zero. The highest element is lost. This is also called a slide operation. Elements may be moved a variable number of elements to the left. The image depicts just a single element shift.



Instruction Formats:

VSHLV Vt, Va, Rb

31 26	25 22	21 17	16 12	11 7	6 0
$\sim_6$	6_4	Rb_5	Va_5	Vt_5	18_7

VSHLV Rt, Ra, Imm_5

31 26	25 22	21 17	16 12	11 7	6 0
$\sim_6$	14_4	Imm_5	Va_5	Vt_5	18_7

Operation

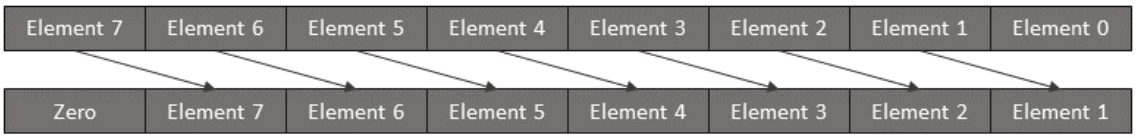
```
Amt = Rb

For x = VL-1 to Amt

        Vt[x] = Va[x-amt]

For x = Amt-1 to 0

        Vt[x] = 0
```

Exceptions: none

VSHRV – SHIFT VECTOR RIGHT

Description

Elements of the vector are transferred downwards to the next element position. The last is loaded with the value zero. This is also called a slide operation. Elements may be moved a variable number of elements to the right. The image depicts just a single element shift.



VSHRV Rt, Ra, Rb

31 26	25 22	21 17	16 12	11 7	6 0
$\sim_6$	7_4	Rb_5	Va_5	Vt_5	18_7

VSHRV Rt, Ra, Imm_5

31 26	25 22	21 17	16 12	11 7	6 0
$\sim_6$	15_4	Imm_5	Va_5	Vt_5	18_7

Operation

Amt = Rb

For x = 0 to VL-Amt

Vt[x] = Va[x+amt]

For x = VL-Amt +1 to VL-1

Vt[x] = 0

Exceptions: none

PREDICATE OPERATIONS

PRLAST – FIND LAST SET BIT

Description

The position of the last bit set in the predicate register is copied to the target register. If no bits are set the value is -1. The search begins at the most significant bit of the mask register and proceeds to the least significant bit.

Instruction Format:

31 29	28 26	25 21	2019	18 15	14 12	11 7	6 0
~$_3$	Pr$_3$	15$_5$	~$_2$	Prb$_4$	~$_3$	Rt$_5$	48h$_8$

Operation

Rt = last set bit number of (Prb)

Exceptions: none

Execution Units: ALUs

BRANCH / FLOW CONTROL INSTRUCTIONS

OVERVIEW

MNEMONICS

There are mnemonics for specifying the comparison method. Floating-point comparisons prefix the branch mnemonic with ‘F’ as in FBEQ. Decimal-floating point comparisons prefix the branch mnemonic with ‘DF’ as in DFBEQ. And finally posit comparisons prefix the branch mnemonic with a ‘P’ as in ‘PBEQ’. For branches that increment register Ra, the mnemonic is prefixed with an ‘I’ as in ‘IBNE’.

PREDICATED EXECUTION

Flow control instructions do not support predicated instruction execution. Instead, a branch instruction must be used to conditionally branch around the instruction. Note it is possible to branch if the predicate register is zero or non-zero. It is also possible to perform a branch-on-bit clear or branch-on-bit set on a predicate register. This takes the place of having a predicate for branch instructions.

CONDITIONS

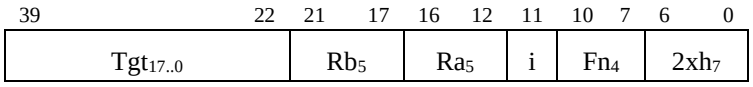
Conditional branches branch to the target address only if the condition is true. The condition is determined by the comparison or logical operation of two general-purpose registers.

> *The original Thor machine used instruction predicates to implement conditional branching. Another instruction was required to set the predicate before branching. Combining compare and branch in a single instruction may reduce the dynamic instruction count. An issue with comparing and branching in a single instruction is that it may lead to a wider instruction format.*

CONDITIONAL BRANCH FORMAT

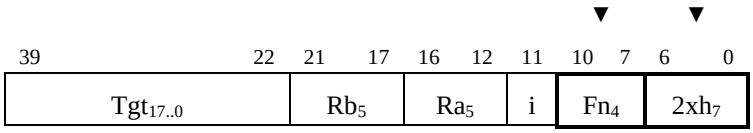
Branches are 40-bit opcodes.

> *A 32-bit opcode does not leave a large enough target field for all cases and would end up using two or more instructions to implement most branches. With the prospect of using two instructions to perform compare then branches as many architectures do, it is more space efficient to simply use a wider instruction format.*



BRANCH CONDITIONS

The branch opcode determines the condition under which the branch will execute.



2x	Data Type Compared
28h	(Unsigned) Address
29h	(Signed) Integers
2Ah	reserved
2Bh	Decimal Float Quad
2Ch	Float half
2Dh	Float single
2Eh	Float double
2Fh	Float quad

Integer / Address Conditions

Fn_4	Unsigned	Signed	Comparison Test
2	LTU	LT	less than
4	GEU	GE	greater or equal
3	LEU	LE	less than or equal
5	GTU	GT	greater than
0	EQ / ENORB	ENOR	Equal
1	NE / EORB	EOR	not equal
6	BC		Bit clear
7	BS		Bit set
8	BC		Bit clear imm
9	BS		Bit set imm
10	NANDB	NAND	And zero
11	ANDB	AND	And non-zero
12	NORB	NOR	Or zero
13	ORB	OR	Or non-zero
others			reserved

Float Conditions

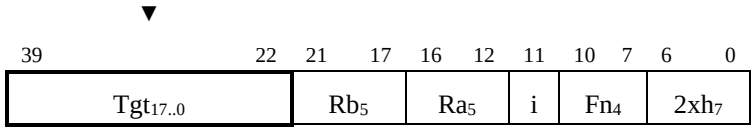
Fn_4	Mnem.	Meaning	Test
0	EQ	equal	!nan & eq
1	NE	not equal	!eq
2	GT	greater than	!nan & !eq & !lt & !inf
3	UGT	Unordered or greater than	Nan || (!eq & !lt & !inf)
4	GE	greater than or equal	Eq || (!nan & !lt & !inf)
5	UGE	Unordered or greater than or equal	Nan || (!lt || eq)
6	LT	Less than	Lt & (!nan & !inf & !eq)
7	ULT	Unordered or less than	Nan | (!eq & lt)
8	LE	Less than or equal	Eq | (lt & !nan)
9	ULE	unordered less than or equal	Nan | (eq | lt)
10	GL	Greater than or less than	!nan & (!eq & !inf)
11	UGL	Unordered or greater than or less than	Nan | !eq
12	ORD	Greater than less than or equal / ordered	!nan
13	UN	Unordered	Nan
14		Reserved	
15		reserved	

BRANCH TARGET

CONDITIONAL BRANCHES

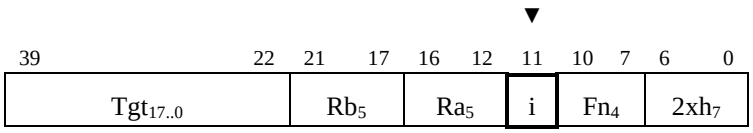
For conditional branches, the target address is formed as the sum of the instruction pointer and a constant specified in the instruction. Relative branches have a range of approximately ±640kB or 20 displacement bits. The target field contains an instruction number relative displacement to the target location. This is the byte displacement divided by five. Encoding targets in this way allows fewer bits to be used to encode the target. Within a subroutine, instructions will always be a multiple of five bytes apart.

The target displacement field is recommended to be at least 16-bits. It is possible to get by with a displacement as small as 12-bits before a significant percentage of branches must be implemented as two or more instructions. The author decided to use a division by five since instructions are five bytes in size and the target must be a multiple of five bytes away from the branches IP. Dividing by five effectively adds two more displacement bits.



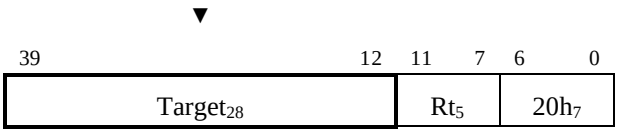
INCREMENTING BRANCHES

Branches may increment the Ra register by one after performing the branch comparison or logical operation. The ‘i’ field of the instruction indicates when an increment should occur. Incrementing branches make use of both the flow control unit and an ALU at the same time.



UNCONDITIONAL BRANCHES

Note that for unconditional branches the target displacement field is byte relative, not instruction relative. This occurs because code functions or subroutines may be relocated at byte addresses and may not be a multiple of five bytes apart. An unconditional subroutine branch call is usually performed to go outside of the current subroutine to a target routine that may be at any byte address. The target displacement field is large enough to accommodate a ±128MB range.



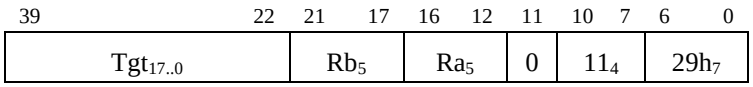
BAND –BRANCH IF LOGICAL AND TRUE

BAND Ra, Rb, label

Description:

Branch if the logical and of source operands results in a non-zero value. The displacement is relative to the address of the branch instruction. This ‘and’ operation reduces the source operand values to a Boolean true or false before performing the operation. A non-zero value is considered true, zero is considered false.

Formats Supported: B



Clock Cycles: 4

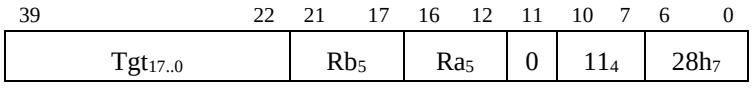
BANDB –BRANCH IF BITWISE AND TRUE

BANDB Ra, Rb, label

Description:

Branch if the bitwise and of source operands results in a non-zero value. The displacement is relative to the address of the branch instruction.

Formats Supported: B



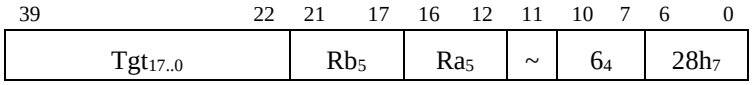
Clock Cycles: 4

BBC – BRANCH IF BIT CLEAR

Description:

This instruction branches to the target address if bit Rb of Ra is clear, otherwise program execution continues with the next instruction. For a further description see Branch Instructions.

Formats Supported: B



Operation:

If (Ra.bit[Rb] == 0)

IP = IP + Constant

Execution Units: Branch

Exceptions: none

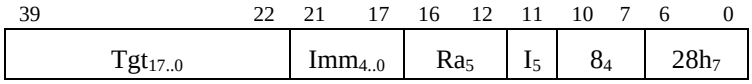
Notes:

BBCI – BRANCH IF BIT CLEAR IMMEDIATE

Description:

This instruction branches to the target address if a bit specified in an immediate field of the instruction of Ra is clear, otherwise program execution continues with the next instruction. For a further description see Branch Instructions.

Formats Supported: B



Operation:

If (Ra.bit[Imm_6] == 0)

IP = IP + Constant

Execution Units: Branch

Exceptions: none

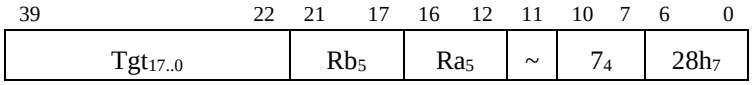
Notes:

BBS – BRANCH IF BIT SET

Description:

This instruction branches to the target address if bit Rb of Ra is clear, otherwise program execution continues with the next instruction. For a further description see Branch Instructions.

Formats Supported: B



Operation:

If (Ra.bit[Rb] == 1)

IP = IP + Constant

Execution Units: Branch

Exceptions: none

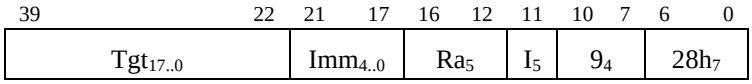
Notes:

BBSI – BRANCH IF BIT SET IMMEDIATE

Description:

This instruction branches to the target address if a bit specified in an immediate field of the instruction of Ra is set, otherwise program execution continues with the next instruction. For a further description see Branch Instructions.

Formats Supported: B



Operation:

If (Ra.bit[Imm$_6$] == 1)

IP = IP + Constant

Execution Units: Branch

Exceptions: none

Notes:

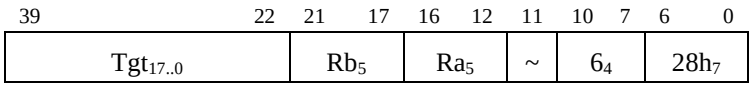
BCC –BRANCH IF CARRY CLEAR

BCC Ra, Rb, label

Description:

Branch if the carry would be set when comparing the first source operand to the second. The first operand is in a register, the second in a register or an immediate value. Both operands are treated as unsigned integer values. The displacement is relative to the address of the branch instruction.

