Table888 v2

# Preface

## For whom this book is written.

This book is for the FPGA enthusiast who’s looking to do a more complex project. It’s advisable that one have a fairly good background in digital electronics and computer systems before attempting a read. Examples are provided in the System Verilog 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. The book attempts to be ‘hands-on’ in nature and provides sample program code.

## Motivation

One might think with a name like ‘Table888’ that this is a book about a diner or dinner date, but it’s really a book about developing a homebrew processor. As I sat down to develop yet another processor I named a table, table888. Then I thought to turn the table into a book, rather than just another ISA (Instruction Set Architecture) description. As time passes, things evolve, Table888 is one of those things. It’s time for v2 of the architecture.

I get to say here things that I’d never post in a hyper-technical document. Develop your own 64-bit processor ? Yeah right. One has to be somewhat nuts to consider it. But, it doesn’t take billions of dollars to develop a processor of one’s own at home; it just takes a lot of time and dedication. If you seek to be an expert on the personal computer or laptop sitting on your desk, there’s nothing like trying to develop your own processor to learn things. It’s possible these days to develop something simple and rudimentary using a small FPGA board available from several different vendors. One can get started working with FPGA’s for well under $100; with free toolsets available it’s not an expensive hobby. It’s no more expensive than a good video game, and can provide a lot of entertainment for the money. For an outlay of a few hundred dollars one can begin to become a real expert on home-grown processors, including some of the more advanced aspects of processor such as memory management and data protection. FPGA stands for ‘Field Programmable Gate Array’, which is a chip with lots of small memories interconnected with a connection network. I’m currently using the Nexys Video board from Digilent. I’ve upgraded several times, to more memory and more logic cells. I’ve used boards from Terasic and BurchEd in the past. Of-course it’s also possible to make your own board if you have the skills. The first board I used was one I wired up myself but it didn’t work very reliably. Be sure to recycle the boards appropriately; I sell my older boards on Ebay to budding students.

The processor presented here isn’t the smallest and fastest RISC processor. It’s also not a simple beginner’s example. Those weren’t my goals. Instead it offers reasonable performance. 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 64 bits 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 double or more the resources that a 32-bit design would.

## 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 20 years professional experience working on banking applications at a variety of language levels including assembler. So, I have some real-world experience developing complex applications. I also have a diploma in electronics engineering technology in what is now classified as a degree course. Some of the cores I work on these days are really too complex and too large to do at home on an inexpensive FPGA. I await bigger, better, faster boards yet to come.

# Overview

## History

Table888 v2 is a work-in-progress beginning in January 2020. Table888 v2 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 a number of other processing cores. In particular a lot is borrowed from the RiSCV architecture and comparison to RiSCV and other processors is made throughout the document.

## Features

The feature set is designed around the idea of supporting a modern operating system.

CPU

* 64-bit integer data path
* 64-bit double precision floating-point data path
* 64-entry unified integer and floating-point register file
* 4 link registers
* variable sized instructions (16,32,48)
* 2-way out-of-order (ooo) superscalar execution
* precise exception handling
* branch prediction with branch target buffer (BTB)
* Instruction L1, L2 and data L1, L2 caches
* 7 entry write buffer
* Dual memory channels
* FT64v10 fetches two instructions at once and can issue up to seven instructions in a single cycle (2 alu, 1 flow control, 2 floating point, 2 memory) and is capable of committing up to two instructions in a single cycle.

MPU

The cpu core is wrapped up in an MPU core which provides even more capabilities for system support. The mpu has the following features:

* 6 programmable timers
* 32 source interrupt controller
* paged memory management unit
* cpu core

## Design Rationale / Goals

A lot is borrowed from the RiSCV architecture. It’s not that the author can’t come up with something original, it’s just that RiSCV is such a good base to start with.

A couple of things about RiSCV. One is that it is a spartan architecture. Only the bare minimum required to get the job done is supplied, this makes a lot of sense in terms of energy efficiency, performance and cost. One of the goals of RiSCV is that it could be used in almost anyplace from small embedded systems and on up. It is a real architecture, there are a number of companies supporting it, it lives outside the domain of academia. There is some motivation then to make the RiSCV footprint as small as possible in a minimum configuration so that it may fit into tiny devices. It could be classified as a speed demon, relying on a streamlined architecture to achieve a high clock rate resulting in good performance. The RiSCV paradigm for missing features is to use fused instructions for some of the instructions found in other architectures. For example, indexed addressing modes or calculating effective addresses. The author prefers a more Brainiac architecture that runs at a slower clock rate but does more work per clock cycle. The author’s goals are somewhat different than RiSCV and that leads to an innovative design. In particular the floating-point unit is low latency versus having a high clocking rate. One goal is for the core to be somewhat educational in nature. Another goal is for the core to be a basis for learning about operating systems. Jumping through hoops to achieve a high clock rate is a goal left to the reader. Instead this core offers decent performance while remaining easier to understand.

## Nomenclature

The ISA (Instruction Set Architecture) refers to primitive object sizes following the convention suggested by Knuth of using Greek.

|  |  |  |
| --- | --- | --- |
| Number of Bits |  | Instructions |
| 8 | byte | LDB, STB |
| 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.

# Parade of Acronyms

Several common acronyms are described here.

SOC

SOC is an acronym for system-on-chip. System-on-chip is an entire system located on a single chip. It includes mpu / cpu / IO devices and memory. Included in FT64’s SoC are keyboard controller, serial communications controller (UART), audio / video controllers and a small amount of memory.

MPU

MPU stands for micro-processing unit. An mpu typically contains additional hardware beyond the cpu proper. MPU components are often separate from the rest of the system. The same mpu may be used in different SoC configurations. The MPU will often contain clock generators, interval timers, gpio (general-purpose I/O), serial ports (UARTS), and interrupt controllers.

PIT

PIT stands for programmable interval timer. It’s a reasonably well-known acronym for such devices. There are two PITs in the mpu, each one having three programmable channels. Three channels on the first PIT are used as timing sources for the time-slice interrupt and garbage collector timing. Two channels on the second PIT are used to drive timing requirements for the paged-memory-management unit.