Instruction Format: B



Clock Cycles: 4

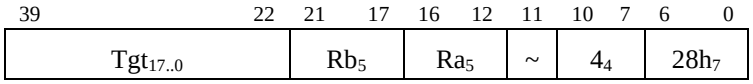
BCS –BRANCH IF CARRY SET

BCS Rm, Rn, label

Description:

This is an alternate mnemonic for the BLO instruction. Branch if the carry would be set because of the comparison of the first operand to the second. The first operand is in a register, the second in a register or an immediate value. Both operands are treated as unsigned integer values. The displacement is relative to the address of the branch instruction.

Instruction Format: B



Clock Cycles: 4

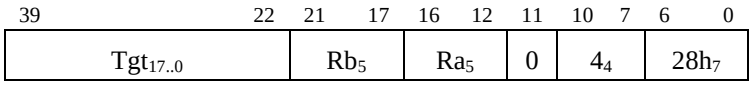
BEOR –BRANCH IF NOT EQUAL

BEOR Ra, Rb, label

Description:

This is an alternate mnemonic for BNE. Branch if source operands are not equal (the exclusive OR is non-zero or true). The displacement is relative to the address of the branch instruction.

Formats Supported: B



Clock Cycles: 4

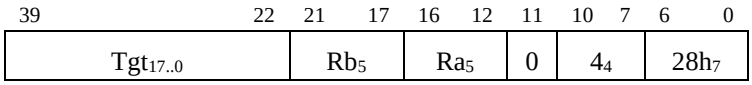
BEQ –BRANCH IF EQUAL

BEQ Ra, Rb, label

Description:

Branch if source operands are equal. The displacement is relative to the address of the branch instruction.

Formats Supported: B



Clock Cycles: 4

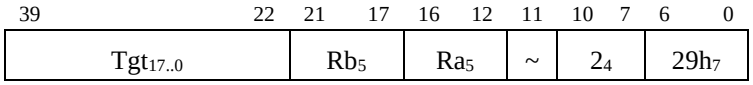
BLE –BRANCH IF LESS THAN OR EQUAL

BLE Ra, Rb, label

Description:

Branch if the first source operand is less than or equal to the second. Both operands are treated as signed integer values. The displacement is relative to the address of the branch instruction.

Formats Supported: B



Clock Cycles: 4

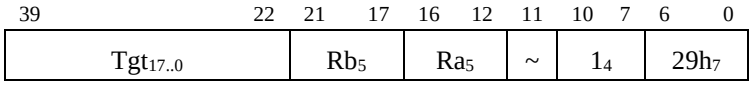
BGE –BRANCH IF GREATER THAN OR EQUAL

BGE Rm, Rn, label

Description:

Branch if the first source operand is greater than or equal to the second. Both operands are treated as signed integer values. The displacement is relative to the address of the branch instruction.

Instruction Format: B



Clock Cycles: 4

BGEU –BRANCH IF UNSIGNED GREATER THAN OR EQUAL

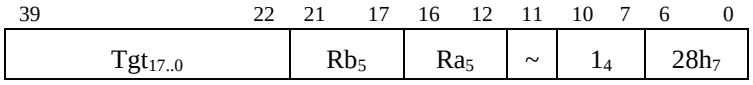
BGEU Rm, Rn, label

Description:

Branch if the first source operand is greater than or equal to the second. The first operand is in a register, the second in a register or an immediate value. Both operands are treated as unsigned integer values. The displacement is relative to the address of the branch instruction.

A postfix instruction containing an immediate value may follow the branch instruction, in which case the immediate is used instead of Rb. Rb should be set to zero.

Instruction Format: B



Clock Cycles: 4

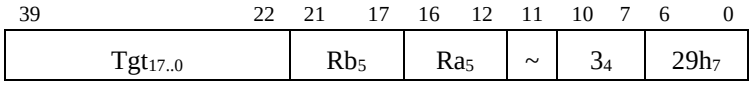
BGT –BRANCH IF GREATER THAN

BGE Rm, Rn, label

Description:

Branch if the first source operand is greater than the second. Both operands are treated as signed integer values. The displacement is relative to the address of the branch instruction.

Instruction Format: B



Clock Cycles: 4

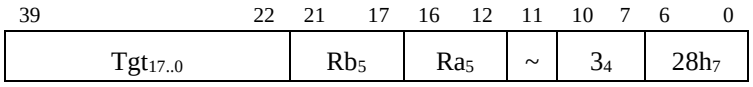
BGTU –BRANCH IF UNSIGNED GREATER THAN

BGE Rm, Rn, label

Description:

Branch if the first source operand is greater than the second. Both operands are treated as unsigned integer values. The displacement is relative to the address of the branch instruction.

Instruction Format: B



Clock Cycles: 4

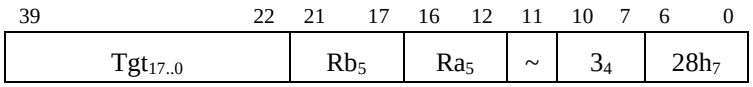
BHI –BRANCH IF HIGHER

BHI Rm, Rn, label

Description:

This is an alternate mnemonic for BGTU. Branch if the first source operand is greater than the second. Both operands are treated as unsigned integer values. The displacement is relative to the address of the branch instruction.

Instruction Format: B



Clock Cycles: 4

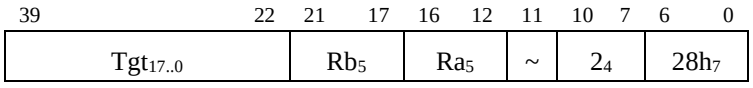
BLEU –BRANCH IF UNSIGNED LESS THAN OR EQUAL

BLEU Ra, Rb, label

Description:

Branch if the first source operand is less than or equal to the second. Both operands are treated as unsigned integer values. The displacement is relative to the address of the branch instruction.

Formats Supported: B



Clock Cycles: 4

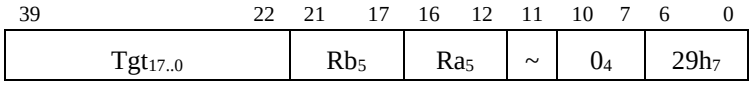
BLT –BRANCH IF LESS THAN

BLT Ra, Rb, label

Description:

Branch if the first source operand is less than the second. Both operands are treated as signed integer values. The displacement is relative to the address of the branch instruction.

Formats Supported: B



Clock Cycles: 4

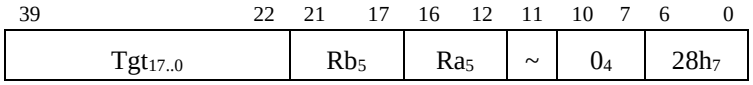
BLTU –BRANCH IF UNSIGNED LESS THAN

BLTU Ra, Rb, label

Description:

Branch if the first source operand is less than the second. Both operands are treated as unsigned integer values. The displacement is relative to the address of the branch instruction.

Formats Supported: B



Clock Cycles: 4

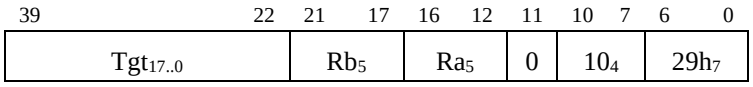
BNAND –BRANCH IF LOGICAL AND FALSE

BNAND Ra, Rb, label

Description:

Branch if the logical and of source operands results in a zero value. The displacement is relative to the address of the branch instruction.

Formats Supported: B



Clock Cycles: 4

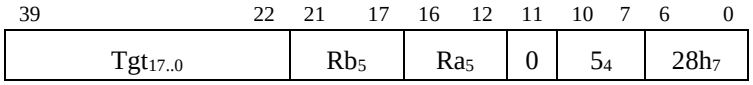
BNE –BRANCH IF NOT EQUAL

BNE Ra, Rb, label

Description:

Branch if source operands are not equal. The displacement is relative to the address of the branch instruction.

Instruction Format: B



Clock Cycles: 4

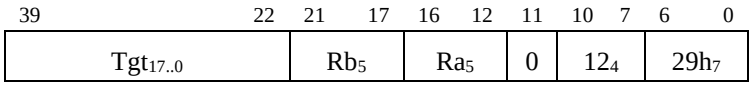
BNOR –BRANCH IF LOGICAL OR FALSE

BNOR Ra, Rb, label

Description:

Branch if the logical or of source operands results in a non-zero value. The displacement is relative to the address of the branch instruction.

Formats Supported: B



Clock Cycles: 4

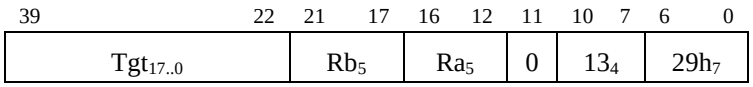
BOR –BRANCH IF LOGICAL OR TRUE

BOR Ra, Rb, label

Description:

Branch if the logical or of source operands results in a non-zero value. The displacement is relative to the address of the branch instruction. This ‘or’ operation reduces the source operand values to a Boolean true or false before performing the operation. A non-zero value is considered true, zero is considered false.

Formats Supported: B



Clock Cycles: 4

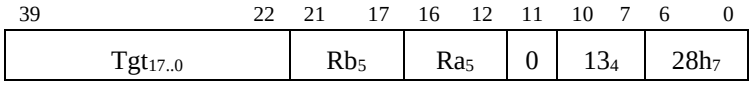
BORB –BRANCH IF BITWISE OR TRUE

BORB Ra, Rb, label

Description:

Branch if the logical or of source operands results in a non-zero value. The displacement is relative to the address of the branch instruction.

Formats Supported: B



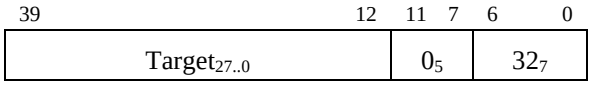
Clock Cycles: 4

BRA – BRANCH ALWAYS

Description:

This instruction always branches to the target address. The target address range is ±128MB.

Formats Supported: BSR



Operation:

IP = IP + Constant

Execution Units: Integer ALU #0

Exceptions: none

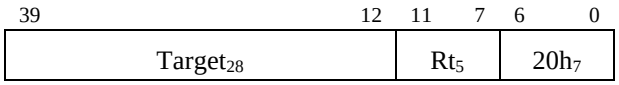
Notes:

BSR – BRANCH TO SUBROUTINE

Description:

This instruction always jumps to the target address. The address of the next instruction is stored in a link register. The target address range is ±128MB. Note the target is used as is.

Formats Supported: BSR



Operation:

Lk = next IP

IP = IP + Constant

Execution Units: Integer ALU #1

Exceptions: none

Notes:

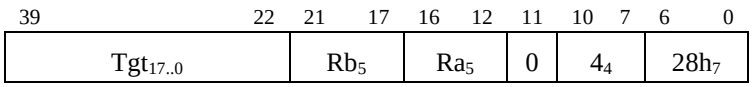
BENOR –BRANCH IF EQUAL

BENOR Ra, Rb, label

Description:

This is an alternate mnemonic for BEQ. Branch if source operands are equal (the exclusive OR is zero or false). The displacement is relative to the address of the branch instruction.

Formats Supported: B



Clock Cycles: 4

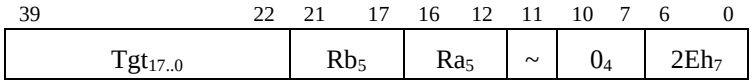
FBEQ –BRANCH IF EQUAL

FBEQ Fa, Fb, label

Description:

Branch if two source operands are equal. The first operand is in a register, the second in a register or an immediate value. Both operands are treated as floating-point values. Positive and negative zero are considered equal. If either operand is a NaN the branch will not be taken. The displacement is relative to the address of the branch instruction.

Formats Supported: B



Mnemonic	Precision	Opcode7
FBEQ.H	Half	2C
FBEQ.S	Single	2D
FBEQ.D	Double	2E
FBEQ.Q	Quad	2F

Clock Cycles: 4

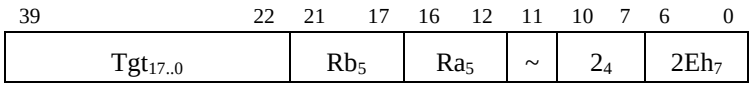
FBGT –BRANCH IF GREATER THAN

FBGT Fa, Fb, label

Description:

Branch if source operand Ra is greater than Rb. Both operands are treated as floating-point values. Positive and negative zero are considered equal. If either operand is a NaN the branch will not be taken. The displacement is relative to the address of the branch instruction.

Formats Supported: B



Mnemonic	Precision	Opcode$_7$
FBGT.H	Half	2C
FBGT.S	Single	2D
FBGT.D	Double	2E
FBGT.Q	Quad	2F

Clock Cycles: 4

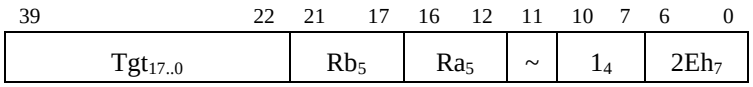
FBNE –BRANCH IF NOT EQUAL

FBNE Fa, Fb, label

Description:

Branch if two source operands are not equal. The first operand is in a register, the second in a register or an immediate value. Both operands are treated as floating-point values. Positive and negative zero are considered equal. The displacement is relative to the address of the branch instruction.

Formats Supported: B



Mnemonic	Precision	Opcode$_7$
FBNE.H	Half	2C
FBNE.S	Single	2D
FBNE.D	Double	2E
FBNE.Q	Quad	2F

Clock Cycles: 4

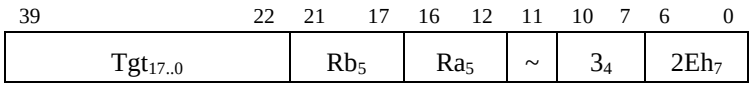
FBUGT –BRANCH IF UNORDERED OR GREATER THAN

FBUGT Fa, Fb, label

Description:

Branch if source operand Ra is greater than Rb or if the comparison is unordered. Both operands are treated as floating-point values. Positive and negative zero are considered equal. The displacement is relative to the address of the branch instruction.

Formats Supported: B



Mnemonic	Precision	Opcode$_7$
FBUGT.H	Half	2C
FBUGT.S	Single	2D
FBUGT.D	Double	2E
FBUGT.Q	Quad	2F

Clock Cycles: 4

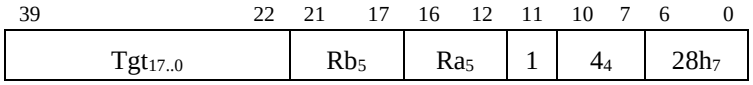
IBEQ – INCREMENT AND BRANCH IF EQUAL

IBEQ Ra, Rb, label

Description:

Branch if source operands are equal. The displacement is relative to the address of the branch instruction. Register Ra is incremented after the compare.

Formats Supported: B



Clock Cycles: 4

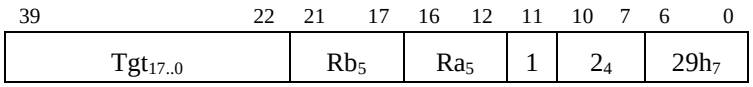
IBLE – INCREMENT AND BRANCH IF LESS THAN OR EQUAL

IBLE Ra, Rb, label

Description:

Branch if the first source operand is less than or equal to the second. Both operands are treated as signed integer values. The displacement is relative to the address of the branch instruction. Register Ra is incremented after the comparison.

Formats Supported: B



Clock Cycles: 4

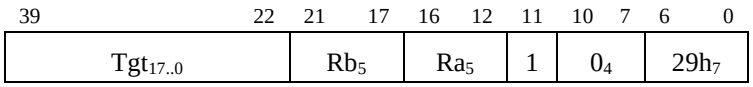
IBLT – INCREMENT AND BRANCH IF LESS THAN

IBLT Ra, Rb, label

Description:

Branch if the first source operand is less than the second. Both operands are treated as signed integer values. The displacement is relative to the address of the branch instruction. Register Ra is incremented after the comparison.

Formats Supported: B



Clock Cycles: 4

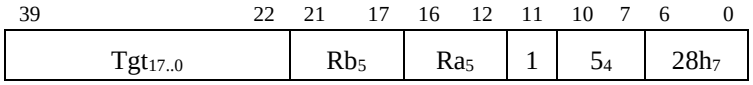
IBNE – INCREMENT AND BRANCH IF NOT EQUAL

BNE Ra, Rb, label

Description:

Branch if source operands are not equal. The displacement is relative to the address of the branch instruction. Register Ra is incremented after the comparison.

Instruction Format: B



Clock Cycles: 4

Execution Units: All ALU's, FCU

JMP – JUMP TO TARGET

Description:

This instruction always jumps to the target address. The target address is the sum of a register and an immediate constant.

Instruction Format: JSR

39 19	18 17	16 12	11 7	6 0
$Immediate_{20..0}$	2_2	Ra_5	0_5	36_7

Operation:

IP = Ra + sign extend (Constant)

Execution Units: Integer ALU #0

Exceptions: none

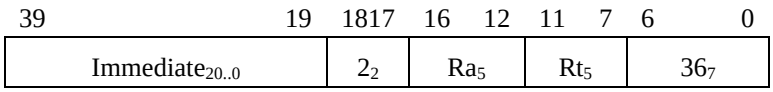
Notes:

JSR – JUMP TO SUBROUTINE

Description:

This instruction always jumps to the target address. The target address is the sum of a register and an immediate constant. The address of the next instruction is stored in a link register.

Instruction Format: JSR



Operation:

Rt = next IP

IP = Ra + sign extend (Constant)

Execution Units: Integer ALU #0

Exceptions: none

Notes:

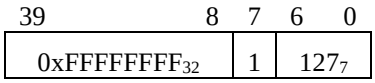
NOP – NO OPERATION

NOP

Description:

This instruction does not perform any operation.

Instruction Format:



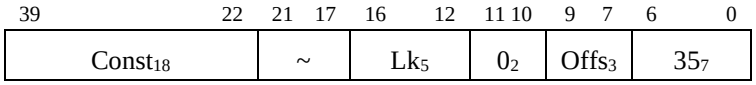
Notes:

RTD – RETURN FROM SUBROUTINE AND DEALLOCATE

Description:

This instruction returns from a subroutine by transferring program execution to the address stored in a link register plus an offset amount. Additionally, the stack pointer is incremented by the amount specified. The const field is shifted left three times before use.

Formats Supported: RTD



Operation:

IP <= Lk + Offs * 5

SP = SP + Const

Execution Units: Branch

Exceptions: none

Notes:

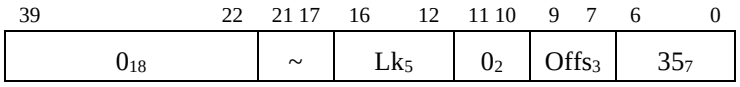
Return address prediction hardware may make use of the RTS instruction.

RTS – RETURN FROM SUBROUTINE

Description:

This instruction returns from a subroutine by transferring program execution to the address stored in a link register.

Formats Supported: RTD



Operation:

IP <= Lk

Execution Units: Branch

Exceptions: none

Notes:

Return address prediction hardware may make use of the RTS instruction.

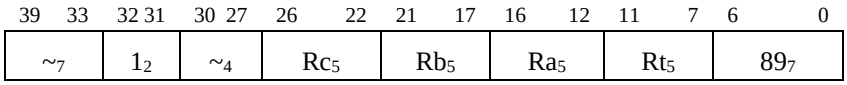
BLEND – BLEND COLORS

Description:

This instruction blends two colors whose values are in Ra and Rb according to an alpha value in Rc. The resulting color is placed in register Rt. The alpha value is a ten-bit value assumed to be a fixed-point number with one whole digit and nine fraction digits. The same alpha value should be placed in each RGB component location of Rc. The color values in Ra and Rb are assumed to be RGB10.10.10 format colors. The result is a RGB10.10.10 format color. Note that a close approximation to 1.0 – alpha is used. Each component of the color is blended independently. Component overflow saturates towards white.

Instruction Format: R3

BLEND Rt, Ra, Rb, Rc



Operation:

Rt.R = (Ra.R * alpha) + (Rb.R * ~alpha)

Rt.G = (Ra.G * alpha) + (Rb.G * ~alpha)

Rt.B = (Ra.B * alpha) + (Rb.B * ~alpha)

Clock Cycles: 2

TRANSFORM – TRANSFORM POINT

Description:

The point transform instruction transforms a point from one location to another using a transform function. The transform function has 12 co-efficients in the form of a matrix used in the calculation.

Points are represented in 18.18 fixed-point format.

Instruction Format:

39 37	36 32	31 27	26 22	21 17	16 12	11 7	6 0
Op_3	Rtz_5	Rty_5	Rz_5	Ry_5	Rx_5	Rtx_5	89_7

Op_3	Operation
0	Return new X,Y,Z
1 to 5	reserved
6	Get coefficient
7	Set coefficient

To set a coefficient Rx specifies which coefficient to set, Ry specifies the value.

Rx	Ry Co-efficient
0	aa
1	ab
2	ac
3	tx
4	ba
5	bb
6	bc
8	ty
9	ca
10	cb
11	cc
12	tz

Clock Cycles: 4

Operation:

Input matrix M:

$$M = \begin{vmatrix} aa & ab & ac & tx \\ ba & bb & bc & ty \\ ca & cb & cc & tz \end{vmatrix}$$

Input point X:

$$X = \begin{vmatrix} x \\ y \\ z \\ 1 \end{vmatrix}$$

Output point X':

$$X' = \begin{vmatrix} x' \\ y' \\ z' \end{vmatrix} = MX = \begin{vmatrix} aa*x + ab*y + ac*z + tx \\ ba*x + bb*y + bc*z + ty \\ ca*x + cb*y + cc*z + tz \end{vmatrix}$$

Clock Cycles: 3

BRK – BREAK

Description:

This instruction is an alternate mnemonic for the CHK instruction where the call number is assumed to be the debug call number (0). This instruction initiates the processor debug routine. The processor enters debug mode. The cause code register is set to indicate execution of a BRK instruction. Interrupts are disabled. The instruction pointer is reset to the vector located from the contents of tvec[3] and instructions begin executing. There should be a jump instruction placed at the break vector location. The address of the BRK instruction is stored in the EIP.

Instruction Format: BRK

39 36	35	34 27	26 22	21 17	16 12	11 9	8 7	6 0
0_4	0	0_8	0_5	0_5	0_5	0_3	0_2	0_7

Operation:

PUSH SR

PUSH IP

EIP = IP

IP = vector at (tvec[3])

Execution Units: Branch

Clock Cycles:

Exceptions: none

Notes:

FENCE – SYNCHRONIZATION FENCE

Description:

All instructions for a particular unit before the FENCE are completed and committed to the architectural state before instructions of the unit type after the FENCE are issued. This instruction is used to ensure that the machine state is valid before subsequent instructions are executed.

Instruction Format:

39 ... 27	26 ... 24	23 ... 16	15 ... 12	11 ... 8	7	6 ... 0
~	Op_3	$Mask_{7..0}$	Aft_4	Bef_4	~	114_7

Mask Bit	Access		
0	Wr	Before	
1	Rd		
2	Out		
3	In		
4	Wr	After	
5	Rd		
6	Out		
7	In		

Aft_4 / Bef_4	Unit
0	MEM
1	ALU
2	FPU
3	Branch
4 to 14	Reserved
15	All units

Op3	Fence Type
0	Normal
1 to 6	reserved
7	TSO fence

IRQ – GENERATE INTERRUPT

Description:

This is an alternate mnemonic of the CHK instruction.

Generate interrupt. This instruction invokes the system exception handler. The return address is pushed onto an internal stack.

The return address stored is the address of the interrupt instruction, not the address of the next instruction. To call system routines use the SYS instruction.

The level of the interrupt is checked and if the interrupt level in the instruction is less than or equal to the current interrupt level then the instruction will be ignored. The interrupt level is specified as the complement of the IPL_3 field.

Instruction Format: CHK

39 36	35	34 27	26 22	21 17	16 12	11 9	8 7	6 0
0_4	0	$Cause_8$	0_5	0_5	0_5	IPL_3	0_2	0_7

Operation:

PUSH SR

PUSH IP

CAUSE = $Cause_8$

IP = vector(tvec[3] + Cause * 8)

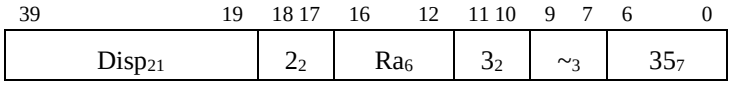
Execution Units: Branch

JMPX – JUMP TO EXCEPTION HANDLER

Description:

This instruction jumps to an exception routine by transferring program execution to the address specified as the sum of a displacement and register Ra. The instruction pointer and status register are pushed onto an internal stack.

Formats Supported: RTE



Operation:

PUSH IP on internal stack

PUSH SR on internal stack

IP = Ra + disp

Execution Units: Branch

Exceptions: none

Notes:

MEMDB – MEMORY DATA BARRIER

Description:

All memory accesses before the MEMDB command are completed before any memory accesses after the data barrier are started. This is an alternate mnemonic for the FENCE instruction.