PIC

PIC stands for programmable interrupt controller, also a reasonably well-known acronym for such. There is a single PIC in the system. The PIC supports 32 interrupt sources. Several of the sources are in use to support the time-slice interrupt and garbage collection interrupts. The entire system on chip (SoC) provides additional interrupt sources.

PMMU

PMMU stand for paged memory-management unit. Another popular acronym. The pmmu provides virtual to physical address translations and acts as a bridge between the cpu and the memory system.

# Choosing an Implementation Language

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

Choosing a language is somewhat of a personal choice. One should choose whatever works best for themselves. There are two popular HDL languages (Verilog, and VHDL) and number of others. I encourage you to search the web for HDL languages and find something you’re comfortable with. 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 Verilog. VHDL is more verbose than Verilog and has tighter control of types. FT64 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 in order to get one’s own toolsets working well. Is it the processor that’s broken ? or the toolset ? That program didn’t work because the assembler didn’t assemble it correctly, it wasn’t a bug in the processor. Keeping everything ‘in sync’ is like a dance, one goes around and around in circles. I’ve had to develop my own assembler, disassembler, compiler, glyph editing program and other things. It’s more involved than one might anticipate to begin with. For instance, in order to get character display on-screen a glyph editor was needed. I looked at a couple of free ones available on the net, but they didn’t quite do what I needed. I needed something that could output FPGA vendor compatible files, and the free glyph editors were geared towards graphics files formats. After spending about a day trying to modify an existing editor I gave up, and decided to roll my own. I first developed a simple assembler about 30 years ago for use at school; I still use the same source code with many, many updates. The assembler has become quite powerful now.

## Documenting the Design

Any processor design is likely to have a number of documents associated with it. One needs to be able to refer to things like what opcode does what, outside of the implementation code itself. For general tasks I’m using MS Office. Word for word processing, and Excel for spreadsheets. 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. One can get by with just paper and pencil but there’s bound to be a lot of changes in any project this complex. Hand drawn schematics to sketch out a basic idea maybe the easiest approach, but then one might want a scanner to immortalize the idea. Having electronic tools is a great boon for development.

## Building the System

In order to actually produce an implementation some sort of FPGA developer tools will be required. The FPGA devices typically have to 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 finished third party software that can generate bitstreams from source code. I’ve used both free toolsets from Altera and Xilinx. The most recent release (Vivado 19.2) of the free Webpack tools from Xilinx seems fairly stable under Window 10.0.

## 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 really fast. 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. 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 FT64. One of these is the LCC compiler. I used the LCC compiler for the Butterfly32 project. It’s fairly straightforward to implement the compiler for a new ISA especially if your ISA is similar to an existing one. Another toolset is the gcc compiler. I haven’t actually put this toolset to use yet, but I’ve had a look at it. It seems somewhat daunting. GCC is very general in nature and supports a lot of target architectures. People have put a lot of work into making this compiler available for any architecture. I know a number of people have been turned off by the complexity however. LLVM is another compiler tool being actively developed.

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

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 it’s almost like the two-headed hydra monster to be slain. 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 have to be able to think in terms of ‘what test can I do ?’ to fix the bug. There are usually a least several wow-zzy bugs. For example, I had a bug where a register exchange instruction only failed on a cache miss, when the instruction was at the end of a cache line. Many programs actually worked fine, and the processor seemed not to work intermittently. It took quite a while to find. I finally noticed the instruction failed when the cache was turned off. So, one thing to try for testing is turning the cache on or off.

## Test Benches

If 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.

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 really minor and well-known, I run simulator traces before attempting to run the code in an FPGA.

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

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

which turns on all the LEDs on the board.

Another suggestion for test-benches is to use the actual system being loaded into the FGPA 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.

## 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. Logic simulators are great but maybe not the best tool for resolving issues in some circumstances. For instance, there was a bug in a video project the author was working on that didn’t manifest itself until after the system was running for several hours. Using a logic simulator to try and find the bug was out of the question. It would simply take far too long for the simulator to reach the point of the bug. There are different styles of emulators which are useful for different tests. An emulator may be cycle-accurate. Cycle accurate emulators emulate the system on a cycle-by-cycle basis. They are often slow compared to other emulators, but capable of revealing issues that wouldn’t be found with a higher speed emulator. Use the right tool for the job.

## 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 a 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 really simple things. I have two versions of keyboard routines. The one that ‘works the right way’ and the one I use for bootstrapping. The bootstrapping routine goes directly to the keyboard port to read a character. It’s really simple, and pauses the whole machine waiting for a character.

## 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’s the time that I’ve 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 really complex design. But all the complicated stuff started out small and simple. One doesn’t need caches, interrupts, branch predictors, and so on in order to have a working design. It’s very rewarding to have even the simplest design working.

## Disabling Interrupts

This bit really only applies if you’ve managed to get some sort of interrupt facility working. A number of smaller, simpler systems don’t make use of interrupts. In fact some contemporary operating systems run with interrupts disabled! 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 in order 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.

## IRQ Live Indicator

An indicator that IRQ’s are happening seems like a friendly image. It can be useful to see that IRQ’s are happening on a regular basis. An IRQ indicator can let one know if the machine is just busy, or really, really stuck. This can be accomplished by incrementing a character at a fixed location on-screen. IF that character stops flipping around one knows there’s real trouble. Another common approach is to use an LED to indicate the presence of IRQ’s. A multi-color LED is a great way to allow visualization of interrupts. With different interrupt source tied to each component of color the LED will vary in color according to what interrupts are happening. When the LED looks the wrong color, something is wrong with the system.

## 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 or not 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. Debug registers are most useful to debug software after the core is working.

### 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 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.

### Integrated Logic Analyzers

For really complex diagnostics an integrated logic analyzer may be useful. This could be a tool that one builds oneself, or more likely a tool provided by the vendor. The logic analyzer allows signals to be recorded and dumped to a display. There may be a number of means of triggering the recording of signals. For the Vivado toolset the ILA is a component that can be plugged into the system being debugged. It is relatively easy to do compared to some other debugging approaches.

## Stuck on a Bug ?

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

## The Rare Chance

There is a rare chance that it’s a problem in the toolset. A problem like this can make things really difficult, especially if it’s a free toolset with no technical support. In about 10 years or so, of using toolsets I’ve found a few bugs. The toolsets generally speaking are superb, so the chance of it being a bug in a toolset is extremely remote but not impossible. The one bug I ran into was in extending a complement of a single bit value. The toolset returned 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.