Instruction Format:

31 30	29 27	26 24	23 16	15 12	11 8	7	6 0
Fmt_2	Pr_3	0_3	$255_{7..0}$	0_4	0_4	~	114_7

Clock Cycles: 1

Execution Units: Memory

MEMSB – MEMORY SYNCHRONIZATION BARRIER

Description:

All instructions before the MEMSB command are completed before any memory access is started. This is an alternate mnemonic for the FENCE instruction.

Instruction Format:

31 30	29 27	26 24	23 16	15 12	11 8	7	6 0
Fmt_2	Pr_3	0_3	$192_{7..0}$	0_4	15_4	~	114_7

Clock Cycles: 1

Execution Units: Memory

PFI – POLL FOR INTERRUPT

Description:

The poll for interrupt instruction polls the interrupt status lines and performs an interrupt service if an interrupt is present. Otherwise, the PFI instruction is treated as a NOP operation. Polling for interrupts is performed by managed code. PFI provides a means to process interrupts at specific points in running software. Rt is loaded with the cause code in the low order twelve bits, and the interrupt level in bits twelve to fourteen of the register.

Instruction Format: OSR2

39 38	37 35	34 13	12 7	6 0
Fmt_2	Pr_3	$Immediate_{21..0}$	Rt_6	115_7

Clock Cycles: 1 (if no exception present)

Operation:

```
if (irq <> 0)
        Rt[11:0] = cause code
        Rt[14:12] = irq level
        PMSTACK = (PMSTACK << 4) | 6
        CAUSE = Const12
        EIP = IP
        IP = tvec[3]
```

Execution Units: Branch

REX – REDIRECT EXCEPTION

Description:

This instruction redirects an exception from an operating mode to a lower operating mode. This instruction if successful jumps to the target exception handler and does not return. If this instruction fails execution will continue with the next instruction.

This instruction may fail if exceptions are not enabled at the target level.

The location of the target exception handler is found in the trap vector register for that operating mode (tvec[xx]).

The cause (cause) and bad address (badaddr) registers of the originating mode are copied to the corresponding registers in the target mode.

If the 'S' bit of the instruction is set, then the privilege level will be set to the bitwise union of the PL_8 field and the value in register Ra. Otherwise the privilege level will remain unchanged.

Instruction Format: EX

39 36	35 28	27	26 22	21 17	16 12	11 9	8 7	6 0
7_4	PL_8	S	~	~	Ra_6	$\sim_3$	Tm_2	112_7

Tm_2	
0	redirect to user mode
1	redirect to supervisor mode
2	redirect to hypervisor mode
3	Redirect to machine mode (from debug)

Clock Cycles: 4

Execution Units: Branch

Example:

```
REX 1               ; redirect to supervisor handler
; If the redirection failed, exceptions were likely disabled at the target level.
; Continue processing so the target level may complete its operation.
RTE                 ; redirection failed (exceptions disabled ?)
```

Notes:

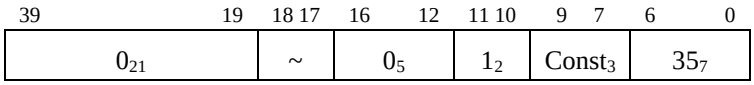
Since all exceptions are initially handled in machine mode the machine handler must check for disabled lower mode exceptions.

RTE – RETURN FROM EXCEPTION

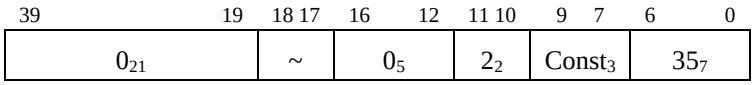
Description:

This instruction returns from an exception routine by transferring program execution to the address stored in an internal stack. This instruction may perform a two-up level return.

Formats Supported: RTE



Formats Supported: RTE – Two up level return.



Operation:

Optionally pop the status register and always pop the instruction pointer from the internal stack. Add Const bytes to the instruction pointer. If returning from an application trap the status register is not popped from the stack.

Execution Units: Branch

Exceptions: none

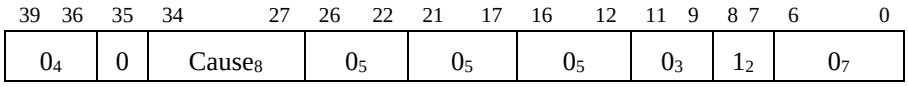
Notes:

SYS – SYSTEM CALL

Description:

This is an alternate mnemonic for the CHK instruction. Perform a system call. Interrupts are disabled. The instruction pointer is reset to the contents of the vector loaded from tvec[3] plus eight times the cause code and instructions begin executing. There should be a jump instruction placed at the break vector location. The address of the instruction following the SYS instruction is pushed onto an internal stack.

Instruction Format: CHK



Operation:

PUSH SR onto internal stack

PUSH IP + 5 onto internal stack

IP = tvec[3]

Execution Units: Branch

Clock Cycles:

Exceptions: none

Notes:

STOP – STOP PROCESSOR

Description:

The STOP instruction waits for an external interrupt to occur before proceeding. While waiting for the interrupt, the processor clock is slowed down or stopped placing the processor in a lower power mode. A sixteen-bit constant is provided for the CPU's stop instruction.

Instruction Format: STOP

39 38	37 35	34 13	12 7	6 0
Fmt_2	Pr_3	$Immediate_{21..0}$	Rt_6	113_7

Clock Cycles: 1 (if no exception present)

Execution Units: Branch

TRAP – TRAP

Description:

Execute trap. The data field is loaded into the specified target register, Rt. The trap number to execute comes from the contents of register Ra or an immediate value encoded in the instruction. The trap number must be between 1 and 511. Trap numbers below 64 are reserved for the system. Trap numbers 64 and above may be used by applications.

Traps below 64 will use the kernel vector base register to lookup the location of the service routine. Traps above 64 will use the application vector base register to lookup the location of the service routine.

Trap routines should return using an RTE instruction.

Instruction Format:

TRAP Rt,Ra,#Imm$_{16}$

39 38	37 35	34 33	32 22	21 19	18 13	12 7	6 0
Fmt_2	Pr_3	0_2	$Immediate_{10..0}$	$\sim_3$	Ra_6	Rt_6	0_7

TRAP Rt, #Vec, #Data

39 38	37 35	34 33	32 22	21 13	12 7	6 0
Fmt_2	Pr_3	1_2	$Immediate_{10..0}$	Vec_9	Rt_6	0_7

Clock Cycles: 1

Operation:

The program counter and the status register are pushed on an internal stack. Next the vector is fetched from the exception vector table and jumped to.

ENTER – ENTER ROUTINE

Description:

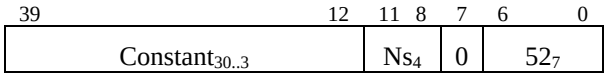
This instruction is used for subroutine linkage at entrance into a subroutine. First it pushes the frame pointer and return address onto the stack, next the stack pointer is loaded into the frame pointer, and finally the stack space is allocated. This instruction is code dense, replacing eight other instructions with a single instruction.

A maximum of 2GB may be allocated on the stack. An immediate postfix may not be used with this instruction. The stack and frame pointers are assumed to be r63 and r62 respectively.

Note that the constant must be a negative number and a multiple of eight.

Note that the instruction reserves room for two words in addition to the return address and frame pointer. One use for the extra words may to store exception handling information.

Integer Instruction Format: RI



Operation:

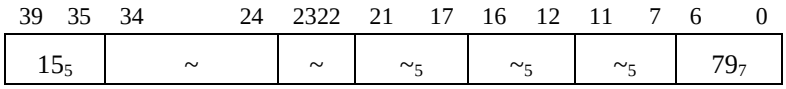
```
SP = SP - 32
Memory[SP] = FP
Memory16[SP] = LR0
Memory32[SP] = 0          ; zero out catch handler address
Memory48[SP] = 0
FP = SP
SP = SP – Ns * 16
Memory[SP] = S0
Memory16[SP] = S1
…
Memory((Ns-1)*16) = S[Ns-1]
SP = SP + constant
```

LDCTX – LOAD CONTEXT

Description:

This instruction loads all the general-purpose registers from the context block including the stack pointer but not including r0. The context block address is specified by the CTX CSR register.

Instruction Format: LSCTX



Operation:

R1 = Mem[CTX+1*8]

R2 = Mem[CTX+2*8]

R3 = Mem[CTX+3*8]

…

R31 = Mem[CTX+31*8]

Clock Cycles: 1 + 31 memory read accesses.

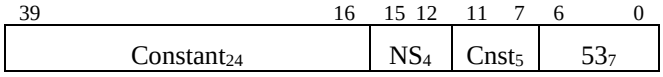
This instruction can take hundreds of clock cycles to complete. For example, if memory access takes eight clocks per access then this instruction will take 249 clock cycles. Note that loading registers and storing them may not take the same length of time.

LEAVE – LEAVE ROUTINE

Description:

This instruction is used for subroutine linkage at exit from a subroutine. It reverses the operations performed by ENTER. First is pops the specified number of callee saved registers from the stack. Next it moves the frame pointer to the stack pointer deallocating any stack memory allocations. Next the frame pointer and return address are popped off the stack. The stack pointer is adjusted by the amount specified in the instruction. Then a jump is made to the return address. This instruction is code dense, replacing between six and sixteen other instructions with a single instruction. The stack pointer adjustment is multiplied by eight keeping the stack pointer word aligned. A six-bit constant is added to the link register to form the return address. This allows returning up to 64 bytes past the normal return address.

Instruction Format: LEAVE



Operation:

SP = FP – NS*16

If (NS > 0) S0 = Memory[SP]

If (NS > 1) S1 = Memory16[SP]

…

If (NS > 9) S9 = Memory144[SP]

SP = FP

FP = Memory[SP]

LR0 = Memory8[SP]

SP = SP + 64 + $Constant_{24}$ * 8

PC = LR0 + $Cnst_5$

POP – POP REGISTERS FROM STACK

Description:

This instruction pops up to six registers from the stack. Note ‘N’ may only vary between one and six.

Instruction Format: POP

39 37	36 32	31 27	26 22	21 17	16 12	11 7	6 0
N_3	Re_5	Rd_5	Rc_5	Rb_5	Ra_5	Rt_5	55_7

Operation:

If (N > 0) Rt = Mem[SP]

If (N > 1) Ra = Mem[SP+8]

If (N > 2) Rb = Mem[SP+16]

If (N > 3) Rc = Mem[SP+24]

If (N > 4) Rd = Mem[SP+32]

If (N > 5) Re = Mem[SP+40]

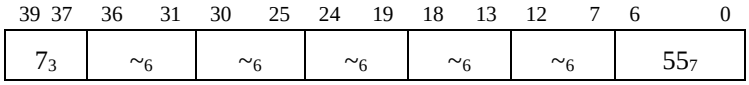
SP = SP + N * 8

POPA – POP ALL REGISTERS FROM STACK

Description:

This instruction pops all the general-purpose registers from the stack including the stack pointer but not including r0. The stack pointer is the last register popped, Its' value should be the current top of stack if the PUSHA instruction was used to push all the registers.

Instruction Format: POP



Operation:

R1 = Mem[SP]

R2 = Mem[SP+1*16]

R3 = Mem[SP+2*16]

…

R63 = Mem[SP+62*16]

SP = SP + 63 * 16

Clock Cycles: 1 + 63 memory read accesses.

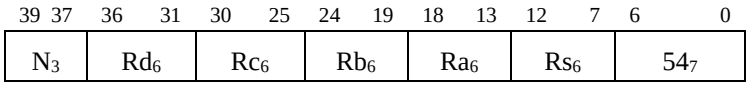
This instruction can take hundreds of clock cycles to complete. For example, if memory access takes eight clocks per access then this instruction will take 505 clock cycles.

PUSH – PUSH REGISTERS ON STACK

Description:

This instruction pushes up to five registers onto the stack. ‘N’ encodes the register count, 1 to 5.

Instruction Format: PUSH



Operation:

SP = SP – N * 16

if (N > 4) $Memory_{16}$[SP+(N-4)*16] = Rd

if (N > 3) $Memory_{16}$[SP+(N-3)*16] = Rc

if (N > 2) $Memory_{16}$[SP+(N-2)*16] = Rb

if (N > 1) $Memory_{16}$[SP+(N-1)*16] = Ra

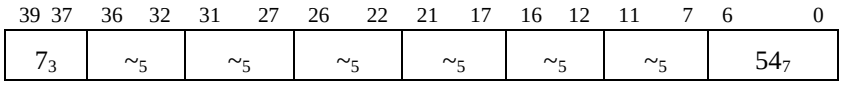
if (N > 0) $Memory_{16}$[SP+N*16] = Rs

PUSHA – PUSH ALL REGISTERS ON STACK

Description:

This instruction pushes all the general-purpose registers onto the current stack including the stack pointer, but not register r0.

Instruction Format: PUSH



Operation:

SP = SP – 31 * 8

…

$Memory_8$[SP] = r1

$Memory_8$[SP+1*8] = r2

…

$Memory_8$[SP+29*8] = r30

$Memory_8$[SP+30*8] = r31

Clock Cycles: 1 + 31 memory write accesses

This instruction can take hundreds of clock cycles to complete. For example, if memory access takes eight clocks per access then this instruction will take 249 clock cycles.

STCTX – STORE CONTEXT

Description:

This instruction stores all the general-purpose registers including the stack pointer, but not register r0, into the context block specified by the CTX CSR register..

Instruction Format: LSCTX

39 35	34 24	2322	21 17	16 12	11 7	6 0
7_5	$\sim_{11}$	$\sim_2$	$\sim_5$	$\sim_5$	$\sim_5$	87_7

Operation:

$Memory_8[CTX+1*8] = r1$

$Memory_8[CTX+2*8] = r2$

…

$Memory_8[CTX+30*8] = r30$

$Memory_8[CTX+31*8] = r31$

Clock Cycles: 1 + 31 memory write accesses

This instruction can take hundreds of clock cycles to complete. For example, if memory access takes eight clocks per access then this instruction will take 249 clock cycles.

ATOM

Description:

Treat the following sequence of instructions as an “atom”. The instruction sequence is executed with interrupts set to the specified mask level. Interrupts may be disabled for up to eleven instructions. The non-maskable interrupt may not be masked.

The 33-bit mask is broken into eleven three-bit interrupt level numbers. Bit 7 to 9 represent the interrupt level for the first instruction, bits 10 to 12 for the second and so on.

Note that since the processor fetches instructions in groups the mask effectively applies to the group. The mask guarantees that at least as many instructions as specified will be masked, but more may be masked depending on group boundaries.

Instruction Format: ATOM

39 7	6 0
$Mask_{33}$	122_7

Modifier Scope	Mask Bit	
	0 to 2	Instruction zero (always 7)
	3 to 5	Instruction one
	6 to 8	Instruction two
	9 to 11	Instruction three
	12 to 14	Instruction four
	15 to 17	Instruction five
	18 to 20	Instruction six
	21 to 23	Instruction seven
	24 to 27	Instruction eight
	28 to 30	Instruction nine
	31 to 33	Instruction ten

Assembler Syntax:

Example:

```
ATOM “777777”
LOAD a0,[a3]
SLT t0,a0,a1
PRED t0,”TTF”
```

```
STORE a2,[a3]
LDI a0,1
LDI a0,0
```

```
ATOM “6666”
LOAD a1,[a3]
ADD t0,a0,a1
MOV a0,a1
STORE t0,[a3]
```

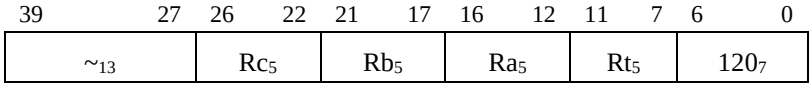
QFEXT

Description:

This modifier extends the register selection for quad precision floating-point operations. Quad precision operations need to use register pairs to contain a quad precision value. The QFEXT modifier specifies the registers used to contain bits 64 to 127 of the quad precision values.

Quad precision values are calculated using the QFEXT modifier before the quad precision instruction.

Instruction Format: QFEXT



PRED

Description:

Apply the predicate to following instructions according to a bit mask. The predicate may be applied to a maximum of eight instructions. The PRED instruction may be applied to vector instructions and act to mask off operation of specific lanes of the vector. If the ‘Z’ bit is set, target register elements are set to zero if not masked. Each byte of the predicate register contains the mask bits for the corresponding instruction.

Instruction Format: PRED

39	38	37 — 22	21 — 17	16 — 12	11 — 7	6 — 0
Z	~	$Mask_{15..0}$	$\sim_5$	Rp_5	$\sim_5$	121_7

	Mask Bit		Rp_6 Bits Tested
Pred Modifier Scope	0,1	Instruction zero	0 to 7
	2,3	Instruction one	8 to 15
	4,5	Instruction two	16 to 23
	6,7	Instruction three	24 to 31
	8,9	Instruction four	32 to 39
	10,11	Instruction five	40 to 47
	12,13	Instruction six	48 to 55
	14,15	Instruction seven	56 to 63

Mask Bit	Meaning
00	Always execute (ignore predicate)
01	Execute only if predicate bit is true
10	Execute only if predicate bit is false
11	Always execute (ignore predicate)

Assembler Syntax:

After the instruction mnemonic the register containing the predicate flags is specified. Next a character string containing ‘T’ for True, ‘F’ for false, or ‘I’ for ignore for the next five instructions is present.

Example:

```
PRED r2,”TIFIIIII”
  ; execute one if true, ignore one, next execute if false, one after always execute
MUL r3,r4,r5          ; executes if True, all regs are vecs
ADD r6,r3,r7          ; always executes, r3 is scalar, others are vecs
```

```
ADD r6,r6,#1234     ; executes if FALSE, vector regs
DIV r3,r4,r5        ; always executes, scalar regs for all three
```

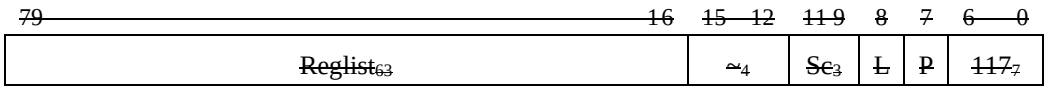
REGS – REGISTERS LIST

Description:

This instruction modifier specifies additional operands for the next instruction. When applied to a load or store operation it causes a multiple register load or store to be performed. The 'P' field of the instruction indicates whether to pack (0) or skip (1) over data addresses when performing the load or store operation. The 'L' field should be set for a load operation and clear for a store operation. The 'Sc' field indicates the indexing scale to use.

Bit 16 to 79 of the instruction correspond to registers 0 to 63.

Instruction Format:



Assembler Syntax:

Example:

ROUND

Description:

Set the rounding mode for following eight instructions. Note that postfixes do not count as instructions.

Instruction Format: ATOM

39 38	37 35	34 31	30 7	6 0
Fmt_2	Pr_3	$\sim_4$	$Mask_{24}$	116_7

Scope Modifier	Mask Bit	
	0 to 2	Instruction zero
	3 to 5	Instruction one
	6 to 8	Instruction two
	9 to 11	Instruction three
	12 to 14	Instruction four
	15 to 17	Instruction five
	18 to 20	Instruction six
	21 to 23	Instruction seven

BINARY FLOAT ROUNDING MODES

Rm3	Rounding Mode
000	Round to nearest ties to even
001	Round to zero (truncate)
010	Round towards plus infinity
011	Round towards minus infinity
100	Round to nearest ties away from zero
101	Reserved
110	Reserved
111	Use rounding mode in float control register

Assembler Syntax:

Example:

OPCODE MAPS

QUPLS ROOT OPCODE

	0	1	2	3	4	5	6	7
0x	0 CHK CHKCPL	1 {ZSxxI}	2 {R3}	3	4 ADDI	5 SUBFI	6 MULI	7 CSR
	8 ANDI	9 ORI	10 EORI	11 CMPI	12	13 DIVI	14 MULUI	15 MOV
1x	16 {FLT}	17 {DFLT}	18 {PST}	19 CMPUI	20 {VZSxxI}	21 DIVUI	22 {VFLT}	23 {VSFLT}
	24 VANDI	25 VORI	26 VEORI	27 VCMPI	28 VADDI	29 VDIVI	30 VMULI	31
2x	32 BRA BSR	33 DBRA	34	35 RET RTE JMPX	36 JMP JSR	37	38 {R3V}	39 {R3VS}
	40 BccU	41 Bcc	42	43 DFBcc	44 FBccH	45 FBccS	46 FBcc / FBccD	47 FBccQ
3x	48 VADDSI	49 ADDSI	50 ANDSI	51 ORSI	52 ENTER	53 LEAVE	54 PUSH PUSHA PUSHV	55 POP POPA POPV
	56 VANDSI	57 LDAX	58 AIPSI	59 EORSI	60 VORSI	61 VEORSI	62	63
4x	64 LDB	65 LDBU	66 LDW	67 LDWU	68 LDT	69 LDTU	70 LDO	71 LDOU
	72 LDH	73 LDV	74	75 CACHE	76 PLDS	77	78	79 {LDX} LDCTX
5x	80 STB	81 STW	82 STT	83 STO	84 STH	85 STV	86 STPTR	87 {STX} STCTX
	88 {SHIFT}	89 BLEND	90 {VSHIFT}	91	92 AMO	93 CAS	94	95
6x	96	97	98	99	100	101	102	103
	104 {VSDFLT}	105 {VDFLT}	106 {VPST}	107 {VSPST}	108 BFND	109 BCMP	110 BSET	111 BMOV
7x	112 REX	113 STOP	114 FENCE	115 PFI	116 ROUND	117	118	119
	120 QFEXT	121 PRED	122 ATOM	123	124	125	126	127 NOP

{R1} OPERATIONS

	0	1	2	3	4	5	6	7
0x	0 CNTLZ	1 CNTLO	2 CNTPOP	3 ABS	4 SQRT	5 REVBIT	6 CNTTZ	7 NOT
	8 NNA_TRIG	9 NNA_STAT	10 NNA_MFACT	11	12	13	14 SM3P0	15 SM3P1
1x	16	17	18 AES64DS	19 AES64DSM	20 AES64ES	21 AES64ESM	22 AES64IM	23
	24 SHA256 SIG0	25 SHA256 SIG1	26 SHA256 SUM0	27 SHA256 SUM1	28 SHA512 SIG0	29 SHA512 SIG1	30 SHA512 SUM0	31 SHA512 SUM1

{R3} OPERATIONS

	0	1	2	3	4	5	6	7
2	0 AND	1 OR	2 EOR	3 CMP	4 ADD	5 SUB	6 CMPU	7 CPUID
	8 NAND	9 NOR	10 ENOR	11 CMOVZ	12 CMOVNZ	13	14	15
2	16 MUL	17 DIV	18 {MINMAX}	19 MULU	20 DIVU	21 MULSU	22 DIVSU	23
	24 MULW	25 MOD	26 {R1}	27 MULUW	28 MODU	29 MULSUW	30 MODSU	31
2	32 PTRDIF	33 MUX	34 BMM	35 BMAP	36 DIF	37 CHARNDX	38 CHARNDX	39 CHARNDX
	40 NNA MTWT	41 NNA MTIN	42 NNA MTBIAS	43 NNA MTFB	44 NNA MTMC	45 NNA MTBC	46	47
2	48 V2BITS	49 BITS2V	50 VEX	51 VEINS	52 VGNDX	53	54 VSHLV	55 VSHRV
	56 V2BITSP	57 PBITS2V	58	59	60	61	62 VSHLVI	63 VSHRVI
2	64 AES64K1I	65 AES64KS2	66 SM4ED	67 SM4KS	68	69	70 CLMUL	71
	72	73	74	75	76	77	78	79
2	80 SEQ	81 SNE	82 SLT	83 SLE	84 SLTU	85 SLEU	86	87 DIVMOD
	88 CLR	89 SET	90 EXT	91 EXTU	92 DEP	93 COM	94	95 DIVMODU
2	96 SEQ	97 SNE	98 SLT	99 SLE	100 SLTU	101 SLEU	102	103
	104 CLRI	105 SETI	106 EXTI	107 EXTUI	108 DEPI	109 COMI	110	111
2	112 ZSEQ	113 ZSNE	114 ZSLT	115 ZSLE	116 ZSLTU	117 ZSLEU	118	119
	120 ZSEQ	121 ZSNE	122 ZSLT	123 ZSLE	124 ZSLTU	125 ZSLEU	126	127 MVVR

{ZSXXI} OPERATIONS

	0	1	2	3	4	5	6	7
1	0 ZSEQI	1 ZSNEI	2 ZSLTI	3 ZSLEI	4 ZSLTUI	5 ZSLEUI	6	7
	8	9	10	11	12	13	14	15

			ZSGTI	ZSGEI	ZSGTUI	ZSGEUI		
1	16 SEQI	17 SNEI	18 SLTI	19 SLEI	20 SLTUI	21 SLEUI	22	23
	24	25	26 SGTI	27 SGEI	28 SGTUI	29 SGEUI	30	31

{FLT3} OPERATIONS

	0	1	2	3	4	5	6	7
16	0 FMA	1 FMS	2 FNMA	3 FNMS	4 VFMA	5 VFMS	6 VFNMA	7 VFNMS
	8 **{FLT2}**	9	10	11	12 **{VFLT2}**	13 **{VSFLT2}**	14 {VFLT2}	15