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.

## Triple Mode Redundancy Testing

Be wary of intermittent bugs that are not actually the toolset’s fault but due to things like bad memory bits in the workstation. With gigabytes of ram occasionally there may be a bad bit of memory. This is often spurious thing that is resolved after a re-boot. This kind of thing manifests itself in the toolset as a signal that stays fixed when it should be varying. The author has run into this several times. Workstations don’t last forever without maintenance. One way to get around this sort of issue is to use triple-mode redundant logic. Even if there is a bug in the development workstation, triple-mode redundant logic may be able to bypass this bug.

# Design Choices

## 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. FT64 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. These are called speed demons. 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. These are called brainiacs. 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. I’m partial to little endian machines; it just seems more natural to me. Whichever endian is chosen, often the machine has instructions(s) for converting from one endian to the other. Myself, I don’t bother with endian conversion; it’s a feature that I probably wouldn’t use. Some implementations even allow the endian of the machine to be set by the user. This seems like overkill to me. The endian of data is important because some file types depend on data being in little or big-endian format. FT64 is a little-endian machine.

## 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 FT64 is an overlapped superscalar pipeline. The following chart shows the relationship ship between pipelining, clock frequency, and design complexity. It’s based on my 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 | 6 | 16 | 2000 | Sequential, non-overlapped |  |
| Raptor64 | 50 MHz | 4 | 12.5 | 10000 | Overlapped pipeline |  |
| Table888v2 | 30 MHz | 2.5 | 12 | 100000 | Superscalar 2-way |  |

A superscalar architecture can hide some of the memory access time by executing other instructions while memory access is taking place. The author has had better luck getting simpler architectures to run at high speed. Note that power consumption is proportional to clock frequency. It’s desirable to run the processing system at a lower clock rate.

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.

FT64 uses a WISHBONE compatible bus to communicate with the outside world. Specs for the WISHBONE bus can be found at OpenCores.org. WISHBONE bus is straightforward and easy to understand and free. It is used by a number of other projects. Another popular bus standard is the AMBA standard. The external bus used by FT64 is a 128-bit bus. This is the size of the system’s data bus. The peripherals in the test system use a bus width that varies from a single byte to 128-bits. The ROM’s and RAM’s in the system are all 128-bits wide. FT64 makes use of two word burst memory accesses to load the instruction cache. A burst access is a number of accesses that occur rapidly in a row in a sequence.

## Choosing an ISA

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

## Readability

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

## Planning for the future

If one leaves no room for future instructions, it’ll be difficult to upgrade the processor at a later date. This 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.

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. I recently reviewed the VAX architecture which is a variable length architecture. For the VAX the instruction formats vary depending on the addressing mode and size of the operands.

### 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. The first version of Table888 uses a 128-bit instruction bundle to house three forty-bit instructions. This version of Table888 has a more sophisticated instruction cache that can handle the 40-bit instructions. There’s no need for the 128-bit bundles.

### 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.

## Register Files

Only the most unusual processors don’t contain any registers. Registers are essential for performance. They act as a cache of memory values.

### Number of 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 actually improve very much in performance over having 20 or so registers. Having more registers impacts the task switch time because they have to 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 the author’s current design he’s chosen to use 64 registers, in part because the register number fits nicely into a byte when coupled with a couple of extra bits. This is a reduction in the number of registers from the original Table888. The original Table888 was either going to be 16 or 256 registers, to make the register number readable. Also, within the FPGA memory resources are allocated in blocks. These blocks are typically 512 or 2048 bytes in size. 256 registers fit nicely into a 256x64 block of memory (2kB).

### 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 register read file path.

### Unified or Not?

One choice for register files is whether or not to incorporate all registers in the same register file, or to use separate register files. This concern is mainly for floating-point versus integer registers. If floating-point and integer registers differ in size, or if floating-point is optional it may be better to have a separate floating-point register file. I’ve seen some comments that a unified register file is a design mistake, because the size of the register file and number of ports required will impact the performance of the processing core. However, within an FPGA register file depth may come ‘for free’ with no additional timing concern. Table888 v2 uses a unified integer and floating-point register file. This makes more registers available if floating point is not required.

### Segment Registers

As part of the memory management portion of a cpu segment registers are often provided. There are usually multiple segment registers in order to support multiple segments which are typically part of a program. Common program segments are: the code segment, the data segment, the uninitialized data segment and the stack segment. There are often other segments as well. 80x88 is famous for its segment registers, but other processors like IBM’s PowerPC also use them as well. Segment registers are a fairly easy to understand and a low cost, low overhead memory management approach. The memory address from an instruction is added to a value from a segment register in order to form a final address. The segment register is often shifted left as it is added in order to allow a greater physical memory range than the range directly supported by the architecture. Segment registers allow programs to be written as if they had specific memory addresses available to them, such as starting at location zero, while in reality the actual physical address of the program is much different.

### Base and Bound Registers

Base and bound registers are almost the same as segment registers. They could be viewed as a synonym except that a segmented register approach is usually a lot more sophisticated than a base and bound register. Segments typically have different types and access rights associated with them. Base and bound registers are like the heart of a segmentation system without all the other baggage that goes along with segmentation. A base register is added to addresses to form a final address and the result compared against and limit established by the bound register.

### 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 v2 has a handful of special registers that are accessed with the ‘CSR’ instruction. The CSR instruction allows atomic read and update of special function register. CSR stands for Control and Status Registers.

### 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. Table888 supports a register move instruction.

### 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. Table888 has some basic register usage constraints. Take a look at the CPU programming model section of the book to see how registers are used for Table888.

## Handling Immediate Values

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

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

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

|  |  |  |
| --- | --- | --- |
| Constant32 | 2Eh | IMM1 |
| Constant32 | 2Fh | IMM2 |
| Constant32 | 2Dh | NOP |

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

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

### SETHI

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

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

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

Disadvantages of this approach:

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

Advantages of this approach:

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

### IMMxx

Second solution: use an immediate prefix or postfix instruction. The constant prefix or postfix instruction simply contains the bits of the constant that wouldn’t fit in the following instruction. A prefix comes before the instruction, a postfix comes after the instruction. It looks like the following:

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

Advantages:

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

Disadvantages:

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

### LW Table

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

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

Advantage:

It’s simple. It doesn’t require a special means (instructions) to handle constants. Uses a means already present in the processor. This may be useful when the size and complexity of a processor is an issue. Sometimes it’s more practical for example with 128-bit or larger constants.