{FLT2} OPERATIONS

	0	1	2	3	4	5	6	7
16	0 FSCALEB	1 **{FLT1}**	2 FMIN	3 FMAX	4 FADD	5 FSUB	6 FMUL	7 FDIV
	8 FSEQ	9 FSNE	10 FSLT	11 FSLE	12 {FTRIG}	13 FCMP	14 FNXT	15 FREM
	16 FSGNJ	17 FSGNJN	18 FSGNJX	19	20	21	22	23
	24	25	26	27	28	29	30 FNMUL	31

{FLT1} OPERATIONS

	0	1	2	3	4	5	6	7
0x	0 FABS	1 FNEG	2 FTOI	3 ITOF	4 FCONST	5	6 FSIGN	7 FSIG
	8 FSQRT	9 FCVTS2D	10 FCVTS2Q	11 FCVTD2Q	12 FCVTH2S	13 FCVTH2D	14 ISNAN	15 FINITE
1x	16 FCVTQ2H	17 FCVTQ2S	18 FCVTQ2D	19	20 FCVTH2Q	21 FTRUNC	22 FRSQRTE	23 FRES
	24	25 FCVTD2S	26	27	28	29	30 FCLASS	31

	0	1	2	3	4	5	6	7
1x	0 FSIN	1 FCOS	2 FTAN					
	8	9	10 FATAN					
2x	16 FSIGMOID							
	24							

{DFLT3} OPERATIONS

	0	1	2	3	4	5	6	7
17	0 FMA	1 FMS	2 FNMA	3 FNMS	4 VFMA	5 VFMS	6 VFNMA	7 VFNMS
	8 **{DFLT2}**	9	10	11	12 **{VDFLT2}**	13 **{VSFLT2}**	14	15

{FLT2} OPERATIONS

	0	1	2	3	4	5	6	7
17	0 DFSCALEB	1 **{DFLT1}**	2 DFMIN	3 DFMAX	4 DFADD	5 DFSUB	6 DFMUL	7 DFDIV
	8 DFSEQ	9 DFSNE	10 DFSLT	11 DFSLE	12	13 DFCMP	14 DFNXT	15 DFREM
	16 DFSGNJ	17 DFSGNJN	18 DFSGNJX	19	20	21	22	23
	24	25	26	27	28	29	30 FNMUL	31

{LDX} – INDEXED LOADS

	0	1	2	3	4	5	6	7
0x	0 LDBX	1 LDBUX	2 LDWX	3 LDWUX	4 LDTX	5 LDTUX	6 LDOX	7 LDOUX
	8 LDHX	9 LDMX	10 LDAX	11 CACHEX	12 PLDSX	13 PLDDX	14	15 LDCTX
1x	16	17	18 FLDHX	19	20 FLDSX	21	22 FLDDX	23
	24 FLDQX	25 DFLDSX	26 DFLDDX	27 DFLDQX	28 BFNDX	29	30	31 CAS
2x	32	33	34	35	36	37	38	39
	40	41	42	43	44	45	46	47
3x	48	49	50	51	52	53	54	55
	56	57	58	59	60	61	62	63

{STX} – INDEXED STORES

	0	1	2	3	4	5	6	7
0x	0 STBX	1 STWX	2 STTX	3 STOX	4 STHX	5 STMX	6 STPTRX	7 STCTX
	8	9 FSTHX	10 FSTSX	11 FSTDX	12 FSTQX	13	14 PSTSX	15 PSTDX
1x	16 STBX	17	18 DFSTSX	19 DFSTDX	20 DFSTQX	21	22	23
	24	25	26	27	28	29	30	31
2x – 3x	FAF							

{AMO} – ATOMIC MEMORY OPS

	0	1	2	3	4	5	6	7
92	0 AMOADD	1 AMOAND	2 AMOOR	3 AMOEOR	4 AMOMIN	5 AMOMAX	6 AMOSWAP	7
	8 AMOASL	9 AMOLSR	10 AMOROL	11 AMOROR	12 AMOMINU	13 AMOMAXU	14	15

{PR} THOR2024 PREDICATE OPERATIONS

	0	1	2	3	4	5	6	7
0x	0 PRASL	1 PRROL	2 PRLSR	3 PRROR	4 ADD	5 SUB	6	7
	8 PRAND	9 PROR	10 PREOR	11 MFPR	12 MTPR	13 PRCNTPOP	14 PRFIRST	15 PRLAST
1x	16 PRANDN	17	18	19	20 PRLDI	21	22	23
	24	25	26	27	28	29	30	31

{R3} THOR2024 R3 OPERATIONS

	0	1	2	3	4	5	6	7
0x	0 MUX	1 PTRDIF	2 MIN3	3 MAX3	4 CMOVNZ	5 CMOVZ	6	7

{EX} EXCEPTION INSTRUCTIONS

	0	1	2	3	4	5	6	7
2	0 IRQ	1	2 FTX	3 FCX	4 FDX	5 FEX	6	7 REX
	8	9	10	11	12	13	14	15

PIC – PROGRAMMABLE INTERRUPT CONTROLLER

OVERVIEW

The programmable interrupt controller manages interrupt sources in the system and presents an interrupt signal to the cpu. The PIC may be used in a multi-CPU system as a shared interrupt controller. The PIC can guide the interrupt to the specified core. If two interrupts occur at the same time the controller resolves which interrupt the cpu sees. While the CPU's interrupt input is only level sensitive the PIC may process interrupts that are either level or edge sensitive. the PIC is a 32-bit I/O device.

SYSTEM USAGE

There is just a single interrupt controller in the system. It supports 31 different interrupt sources plus a non-maskable interrupt source.

The PIC is located at an address determined by BAR0 in the configuration space.

PRIORITY RESOLUTION

Interrupts have a fixed priority relationship with interrupt #1 having the highest priority and interrupt #31 the lowest. Note that interrupt priorities are only effective when two interrupts occur at the same time.

CONFIG SPACE

A 256-byte config space is supported. Most of the config space is unused. The only configuration is for the I/O address of the register set.

Regno	Width	R/W	Moniker	Description		
000	32	RO	REG_ID	Vendor and device ID		
004	32	R/W				
008	32	RO				
00C	32	R/W				
010	32	R/W	REG_BAR0	Base Address Register		
014	32	R/W	REG_BAR1	Base Address Register		
018	32	R/W	REG_BAR2	Base Address Register		
01C	32	R/W	REG_BAR3	Base Address Register		
020	32	R/W	REG_BAR4	Base Address Register		
024	32	R/W	REG_BAR5	Base Address Register		
028	32	R/W				
02C	32	RO		Subsystem ID		
030	32	R/W		Expansion ROM address		
034	32	RO				
038	32	R/W		Reserved		

03C	32	R/W		Interrupt		
040 to 0FF	32	R/W		Capabilities area		

REG_BAR0 defaults to $FEE20001 which is used to specify the address of the controller's registers in the I/O address space.

The controller will respond with a memory size request of 0MB (0xFFFFFFFF) when BAR0 is written with all ones. The controller contains its own dedicated memory and does not require memory allocated from the system.

Parameters

CFG_BUS defaults to zero

CFG_DEVICE defaults to six

CFG_FUNC defaults to zero

Config parameters must be set correctly. CFG device and vendors default to zero.

REGISTERS

The PIC contains 40 registers spread out through a 256 byte I/O region. All registers are 32-bit and only 32-bit accessible. There are two different means to control interrupt sources. One is a set of registers that works with bit masks enabling control of multiple interrupt sources at the same time using single I/O accesses. The other is a set of control registers, one for each interrupt source, allowing control of interrupts on a source-by-source basis.

Regno	Access	Moniker	Purpose
00	R	CAUSE	interrupt cause code for currently interrupting source
04	RW	RE	request enable, a 1 bit indicates interrupt requesting is enabled for that interrupt, a 0 bit indicates the interrupt request is disabled.
08	W	ID	Disables interrupt identified by low order five data bits.
0C	W	IE	enables interrupt identified by low order five data bits
10			reserved
14	W	RSTE	resets the edge-sense circuit for edge sensitive interrupts, 1 bit for each interrupt source. This register has no effect on level sensitive sources. This register automatically resets to zero.
18	W	TRIG	software trigger of the interrupt specified by the low order five data bits.
20	W	ESL	The low bit for edge sensitivity selection. ESL and ESH combine to form a two bit select of the edge sensitivity.

ESH,EHL	Sensitivity
00	level sensitive interrupt
01	positive edge sensitive
10	negative edge sensitive

			11 either edge sensitive
24	W	ESH	The high bit for edge sensitivity selection
80	RW	CTRL0	control register for interrupt #0
84	RW	CTRL1	control register for interrupt #1
…		…	
FC	RW	CTRL31	control register for interrupt #31

CONTROL REGISTER

All the control registers are identical for all interrupt sources, so only the first control register is described here.

Bits

0 to 7	CAUSE	The cause code associated with the interrupt; this register is copied to the cause register when the interrupt is selected.
8 to 10	IRQ	This register determines which signal lines of the cpu are activated for the interrupt. Signal lines are typically used to resolve priority.
16	IE	This is the interrupt enable bit, 1 enables the interrupt, 0 disables it. This is the same bit reflected in the RE register.
17	ES	This bit controls edge sensitivity for the interrupt 0 = level, 1 = pos. edge sensitive. This same bit is present in the ESL register.
18		reserved
19	IRQAR	Respond to an IRQ Ack cycle
20 to 23		reserved
24 to 29	CORE	Core number to select for interrupt processing
30 to 31		reserved

PIT – PROGRAMMABLE INTERVAL TIMER

OVERVIEW

Many systems have at least one timer. The timing device may be built into the cpu, but it is frequently a separate component on its own. The programmable interval timer has many potential uses in the system. It can perform several different timing operations including pulse and waveform generation, along with measurements. While it is possible to manage timing events strictly through software it is quite challenging to perform in that manner. A hardware timer comes into play for the difficult to manage timing events. A hardware timer can supply precise timing. In the test system there are two groups of four timers. Timers are often grouped together in a single component. The PIT is a 64-bit peripheral. The PIT while powerful turns out to be one of the simpler peripherals in the system.

SYSTEM USAGE

One programmable timer component, which may include up 32 timers, is used to generate the system time slice interrupt and timing controls for system garbage collection. The second timer component is used to aid the paged memory management unit. There are free timing channels on the second timer component.

Each PIT is given a 64kB-byte memory range to respond to for I/O access. As is typical for I/O devices part of the address range is not decoded to conserve hardware.

PIT#1 is located at $FFFFFFFFFFEE4xxxx

PIT#2 is located at $FFFFFFFFFFEE5xxxx

CONFIG SPACE

A 256-byte config space is supported. Most of the config space is unused. The only configuration is for the I/O address of the register set and the interrupt line used.

Regno	Width	R/W	Moniker	Description		
000	32	RO	REG_ID	Vendor and device ID		
004	32	R/W				
008	32	RO				
00C	32	R/W				
010	32	R/W	REG_BAR0	Base Address Register		
014	32	R/W	REG_BAR1	Base Address Register		
018	32	R/W	REG_BAR2	Base Address Register		
01C	32	R/W	REG_BAR3	Base Address Register		
020	32	R/W	REG_BAR4	Base Address Register		
024	32	R/W	REG_BAR5	Base Address Register		
028	32	R/W				
02C	32	RO		Subsystem ID		
030	32	R/W		Expansion ROM address		
034	32	RO				

038	32	R/W		Reserved		
03C	32	R/W		Interrupt		
040 to 0FF	32	R/W		Capabilities area		

REG_BAR0 defaults to $FEE40001 which is used to specify the address of the controller's registers in the I/O address space. Note for additional groups of timers the REG_BAR0 must be changed to point to a different I/O address range. Note the core uses only bits determined by the address mask in the address range comparison. It is assumed that the I/O address select input, cs_io, will have bits 24 and above in its decode and that a 64kB page is required for the device, matching the MMU page size.

The controller will respond with a mask of 0x00FF0000 when BAR0 is written with all ones.

Parameters

CFG_BUS defaults to zero

CFG_DEVICE defaults to four

CFG_FUNC defaults to zero

CFG_ADDR_MASK defaults to 0x00FF0000

CFG_IRQ_LINE defaults to 29

Config parameters must be set correctly. CFG device and vendors default to zero.

PARAMETERS

NTIMER: This parameter controls the number of timers present. The default is eight. The maximum is 32.

BITS: This parameter controls the number of bits in the counters. The default is 48 bits. The maximum is 64.

PIT_ADDR: This parameter sets the I/O address that the PIT responds to. The default is $FEE40001.

PIT_ADDR_ALLOC: This parameter determines which bits of the address are significant during decoding. The default is $00FF0000 for an allocation of 64kB. To compute the address range allocation required, 'or' the value from the register with $FF000000, complement it then add 1.