Disadvantages:

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

It uses a register.

### Half-Operand Instructions

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

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

Advantages:

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

Disadvantages:

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

2) Increases the complexity of the processor.

## The Branch Set

One of the first things I look at when evaluating an ISA is the branch set. Is it semi-sensible or non-sense ? Branches may represent up to one quarter of instruction executed. Branches are one item that have to be well done in an architecture. What conditions will the processor branch on ? Is it a simple branch on zero / non-zero test or are there more complex conditions available ? What the branch set supports impacts what other instructions need to be available in the architecture. If branching only supports a zero / non-zero test, then other instructions must be present to setup the branch test. In the DLX architecture for instance, there are a set of ‘set’ instructions that set a register to a one or zero based on a condition. After a set instruction is done, then a conditional branch may occur. Many architectures include a compare instruction(s). For instance, the MMIX architecture includes both signed (CMP) and unsigned compare (CMPU) instructions that set the value of a register to -1, 0, or 1 for less than, equal, or greater than another register. The same paradigm was used for the Raptor64 processor. For the Table888 processor there is a fairly standard set of branches that act like they are branching on a flag register value. If you’re used to the 6800 / 68x00 / 6502 series processor, these branches will look familiar. Table888 v2 uses slightly different mnemonics for branches, the basic ideas are the same.

### Branch Targets

Branches which change program flow conditionally are usually implemented as relative branches. One reason to implement using relative addresses is that it takes fewer bits to represent the target address of the branch. In many designs, typically 16 bits are allowed for, for a branch displacement even though only 12 bits are really necessary. It has to do with keeping the format of instructions simple and there is usually room in a branch instruction for sixteen bits. Even in byte-code architectures that use eight-bit branch displacements by default, there is often a longer form for branches supported (for example the 6809). A second reason to use relative branching is that it allows code to be relocated in memory without having to modify the branch instructions. 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.

The Table888 v2 design allows 12 bits for the branch displacement. There’s a little bit less room available in the branch instruction than was present in the first version, due to the compare-and-branch nature of the more recent version. In the Table888 design 21 bits were allowed for because there were 24 bits available. This may seem like overkill, but it was trying to look into the future of branches. When people write structured subroutines, they typically don’t create a routine more than a few pages long. This results in branching that branches within a few kilobytes of the branch location because branches are located within a subroutine. Hence the reason 12 bits is adequate. However, if one is using an automated code generator, the code generator may generate larger subroutines. Branch instructions formats can be found later in the book under the CPU description.

### Branch Prediction

Branch prediction enhances performance by predicting whether or not a branch will occur. This allows the processor to fetch instructions in an uninterrupted fashion. 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 had little value for the first version of the Table888 processor as it’s a non-overlapped pipeline. It took multiple cycles to execute a branch whether or not prediction was present. Branch prediction adds additional complexity to the processor. Version 2 of Table888 incorporates a branch predictor. There are several options when it comes to branch prediction. Branch prediction is a complex topic that could consume several chapters of a book. Only a brief outline is given here.

#### The Pattern History Table (PHT)

PHT is an acronym for pattern history table. The pattern history table is a small table used to record the pattern of branch activity for a particular branch. A pattern history table is often implemented as two-bit saturating counters. The counter increases when a branch is taken and decreases when a branch is not taken. The count then determines the predictability of the branch.

#### Global Branch History

Global branch history is a global record of the taken or not taken status of branches as they occur. The global history is usually recorded in a shift register which shifts every time a new record is entered. Obviously there is a limit to the amount of global history that can be recorded and is useful.

#### gSelect Predictor

gSelect refers to global selection predictor. A g-select predictor uses global branch history to select from among a number of pattern history tables.

#### gShare Predictor

gShare – a gshare predictor is similar to a gSelect predictor except that the global history is xor’d with part of the address of the branch instruction to determine the PHT to use.

#### Perceptron Predictor

Perceptron predictors makes use of an emulated perceptron neural network to predict branches. It multiplies histories times weights and sums the result. A good description is beyond the scope oif this book. The perceptron predictor is a bit slower than the other predictors and may take two clock cycles rather than having a prediction result available immediately.

#### Tournament Predictors

A tournament predictor implements multiple branch predictors and tracks which predictor is the most accurate for given branches. The branch prediction from the most accurate predictor is used to estimate the branch outcome.

### Looping Constructs

Sometimes processors support looping constructs directly. 680x0 has a decrement and branch instruction. 80x88 has loop instructions which decrement the CX register and branch. Decrementing a register then branching if it is non-zero is a common operation, so a number of processors implement these two operations together with a single instruction. It’s really like executing two instructions at once. Table888 v2 supports the incrementation or decrementation of a register during a branch test in some circumstances.

## 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 v2 makes use of a JAL instruction, storing the return address in a register. This differs from version one which automatically stored 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).

# Development Aspects

## Device Target

The core has been developed with FPGA usage in mind. FPGA’s are a relatively low cost means to test novel hardware ideas. They are available to many hobbyists and practical as an educational tool. Using an FPGA allows logic to be designed using EDA tools and languages used for real designs. This is as opposed to implementing an instruction set using a high-level programming language.

## Implementation Language

The core is implemented in the System Verilog language primarily for its ability to process array objects. Much of the core is plain vanilla Verilog code. Not all features of the System Verilog language are in use. Some features were found to be not present yet in the EDA toolset.