REGISTERS

The PIT has 134 registers addressed as 64-bit I/O cells. It occupies 2048 consecutive I/O locations. All registers are read-write except for the current counts which are read-only. All registers all 64-bit accessible; all 64 bits must be read or written. Values written to registers do not take effect until the synchronization register is written.

Note the core may be configured to implement fewer timers in which case timers that are not implemented will read as zero and ignore writes. The core may also be configured to support fewer bits per count register in which case the unimplemented bits will read as zero and ignore writes.

Regno	Access	Moniker	Purpose
00	R	CC0	Current Count
08	RW	MC0	Max count
10	RW	OT0	On Time
18	RW	CTRL0	Control
20 to 7F8	…	…	Groups of four registers for timer #1 to #63
800	RW	USTAT	Underflow status
808	RZW	SYNC	Synchronization register
810	RW	IE	Interrupt enable
818	RW	TMP	Temporary register
820	RO	OSTAT	Output status
828	RW	GATE	Gate register
830	RZW	GATEON	Gate on register
838	RZW	GATEOFF	Gate off register

CONTROL REGISTER

This register contains bits controlling the overall operation of the timer.

Bit		Purpose
0	LD	setting this bit will load max count into current count, this bit automatically resets to zero.
1	CE	count enable, if 1 counting will be enabled, if 0 counting is disabled and the current count register holds its value. On counter underflow this bit will be reset to zero causing the count to halt unless auto-reload is set.
2	AR	auto-reload, if 1 the max count will automatically be reloaded into the current count register when it underflows.
3	XC	external clock, if 1 the counter is clocked by an external clock source. The external clock source must be of lower frequency than the clock supplied to the PIT. The PIT contains edge detectors on the external clock source and counting occurs on the detection of a positive edge on the clock source. This bit is forced to 0 for timers 4 to 31.
4	GE	gating enable, if 1 an external gate signal will also be required to be active high for the counter to count, otherwise if 0 the external gate is ignored. Gating the counter using the external gate may allow pulse-width measurement. This bit is forced to 0 for timers 4 to 31.
5 to 63	~	not used, reserved

CURRENT COUNT

This register reflects the current count value for the timer. The value in this register will change by counting downwards whenever a count signal is active. The current count may be automatically reloaded at underflow if the auto reload bit (bit #2) of the control byte is set. The current count may also be force loaded to the max count by setting the load bit (bit #0) of the counter control byte.

MAX COUNT

This register holds onto the maximum count for the timer. It is loaded by software and otherwise does not change. When the counter underflows the current count may be automatically reloaded from the max count register.

ON TIME

The on-time register determines the output pulse width of the timer. The timer output is low until the on-time value is reached, at which point the timer output switches high. The timer output remains high until the counter reaches zero at which point the timer output is reset back to zero. So, the on time reflects the length of time the timer output is high. The timer output is low for max count minus the on-time clock cycles.

UNDERFLOW STATUS

The underflow status register contains a record of which timers underflowed.

Writing the underflow register clears the underflows and disable further interrupts where bits are set in the incoming data. Interrupt processing should read the underflow register to determine which timers underflowed, then write back the value to the underflow register.

SYNCHRONIZATION REGISTER

The synchronization register allows all the timers to be updated simultaneously. Values written to timer registers do not take effect until the synchronization register is written. The synchronization register must be written with a '1' bit in the bit position corresponding to the timer to update. For instance, writing all one's to the sync register will cause all timers to be updated. The synchronization register is write-only and reads as zero.

INTERRUPT ENABLE REGISTER

Each bit of the interrupt enable register enables the interrupt for the corresponding timer. Interrupts must also be globally enabled by the interrupt enable bit in the config space for interrupts to occur. A '1' bit enables the interrupt, a '0' bit value disables it.

TEMPORARY REGISTER

This is merely a register that may be used to hold values temporarily.

OUTPUT STATUS

The output status register reflects the current status of the timers output (high or low). This register is read-only.

GATE REGISTER

The internal gate register is used to temporarily halt or resume counting for the timer corresponding to the bit position of this register. Writing a value to this register will turn on all timers where there is a '1' bit in the value and turn off all timers where there is a '0' bit in the value.

GATE ON REGISTER

The internal gate 'on' register is used to resume counting for the timer corresponding to the bit position of this register. Writing a value to this register will turn on all timers where there is a '1' bit in the value. Where there is a '0' in the value the timer will not be affected. This register reads as zero.

GATE OFF REGISTER

The internal gate 'off' register is used to halt counting for the timer corresponding to the bit position of this register. Writing a value to this register will turn off all timers where there is a '1' bit in the value. Where there is a '0' in the value the timer will not be affected. This register reads as zero.

PROGRAMMING

The PIT is a memory mapped i/o device. The PIT is programmed using 64-bit load and store instructions (LDO and STO). Byte loads and stores (LDB, STB) may be used for control register access. It must reside in the non-cached address space of the system.

INTERRUPTS

The core is configured use interrupt signal #29 by default. This may be changed with the CFG_IRQ_LINE parameter. Interrupts may be globally disabled by writing the interrupt disable bit in the config space with a '1'. Individual interrupts may be enabled or disabled by the setting of the interrupt enable register in the I/O space.

GLOSSARY

ABI

An acronym for application binary interface. An ABI is a description of the interface between software and hardware, or between software modules. It includes things like the expected register usage by the compiler. Some registers hardware has specific requirements for are noted in the ABI, for instance r0 may always be zero or it may be a usable register. The stack pointer may need to be a specific register. A good ABI is an aid to guaranteeing that software works when coming from multiple sources.

AMO

AMO stands for atomic memory operation. An atomic memory operation typically reads then writes to memory in a fashion that may not be interrupted by another processor. Some examples of AMO operations are swap, add, and, and or. AMO operations are typically passed from the CPU to the memory controller and the memory controller performs the operation.

ASSEMBLER

A program that translates mnemonics and operands into machine code OR a low-level language used by programmers to conveniently translate programs into machine code. Compilers are often capable of generating assembler code as an output.

ATC

ATC stands for address translation cache. This buffer is used to cache address translations for fast memory access in a system with an mmu capable of performing address translations. The address translation cache is more commonly known as the TLB.

BASE POINTER

An alternate term for frame pointer. The frame or base pointer is used by high-level languages to access variables on the stack.

BURST ACCESS

A burst access is several bus accesses that occur rapidly in a row in a known sequence. If hardware supports burst access the cycle time for access to the device is drastically reduced. For instance, dynamic RAM memory access is fast for sequential burst access, and somewhat slower for random access.

BTB

An acronym for Branch Target Buffer. The branch target buffer is used to improve the performance of a processing core. The BTB is a table that stores the branch target from previously executed branch instructions. A typical table may contain 1024 entries. The table is typically

indexed by part of the branch address. Since the target address of a branch type instruction may not be known at fetch time, the address is speculated to be the address in the branch target buffer. This allows the machine to fetch instructions in a continuous fashion without pipeline bubbles. In many cases the calculated branch address from a previously executed instruction remains the same the next time the same instruction is executed. If the address from the BTB turns out to be incorrect, then the machine will have to flush the instruction queue or pipeline and begin fetching instructions from the correct address.

CARD MEMORY

A card memory is a memory reserved to record the location of pointer stores in a garbage collection system. The card memory is much smaller than main memory; there may be card memory entry for a block of main memory addresses. Card memory covers memory in 128 to 512-byte sized blocks. Usually, a byte is dedicated to record the pointer store status even though a bit would be adequate, for performance reasons. The location of card memory to update is found by shifting the pointer value to the right some number of bits (7 to 9 bits) and then adding the base address of the table. The update to the card memory needs to be done with interrupts disabled.

COMMIT

As in commit stage of processor. This is the stage where the processor is dedicated or committed to performing the operation. There are no prior outstanding exceptions or flow control changes to prevent the instruction from executing. The instruction may execute in the commit stage, but registers and memory are not updated until the retire stage of the processor.

DECIMAL FLOATING POINT

Floating point numbers encoded specially to allow processing as decimal numbers. Decimal floating point allows processing every-day decimal numbers rounding in the same manner as would be done by hand.

DECODE

The stage in a processor where instructions are decoded or broken up into simpler control signals. For instance, there is often a register file write signal that must be decoded from instructions that update the register file.

DIADIC

As in diadic instruction. An instruction with two operands.

ENDIAN

Computing machines are often referred to as big endian or little endian. The endian of the machine has to do with the order bits and bytes are labeled. Little endian machines label bits from right to left with the lowest bit at the right. Big endian machines label bits from left to right with the lowest numbered bit at the left.

FIFO

An acronym standing for ‘first-in first-out’. Fifo memories are used to aid data transfer when the rate of data exchange may have momentary differences. Usually when fifos transfer data the

average data rate for input and output is the same. Data is stored in a buffer in order then retrieved from the buffer in order. Uarts often contain fifos.

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 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.

FLOATING POINT

A means of encoding numbers into binary code to allow processing. Floating point numbers have a range within which numbers may be processed, outside of this range the number will be marked as infinity or zero. The range is usually large enough that it is not a concern for most programs.

FRAME POINTER

A pointer to the current working area on the stack for a function. Local variables and parameters may be accessed relative to the frame pointer. As a program progresses a series of "frames" may build up on the stack. In many cases the frame pointer may be omitted, and the stack pointer used for references instead. Often a register from the general register file is used as a frame pointer.

HDL

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

HLL

An acronym that stands for "High Level Language"

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.

INSTRUCTION POINTERS

A processor register dedicated to addressing instructions in memory. It is also often called a program counter. The program counter got its name because it usually increments (or counts) automatically after an instruction is fetched. In early machines in some rare cases the program counter did not count in a sequential binary fashion, but instead used other forms of a counter such as a grey counter or linear feedback shift register. In some machines the program counter addresses bundles of instructions rather than individual instructions. This is common with some stack machines where multiple instructions are packed into a memory word.

INSTRUCTION PREFIX

An instruction prefix applies to the following instruction to modify its operation. An instruction prefix may be used to add more bits to a following immediate constant, or to add additional register fields for the instruction. The prefix essentially extends the number of bits available to encode instructions. An instruction prefix usually locks out interrupts between the prefix and following instruction.

INSTRUCTION MODIFIER

An instruction modifier is similar to an instruction prefix except that the modifier may apply to multiple following 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. RTF64 falls somewhere in between with features of both RISC and CISC architectures.

JIT

An acronym standing for Just-In-Time. JIT compilers typically compile segments of a program just before usage, and hence are called JIT compilers.

KEYED MEMORY

A memory system that has a key associated with each page to protect access to the page. A process must have a matching key in its key list in order to access the memory page. The key is often 20 bits or larger. Keys for pages are usually cached in the processor for performance reasons. The key may be part of the paging tables.

LINEAR ADDRESS

A linear address is the resulting address from a virtual address after segmentation has been applied.

MACHINE CODE

A code that the processing machine is able execute. Machine code is lowest form of code used for processing and is not usually delt with by programmers except in debugging cases. While it is possible to assemble machine code by hand usually a tool called an assembler is used for this purpose.

MILLI-CODE

A short sequence of code that may be used to emulate a higher-level instruction. For instance, a garbage collection write barrier might be written as milli-code. Milli-code may use an alternate link register to return to obtain better performance.

MONADIC

An instruction with just a single operand.

OPCODE

A short form for operation code, a code that determines what operation the processor is going to perform. Instructions are typically made up of opcodes and operands.

OPERAND

The data that an opcode operates on, or the result produced by the operation. Operands are often located in registers. Inputs to an operation are referred to as source operands, the result of an operation is a destination operand.

PHYSICAL ADDRESS

A physical address is the final address seen by the memory system after both segmentation and paging have been applied to a virtual address. One can think of a physical address as one that is "physically" wired to the memory.

PHYSICAL MEMORY ATTRIBUTES (PMA)

Memory usually has several characteristics associated with it. In the memory system there may be several different types of memory, rom, static ram, dynamic ram, eeprom, memory mapped I/O devices, and others. Each type of memory device is likely to have different characteristics. These characteristics are called the physical memory attributes. Physical memory attributes are associated with address ranges that the memory is located in. There may be a hardware unit dedicated to verifying software is adhering to the attributes associated with the memory range. The hardware unit is called a physical memory attributes checker (PMA checker).

POSITS

An alternate representation of numbers.

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 its name because it usually increments (or counts) automatically after an instruction is fetched. In early machines in some rare cases the program counter did not count in a sequential binary fashion, but instead used other forms of a counter such as a grey counter or linear feedback shift register. In some machines the program counter addresses bundles of instructions rather than individual instructions. This is common with some stack machines where multiple instructions are packed into a memory word.