# CPU

# Programming Model

## Registers

Overview

Table888 v2 is a register-oriented machine with 64 general purpose registers. Registers are 64-bits wide. The register file is *unified*, holding either integer or floating-point values. There are also a number of special purpose registers in the architecture.

|  |  |
| --- | --- |
| Integer or Float | |
| Reg |  |
| r0 | always zero (or +0.0) |
| r1 | first return value |
| r2 | second return value |
| r3 |  |
| r4 |  |
| r5 |  |
| r6 |  |
| r7 |  |
| r8 |  |
| … | … |
| r48 |  |
| r49 |  |
| r50 |  |
| r51 | garbage collector |
| r52 | garbage collector |
| r53 | garbage collector |
| r54 | assembler usage |
| r55 | type number |
| r56 | class pointer |
| r57 | thread pointer |
| r58 | global pointer |
| r59 | return address |
| r60 | exception linkage |
| r61 | exception sp offset |
| r62 | user stack pointer |
| r63 | system stack pointer |
| Link Registers | |
| lk0 | always 0 |
| lk1 | return address |
| lk2 |  |
| lk3 |  |
| Instruction Pointer | |
| ip | points to instruction |

The stack and frame pointer registers are subject to stack bounds checking.

## Control and Status Registers

Control and status registers are accessed with the CSR instruction described later in the book.

### HARTID (0x001)

This register contains a number that is externally supplied on the hartid\_i input bus to represent the hardware thread id or the core number. No core should have the value zero as the hartid.

### TICK (0x002)

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.

### PTA (0x003)

This register contains the base address of the highest-level page directory for memory management, the paging table depth and the size of the pages mapped. The base address must be page aligned (16kB).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 63 14 | 13 11 | 10 8 | 7 6 0 | |
| Paging Directory Base Address63..14 | ~ | TD | S | ~ |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| TD |  | Address Bits |  | S |  |
| 0 | 1 level lookup | 24 |  | 0 | map 16kB pages |
| 1 | 2 level lookup | 35 |  | 1 | map 4MB pages |
| 2 | 3 level lookup | 46 |  |  |  |
| 3 | 4 level lookup | 57 |  |  |  |
| 4 | 5 level lookup | 63 |  |  |  |
| 5 to 7 | reserved |  |  |  |  |

### CAUSE (0x006)

This register contains a code indicating the cause of an exception or interrupt. The break handler will examine this code in order to determine what to do. Only the low order 13 bits are implemented. The high order bits read as zero and are not updateable.

### BADADDR (CSR 0x007)

This register contains the effective 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 XL register.

### BAD\_INSTR (CSR 0x00B)

This register contains a copy of the exceptioned instruction.

### FSTAT (CSR 0x014) Floating Point Status and Control Register

The floating-point status and control register may be read using the CSR instruction. Unlike other CSR’s the control register has its own dedicated instructions for update. See the section on floating point instructions for more information.

### KEYS – (CSR 0x020 to 0x022)

These registers contain a collection of keys associated with the process for the memory system. Each key is twenty bits in size. Each register contains three keys with a total of eight keys. All three registers are searched in parallel for keys matching the one associated with the memory page.

|  |  |  |  |
| --- | --- | --- | --- |
| 63 60 | 59 40 | 39 20 | 19 0 |
| ~6 | key3 | Key2 | key1 |

### DOI\_STACK (0x040)

This register contains the stacks for the data operating level, code operating level and interrupt mask. All three stacks are packed into a single register for convenience and performance if the stacks are required to be saved or restored as part of context. When an exception or interrupt occurs, a) the interrupt stack is shifted to the left by three and the low order bits are set to all ones causing all interrupts to be masked b) the code operating level stack is shifted to the left and the low order bits are set to zero causing a switch to the machine operating level for code 3) the data operating level stack is shifted to the left and the low order bits are set to zero causing the machine operating level to be used for data access.

When an RTI instruction is executed these registers are shifted to the right, restoring the previous settings. a) The last interrupt stack entry is set to seven masking all interrupts on stack underflow. The low order three bits represent the current interrupt mask level. b) The last code operating level stack entry is set to zero causing a switch to machine mode on stack underflow. c) The last data operating level stack entry is set to zero causing the machine operating level to be used for data access.

Only the low order 45 bits of the register are implemented, remaining bits read as zero.

Bits 0 to 2 represent the current interrupt mask setting.

Bits 15 to 17 represent the current code operating level setting.

Bits 30 to 32 represent the current data operating level setting.

### TIME (CSR 0xFE0)

The TIME register corresponds to the wall clock real time. This register can be used to compute the current time based on a known reference point. The register value will typically be a fixed number of seconds offset from the real wall clock time. CSR 0xFE0 bits are driven by the tm\_clk\_i clock time base input which is independent of the cpu clock. The tm\_clk\_i input is a fixed frequency used for timing that cannot be less than 10MHz or more than 256MHz. It is suggested to use the slowest clock in range available in the system. The low order 28 bits represent the fraction of one second. The high order 36 bits represent seconds passed. For example, if the tm\_clk\_i frequency is 100MHz the bits should count from 0 to 99,999,999 then cycle back to 0 again. When the bits cycle back to 0 again, the high order bits of the CSR 0xFE0 register are incremented.

Note that this register has a fixed time basis, unlike the TICK register whose frequency may vary with the cpu clock. The cpu clock input may vary in frequency to allow for performance and power adjustments.

### INFO (0xFF0 to 0xFFF)

This set of registers contains general information about the core including the manufacturer name, cpu class and name, and model number.

## Instructions

Opcodes are grouped together in part according to the functional unit they will be executing on. By grouping the opcodes it’s easy to decode where they will execute. This leads to better processor performance. RiSCV uses groups of eight opcodes, Table888 v2 uses groups of sixteen opcodes to allow more operations to be specified in a group. Instructions are 32-bits in size. This results in greater code density over Table888 v1.

Group4

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | xx00 | xx01 | xx10 | xx11 |
| 00xx | System / JAL | ALU immediate | Load | Float Immediate |
| 01xx | branch | ALU Immediate | Store | Float Registered |
| 10xx | reserved | ALU registered |  |  |
| 11xx | reserved | ALU registered |  | IMM prefix |

## ALU Register-Immediate Instruction Format

The register immediate format allows a register value in Ra to be combined with an immediate constant supplied in the instruction. A number of operations are possible. These include addition (ADDI) logical operators (bitwise ANDI, ORI, and XORI), multiplication (MULI), division (DIVI) and a group of set instructions.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 20 | 19 14 | 13 8 | 7 4 | 3 0 |
| imm12 | Ra6 | Rt6 | Group4 | Fnc4 |

Set instructions set the target register to true if the condition is met, false otherwise. Note there is a full complement of set instructions, compared to RiSCV which provides only SLTI / SLTUI instructions in part the author is sure because of encoding limitations. To get any other set operations with RiSCV multiple or alternate instructions must be used.

The shift instructions have their own subgroup within the immediate mode instructions because they do not require a 16-bit immediate value and better use of the coding space is made.

Group 1

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Fnc4 |  |  | Fnc4 |  |  |
| 0 | ADDI | add | 8 | SLTI | set if less than |
| 1 | reserved |  | 9 | SGEI | set if greater than or equal |
| 2 | {SHIFT} | shift operations | 10 | SLEI | set if less than or equal |
| 3 | ANDI | bitwise ‘and’ | 11 | SGTI | set if greater than |
| 4 | ORI | bitwise ‘or’ | 12 | SLTUI | set if less than – unsigned args |
| 5 | XORI | bitwise ‘xor’ | 13 | SGEUI | set if greater than or equal – unsigned |
| 6 | SEQI | set if equal | 14 | SLEUI | set if less than or equal – unsigned |
| 7 | SNEI | set if no equal | 15 | SGTUI | set if greater than - unsigned |

Group 5

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Fnc4 |  |  | Fnc4 |  |  |
| 0 | MULI | multiply | 8 |  |  |
| 1 | DIVI | divide | 9 |  |  |
| 2 | MODI | modulus | 10 |  |  |
| 3 |  |  | 11 |  |  |
| 4 | MULUI | unsigned multiply | 12 |  |  |
| 5 | DIVUI |  | 13 |  |  |
| 6 | MODUI |  | 14 |  |  |
| 7 |  |  | 15 |  |  |

### Shift Immediate Instruction Format

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 26 | 25 20 | 19 14 | 13 8 | 7 4 | 3 0 |
| SFunc3 | ~3 | imm6 | Ra6 | Rt6 | 14 | 24 |

There are more shift functions available than are present in RiSCV. RiSCV does not support rotates. Rotating values has limited support by high-level languages and often rotates are built out of shift operations. The mnemonics used here are slightly different but are similar to mnemonics used in many architectures.

|  |  |  |
| --- | --- | --- |
| SFunc3 |  |  |
| 0 |  | reserved |
| 1 | ASL | shift to the left, fill lsb with zero |
| 2 | ROL | rotate left, fill lsb with msb |
| 3 | LFSL | linear feedback shift |
| 4 | LSR | logical shift to right, fill msb with zero |
| 5 | ASR | arithmetic shift right, preserve msb |
| 6 | ROR | rotate to right, fill msb with lsb |
| 7 |  | reserved |

The shift count is ignored for a linear feedback shift which always shifts a single bit at a time.

Immediate Prefix Instruction

What happens when an immediate value is too large to be encoded in the instruction? An immediate prefix instruction is used to extend the range of the following instruction. There may be one or two immediate mode prefix instructions depending on the number of significant constant bits required. Immediate mode prefixes are covered in more detail under the flow control section.

## ALU Register-Register Instructions

Register-register instructions support the same functionality as register-immediate instructions, except that two register values are combined rather than a register and immediate. Note that some of the set instructions are not present because they would be redundant. The same functionality is supported by other set instructions.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 26 | 25 20 | 19 14 | 13 8 | 7 4 | 3 0 |
| ~6 | Rb6 | Ra6 | Rt6 | Opcd4 | Fnc4 |

Group 9

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Fnc4 |  |  | Fnc4 |  |  |
| 0 | ADD | add | 8 | SLT | set if less than |
| 1 | SUB | subtract | 9 | SGE | set if greater than or equal |
| 2 | {SHIFT} | shift operations | 10 |  | reserved |
| 3 | AND | bitwise ‘and’ | 11 |  | reserved |
| 4 | OR | bitwise ‘or’ | 12 | SLTU | set if less than – unsigned args |
| 5 | XOR | bitwise ‘xor’ | 13 | SGEU | set if greater than or equal – unsigned |
| 6 | SEQ | set if equal | 14 |  | reserved |
| 7 | SNE | set if no equal | 15 |  | reserved |

Group 13

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Fnc4 |  |  | Fnc4 |  |  |
| 0 | MUL | multiply | 8 | PERM | byte permute |
| 1 | DIV | divide | 9 |  |  |
| 2 | MOD | modulus | 10 |  |  |
| 3 |  |  | 11 |  |  |
| 4 | MULU | unsigned multiply | 12 |  |  |
| 5 | DIVU |  | 13 |  |  |
| 6 | MODU |  | 14 |  |  |
| 7 |  |  | 15 |  |  |

### Register-Register Shift Format

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 26 | 25 20 | 19 14 | 13 8 | 7 4 | 3 0 |
| SFunc3 | ~3 | Rb6 | Ra6 | Rt6 | Opcd4 | Fnc4 |

The SFunc3 field has the same meaning as for immediate shifts.

## Program Flow Control

### Unconditional Control Transfer

The unconditional program flow transfer instruction is JAL – jump and link. JAL allows subroutine or method calls to be performed. Jump and link loads the low order 23 bits of the instruction pointer with the value specified in the instruction. The value specified in the instruction is shifted left once before use. A direct load differs from the RiSCV which uses relative addressing which adds the value specified in the instruction to the instruction pointer. This is also a good choice given the more limited address field in RiSCV. The address of the next instruction (ip + 4) is stored in a link register. If a plain jump is desired the return address may be discarded by writing to link register zero, which always contains a zero. It’s important for performance reasons that this instruction not read any registers. By not relying on a register read,, it means the jump can take place immediately in the fetch stage, maximizing performance.

|  |  |  |  |
| --- | --- | --- | --- |
| 31 10 | 9 8 | 7 4 | 3 0 |
| Imm22 | Lt2 | 04 | 84 |

What happens if a 23-bit address isn’t large enough to specify the target? A register indirect jump is needed which may have an immediate prefix to extend the address range.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 20 | 19 14 | 13 10 | 9 8 | 7 4 | 3 0 |
| Imm12 | Ra6 | ~4 | Lt2 | 04 | 94 |

Now that its possible to call a subroutine some means is required to return from one. This is where the RET instruction comes into play. In order to return, a link register is loaded into the instruction pointer. Additionally, a value which is the sum of a register and an immediate constant in the instruction is loaded into the stack pointer. The register forming part of the sum is usually the stack pointer.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 20 | 19 14 | 13 10 | 9 8 | 7 4 | 3 0 |
| Imm12 | Ra8 | ~4 | Ls2 | 04 | 104 |

Group 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | System | |  | JAL | |
| Fnc4 |  |  | Fnc4 |  |  |
| 0 | BRK |  | 8 | JAL |  |
| 1 | TRAP |  | 9 | JALR |  |
| 2 | IRQ |  | 10 | RET |  |
| 3 | NMI |  | 11 |  |  |
| 4 | RST |  | 12 | NOP |  |
| 5 | RTI |  | 13 |  |  |
| 6 |  |  | 14 | PFX1 | supplies bits 48 to 79 of a constant |
| 7 |  |  | 15 | PFX2 | supplies bits 16 to 47 of a constant |

The BRK instruction is defined as all zeros. It invokes the exception hander at address $FF…FC0000.