RAT

Anacronym for Register Alias Table. The RAT stores mappings of architectural registers to physical registers.

RETIRE

As in retire an instruction. This is the stage in processor in which the machine state is updated. Updates include the register file and memory. Buffers used for instruction storage are freed.

ROB

An acronym for ReOrder Buffer. The re-order buffer allows instructions to execute out of order yet update the machine's state in order by tracking instruction state and variables. In FT64 the re-order buffer is a circular queue with a head and tail pointers. Instructions at the head are committed if done to the machine's state then the head advanced. New instructions are queued at the buffer's tail as long as there is room in the queue. Instructions in the queue may be processed out of the order that they entered the queue in depending on the availability of resources (register values and functional units).

RSB

An acronym that stands for return stack buffer. A buffer of addresses used to predict the return address which increases processor performance. The RSB is usually small, typically 16 entries. When a return instruction is detected at time of fetch the RSB is accessed to determine the address of the next instruction to fetch. Predicting the return address allows the processing core to continuously fetch instructions in a speculative fashion without bubbles in the pipeline. The return address in the RSB may turn out to be detected as incorrect during execution of the return instruction, in which case the pipeline or instruction queue will need to be flushed and instructions fetched from the proper address.

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 by convention 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. Sometimes machines have multiple stack pointers for different purposes, but they all work on the idea of a stack. For instance, in Forth machines there are typically two stacks, one for data and one for return addresses.

TELESCOPIC MEMORY

A memory system composed of layers where each layer contains simplified data from the topmost layer downwards. At the topmost layer data is represented verbatim. At the bottom layer there may be only a single bit to represent the presence of data. Each layer of the telescopic memory uses far less memory than the layer above. A telescopic memory could be used in garbage collection systems. Normally however the extra overhead of updating multiple layers of memory is not warranted.

TLB

TLB stands for translation look-aside buffer. This buffer is used to store address translations for fast memory access in a system with an mmu capable of performing address translations.

TRACE MEMORY

A memory that traces instructions or data. As instructions are executed the address of the executing instruction is stored in a trace memory. The trace memory may then be dumped to allow debugging of software. The trace memory may compress the storage of addresses by storing branch status (taken or not taken) for consecutive branches rather than storing all addresses. It typically requires only a single bit to store the branch status. However, even when branches are traced, periodically the entire address of the program executing is stored. Often trace buffers support tracing thousands of instructions.

TRIADIC

An instruction with three operands.

VECTOR LENGTH (VL REGISTER)

The vector length register controls the maximum number of elements of a vector that are processed. The vector length register may not be set to a value greater than the number of elements supported by hardware. Vector registers often contain more elements than are required by program code. It would be wasteful to process all elements when only a few are needed. To improve the processing performance only the elements up to the vector length are examined.

VECTOR MASK (VM)

A vector mask is used to restrict which elements of a vector are processed during a vector operation. A one bit in a mask register enables the processing for that element, a zero bit disables it. The mask register is commonly set using a vector set operation.

VIRTUAL ADDRESS

The address before segmentation and paging has been applied. This is the primary type of address a program will work with. Different programs may use the same virtual address range without being concerned about data being overwritten by another program. Although the virtual address may be the same the final physical addresses used will be different.

WRITEBACK

A stage in a pipelined processing core where the machine state is updated. Values are ‘written back’ to the register file.

MISCELLANEOUS

REFERENCE MATERIAL

Below is a short list of some of the reading material the author has studied. The author has downloaded a fair number of documents on computer architecture from the web. Too many to list.

Modern Processor Design Fundamentals of Superscalar Processors by John Paul Shen, Mikko H. Lipasti. Waveland Press, Inc.

Computer Architecture A Quantitative Approach, Second Edition, by John L Hennessy & David Patterson, published by Morgan Kaufman Publishers, Inc. San Franciso, California is a good book on computer architecture. There is a newer edition of the book available.

Memory Systems Cache, DRAM, Disk by Bruce Jacob, Spencer W. Ng., David T. Wang, Samuel Rodriguez, Morgan Kaufman Publishers

PowerPC Microprocessor Developer's Guide, SAMS publishing. 201 West 103rd Street, Indianapolis, Indiana, 46290

80386/80486 Programming Guide by Ross P. Nelson, Microsoft Press

Programming the 286, C. Vieillefond, SYBEX, 2021 Challenger Drive #100, Alameda, CA 94501

Tech. Report UMD-SCA-2000-02 ENEE 446: Digital Computer Design — An Out-of-Order RiSC-16

Programming the 65C816, David Eyes and Ron Lichty, Western Design Centre Inc.

Microprocessor Manuals from Motorola, and Intel,

The SPARC Architecture Manual Version 8, SPARC International Inc, 535 Middlefield Road. Suite210 Menlo Park California, CA 94025

The SPARC Architecture Manual Version 9, SPARC International Inc, Sab Jose California, PTR Prentice Hall, Englewood Cliffs, New Jersey, 07632

The MMIX processor: 5

RISCV 2.0 Spec, Andrew Waterman, Yunsup Lee, David Patterson, Krste Asanovi´c CS Division, EECS Department, University of California, Berkeley {waterman|yunsup|pattrsn|krste}@eecs.berkeley.edu

The Garbage Collection Handbook, Richard Jones, Antony Hosking, Eliot Moss published by CRC Press 2012

RISC-V Cryptography Extensions Volume I Scalar & Entropy Source Instructions See github.com/riscv/riscv-crypto for more information.

TRADEMARKS

WISHBONE COMPATIBILITY DATASHEET

The Qupls core now uses the FTA bus which is not compatible with WISHBONE. Many signals serve a similar function to those on the WISHBONE bus so they are listed here. A bus bridge is required to interface FTA bus to WISHBONE as WISHBONE is a synchronous bus and FTA is asynchronous.

WISHBONE Datasheet WISHBONE SoC Architecture Specification, Revision B.3		
Description:	Specifications:	
General Description:	Central processing unit (CPU core)	
Supported Cycles:	MASTER, READ / WRITE MASTER, READ-MODIFY-WRITE MASTER, BLOCK READ / WRITE, BURST READ (FIXED ADDRESS)	
Data port, size: Data port, granularity: Data port, maximum operand size: Data transfer ordering: Data transfer sequencing	128 bit 8 bit 128 bit Little Endian any (undefined)	
Clock frequency constraints:	tm_clk_i must be >= 10MHz	
Supported signal list and cross reference to equivalent WISHBONE signals	Signal Name: Resp.ack_i Req.adr_o(31:0) clk_i resp.dat(127:0) req.dat(127:0) req.cyc req.stb req.wr req.sel(7:0)	WISHBONE Equiv. ACK_I ADR_O() CLK_I DAT_I() DAT_O() CYC_O STB_O WE_O SEL_O

	req.cti(2:0) req.bte(1:0)	CTI_O BTE_O
Special Requirements:		

FTA BUS

OVERVIEW

The FTA bus is an asynchronous bus meaning it does not wait for responses before beginning the next bus cycle. It is a request and response bus. Requests are outgoing from a bus master and incoming to a bus slave. Responses are output by a bus slave and input by a bus master. FTA bus includes standard signals for address, data, and control. These signals should be like those found on many other busses.

BUS TAGS

The bus has tagged transactions; there is an id tag associated with each bus transaction. The id tag contains identifiers for the core, channel, and transaction. The core is a core number for a multi-core CPU. Channel selects a particular channel in the core which may for instance be a data channel or an instruction channel. Finally, the transaction id identifies the specific transaction. Incoming responses are matched against transactions that were outgoing. For instance, a bus master may issue a burst request for four bus transactions to fill a cache line. Each transaction will have an id associated with it. When the slave receives the transactions it sends back responses for each of the four requests with ids that match those in the request. The slave does not necessarily send back responses in the same order. Transaction requests from the master may not arrive in order.

*An id tag of all zeros is illegal – it represents the bus available state.

SINGLE CYCLE

The bus operates on a single cycle basis. Transaction requests and responses are routed through the bus network as the bus is available and are present for only a single clock cycle. Bus bridges may buffer the transactions for a short period of time.

RETRY

If the bus is unavailable the retry response signal is asserted to the master. The master must retry the transaction.

SIGNAL DESCRIPTION

Following is a signal description for requests and responses for a 128-bit data version of the bus. Signal values have been chosen so that a value of zero represents a bus idle state. If nothing is on the bus it will be all zeros.

REQUESTS

Signal	Width	Description	
Om	2	Operating mode	
Cmd	5	Command for bus controller or memory controller	
Bte	3	Burst type	
Cti	3	Cycle type	
Blen	6	Burst length -1 (0=1 to 63=64)	
sz	4	Transfer size	

Segment	3	Code, data, or stack	
Cyc	1	Bus cycle is valid	
Stb	1	Data strobe	
We	1	Write enable	
Asid	16	Address space id	
Vadr	32/64	Virtual address	
Padr	32/64	Physical address	
Sel	16	Byte lane selects	
Data1	128	First data item	
Data2	128	Second data item	
Tid	13	Transaction id	
Csr	1	Clear or set address reservation	
Pl	8	Privilege level	
Pri	4	Transaction priority (higher is better)	
Cache	4	Transaction cacheability	

RESPONSES

Signal	Width	Description	
Tid	13	Transaction id	
Stall	1	Stall pipeline	
Next	1	Advance to next transaction	
Ack	1	Request acknowledgement (data is available)	
Rty	1	Retry transaction	
Err	1	An error occurred	
Pri	4	Transaction priority	
Adr	32/64	Physical address	
Dat	128	Response data	

OM

Operating mode, this corresponds to the operating mode of the CPU. Some devices are limited to specific modes.

CMD

Command for memory controller. This is how the memory controller knows what to do with the data.

Oridinal		
0	CMD_NONE	No command
1	CMD_LOAD	Perform a sign extended data load operation

2	CMD_LOADZ	Perform a zero extended data load operation
3	CMD_STORE	Perform a data store operation
4	CMD_STOREPTR	Perform a pointer store operation
7	CMD_LEA	Load the effective address
10	CMD_DCACHE_LOAD	Perform load operation intended for data cache
11	CMD_ICACHE_LOAD	Perform load operation intended for instruction cache
13	CMD_CACHE	Issue a cache control command
16	CMD_SWAP	AMO swap operation
18	CMD_MIN	AMO min operation
19	CMD_MAX	AMO max operation
20	CMD_ADD	AMO add operation
22	CMD_ASL	AMO left shift operation
23	CMD_LSR	AMO right shift operation
24	CMD_AND	AMO and operation
25	CMD_OR	AMO or operation
26	CMD_EOR	AMO exclusive or operation
28	CMD_MINU	AMO unsigned minimum operation
29	CMD_MAXU	AMO unsigned maximum operation
31	CMD_CAS	AMO compare and swap
Others		reserved

BTE

Burst type extension.

Ordinal	
0	Linear
1	Wrap 4
2	Wrap 8
3	Wrap 16
4	Wrap 32
5	Wrap 64
6	Wrap 128
7	reserved

CTI

Cycle Type Indicator

Ordinal		Comment
0	Classic	

1	fixed	Constant data address
2	Incr	Incrementing data address
3	erc	Record errors on write
4	Irqa	Interrupt acknowledge
7	Eob	End of burst
others		reserved

Normally write cycles do not send a response back to the master. The ERC cycle type indicates that the master wants a response back from a write operation.

BLEN

Burst length, this is the number of transactions in the burst minus one. There is a maximum of 64 transactions. With a 128-bit bus this is 1024 bytes of data.

SZ

Transfer size.

Ordinal		Transfer size
0	Nul	Nothing is transferred
1	Byt	A single byte
2	Wyde	Two bytes
3	Tetra	Four bytes
4	Penta	Five bytes
5	Octa	Eight bytes
6	Hexi	Sixteen bytes
10	vect	A vector 64 bytes (512 bit bus)
Others		Reserved

SEGMENT

The memory segment associated with the transfer.

Ordinal	
0	data
6	stack
7	code
others	reserved

TID

Transaction ID. This is made up of three fields.

Size	Use
6	Core number
3	Channel
4	Tran id

CACHE

Cache-ability of transaction. A transaction may be non-cacheable meaning as it progresses through the cache hierarchy it does not store data in the cache. It only stores data when it reaches the final memory destination.

Ordinal		
0	NC_NB	Non cacheable, non bufferable
1	NON_CACHEABLE	
2	CACHEABLE_NB	Cacheable, non bufferable
3	CACHEABLE	
8	WT_NO_ALLOCATE	Write-through without allocating
9	WT_READ_ALLOCATE	
10	WT_WRITE_ALLOCATE	
11	WT_READWRITE_ALLOCATE	
12	WB_NO_ALLOCATE	Write-back without allocating
13	WB_READ_ALLOCATE	
14	WB_WRITE_ALLOCATE	
15	WB_READWRITE_ALLOCATE	