IRQ, NMI and RST all invoke the exception handler at address $FF…FC0000. This is described in more detail later.

RTI return from an interrupt or exception.

## Floating-Point Instructions

### Immediate Form

Note there are no single operand immediate forms. It makes little sense to perform an operation on a constant when that constant can be calculated by the assembler or compiler.

Note the immediate constant used for float-point ops may be extended with immediate prefixes to increase the range and precision of the constant. This is described under immediate prefixes.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 25 | 24 20 | 19 14 | 13 8 | 7 4 | 3 2 | 1 0 |
| mantiss7 | exp5 | Ra6 | Rt6 | 64 | Fn4 | |

Only basic floating-point operations are supported which include addition (FADD), subtraction (FSUB), comparison (FCMP), multiplication (FMUL), and division (FDIV).

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Fnc4 |  |  | Fnc4 |  |  |
| 0 | FADDI | add | 8 | FSLTI | set if less than |
| 1 | FSUBI | subtract | 9 | FSGEI | set if greater than or equal |
| 2 | FCMPI | compare | 10 | FSLEI | set if less than or equal |
| 3 | FMULI | multiply | 11 | FSGTI | set if greater than |
| 4 | FDIVI | divide | 12 | FSEQI | set if equal |
| 5 |  |  | 13 | FSNEI | set if not equal |
| 6 |  |  | 14 |  |  |
| 7 |  |  | 15 |  |  |

Registered Form

Note multiply and add instructions (FMA, FMS, FNMA, FNMS) make use of a third register, other instructions do not. For the other instructions Rc6 should be specified as R0.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 26 | 25 20 | 19 14 | 13 8 | 7 4 | 3 0 |
| Rc6 | Rb6 | Ra6 | Rt6 | 74 | Fnc4 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Fnc4 |  |  | Fnc4 |  |  |
| 0 | FADD | add | 8 | FSLT | set if less than |
| 1 | FSUB | subtract | 9 | FSGE | set if greater than or equal |
| 2 | FCMP | compare | 10 | FSLE | set if less than or equal |
| 3 | FMUL | multiply | 11 | FSGT | set if greater than |
| 4 | FDIV | divide | 12 | FSEQ | set if equal |
| 5 | {Float 1} | single operand instr. | 13 | FSNE | set if not equal |
| 6 | FMA | multiply-add | 14 | FNMA | negate multiply add |
| 7 | FMS | multiply-subtract | 15 | FNMS | negate multiply subtract |

Single Operand Register Form

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 26 | 25 20 | 19 14 | 13 8 | 7 4 | 3 0 |
| ~6 | Opcd6 | Ra6 | Rt6 | 74 | 54 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Opcd6 |  |  | Opcd6 |  |  |
| 0 | FMOV | move register | 16 | FTX | trigger fp exception |
| 1 |  |  | 17 | FCX | clear exception |
| 2 | FTOI | float to integer | 18 | FEX | enable exception |
| 3 | ITOF | integer to float | 19 | FDX | disable exception |
| 4 | FNEG | negate | 20 | FRM | set dynamic rounding mode |
| 5 | FABS | absolute value | 21 | FTRUNC | truncate |
| 6 | FSIGN | return sign of n | 22 |  |  |
| 7 | FMAN | return mantissa | 23 | FRES | reciprocal estimate |
| 8 | FNABS | negative abs | 24 |  |  |
| 9 |  |  | 25 |  |  |
| 10 |  |  | 26 |  |  |
| 11 |  |  | 27 |  |  |
| 12 | FSTAT | get float status | 28 |  |  |
| 13 | FSRQT | square root | 29 | FRSQRTE | reciprocal square root estimate |
| 14 | ISNAN | test for NaN | 30 | FCLASS | classify |
| 15 | FINITE | test for finite | 31 | UNORD | test for orderliness |

## Immediate Prefix Instruction

What happens when an immediate value is too large to be encoded in the instruction? An immediate prefix instruction is used to extend the range of the following instruction. There are two immediate prefix instructions when combined together with the constant in the instruction a full 64-bit constant may be used. Each immediate prefix encodes 26 bits more of the constant.

|  |  |  |  |
| --- | --- | --- | --- |
| 31 8 | 7 4 | 3 2 | 1 0 |
| imm24 | 154 | 0 | im2 |

Layout of instructions in memory.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 8 | | | 7 4 | 3 2 | 1 0 |
| imm24 | | | 154 | 1 | im2 |
| imm24 | | | 154 | 0 | im2 |
| imm12 | Ra6 | Rt6 | Group4 | Fn4 | |

Interrupts are not allowed to occur between a prefix and a following instruction.

### Floating-Point Interpretation of Immediate Fields

For a constant encoded directly in the instruction, the constant field is broken up into a five-bit exponent and seven-bit mantissa. The constant is assumed to be positive (sign bit is zero). The mantissa is the leading seven bits of the mantissa of a 64-bit double precision number. The five-bit exponent is biased by 1008, centering it around a zero exponent of 1023. This gives a range of 2-15 to 216 with seven significant bits.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 25 | 24 20 | 19 14 | 13 8 | 7 4 | 3 2 | 1 0 |
| mantiss7 | exp5 | Ra6 | Rt6 | Group4 | Fn4 | |

For constants using a single prefix instruction a 29-bit mantissa and eight-bit exponent are encoded. The exponent is biased by 896 centering it around a zero exponent of 1023. The sign bit for the number is also present. This allows numbers in the range 2-127 to 2128 with 29 significant bits. This is roughly equivalent to a single precision number with more bits of precision.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 8 | | | | | | 7 4 | 3 2 | 1 0 |
| mantissa20 | | | exp3 | | S | 154 | 0 | im2 |
| mantissa7 | exp5 | Ra6 | | Rt6 | | Group4 | Fn4 | |

For constants using a double immediate prefix an entire 64-bit double precision number may be specified.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 8 | | | | | | | 7 4 | 3 2 | 1 0 |
| mantissa21 | | | | | exp3 | | 154 | 1 | m2 |
| mantissa20 | | | | exp3 | | S | 154 | 0 | m2 |
| mantissa7 | exp5 | Ra6 | Rt6 | | | | Group4 | Fn4 | |

## Conditional Branches

All conditional branches share the same instruction format. Unlike the JAL instruction, branches use instruction-pointer relative addressing with a 12-bit branch displacement. They may branch up to +/- 8kB from the current instruction. The displacement is shifted left once before use.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 20 | 19 14 | 13 8 | 7 4 | 3 0 |
| Disp12 | Rb6 | Ra6 | 44 | Fnc4 |

Conditional branches compare two register values and branch according to the condition. For some branches register Ra may also be incremented or decremented by the branch instruction. Often at the end of a loop there is a counter or index that needs to be incremented or decremented. Combining this into the branch instruction increases code density. Note the location of the register fields in the instruction.

|  |  |  |
| --- | --- | --- |
| Fnc4 | Mnemonic | Operation |
| 0 | BLT | branch less than |
| 1 | BGE | branch greater than or equal |
| 2 | BLTU | branch less than unsigned |
| 3 | BGEU | branch if greater or equal unsigned |
| 4 | BEQ | branch if equal |
| 5 | BNE | branch if not equal |
| 6 | BBS | branch on bit set, Rb field specifies a bit 0 to 63 |
| 7 | BBC | branch on bit clear |
| 8 | AOBLT | add one and branch if less than |
| 9 | SOBGE | sub one and branch if greater or equal |
| 10 | AOBLTU | add one and branch if less than unsigned |
| 11 | SOBGEU | sub one and branch if greater than or equal |
| 12 | AOBNE | add one and branch if not equal |
| 13 | SOBNE | sub one and branch if not equal |
| 14 |  | reserved |
| 15 |  |  |

Note that BGT and BLE instructions can be used as they are the same instructions as BLT and BGE but with the operands switched around.

Branch on bit-set / clear branches are not supported in RiSCV. They fall under the category of brainiac instructions. The same thing can be accomplished using an and mask operation prior to a branch instruction.

## Load / Store Instructions

The only instructions accessing memory are load and store instructions. By restricting memory access to load and store instructions only, exception processing is greatly simplified. There’s no need to worry about how to restart an instruction in the middle of an operation if a memory exception occurs.

There are two addressing modes associated with load and store instructions, register indirect with displacement mode, and scaled indexed addressing mode. RiSCV only supports the register indirect with displacement mode. Register indirect with displacement addressing forms a memory address by adding a displacement field located in the instruction to the contents of a register. Scaled indexed addressing forms a memory address by adding the contents of two registers together. The second, indexing register may also be multiplied by a small constant (1, 2, 4 or 8) before the addition takes place. Scaled indexed addressing is useful in array processing and a few other cases. If the displacement field value is too large to allow register indirect with displacement addressing to be used, a value for the displacement may be loaded into a register and indexed addressing used instead. This conserves code space by removing the need for an additional add instruction and additional registers.

RiSCV attempts to maximize potential performance by fixing the location of register read and write ports and keeping the register fields single purpose. As such register Rb is used to read the data for a store, while Rt is used to load data from a read. Note that even in RiSCV some multiplexing is present in the register specification path to force the target register to zero in some circumstances. In an FPGA this is done with a lookup table to implement the ‘and’ gate required. In an FPGA however logic is not limited to an ‘and’ gate and more multiplexing may be present at little cost. Hence the address for the third read port required to support indexed store operations is multiplexed with the Rs field of the instruction.

Load instruction format – register indirect with displacement:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 20 | 19 14 | 13 8 | 7 4 | 3 0 |
| imm12 | Ra6 | Rt6 | 24 | Fnc4 |

Load instruction format – scaled indexed addressing:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 28 | 27 26 | 25 20 | 19 14 | 13 8 | 7 4 | 3 0 |
| ~4 | Sc2 | Rb6 | Ra6 | Rt6 | 24 | Fnc4 |

Group 2

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Register Indirect with Displacement | | | Scaled Indexed Addressing | | |
| Fnc4 | Mne. |  | Fnc4 | Mne. |  |
| 0 | LDB | load byte, sign extend | 8 | LDB |  |
| 1 | LDBU | load byte zero extend | 9 | LDBU |  |
| 2 | LDW | load wyde, sign extend | 10 | LDW |  |
| 3 | LDWU | load wyde, zero extend | 11 | LDWU |  |
| 4 | LDT | load tetra, sign extend | 12 | LDT |  |
| 5 | LDTU | load tetra, zero extend | 13 | LDTU |  |
| 6 | LDO | load octet | 14 | LDO |  |
| 7 |  |  | 15 |  |  |

Store instruction format – register indirect with displacement:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 20 | 19 14 | 13 8 | 7 4 | 3 0 |
| imm12 | Ra6 | Rs6 | 64 | Fnc4 |

Store instruction format – scaled indexed addressing:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 28 | 27 26 | 25 20 | 19 14 | 13 8 | 7 4 | 3 0 |
| ~4 | Sc2 | Rb6 | Ra6 | Rs6 | 64 | Fnc4 |

Note the store format differs from RiSCV in that the source register field is located where the target register field is normally located. This occurs because the Rb field of the instruction is used for indexed addressing.

Cache control instructions are lumped into the store instruction group as they have memory addressing requirements even though no memory is directly accessed. For a cache control the Rs6 field specifies the operation.

Group 6

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Register Indirect with Displacement | | | Scaled Indexed Addressing | | |
| Fnc4 | Mne. |  | Fnc4 | Mne. |  |
| 0 | STB | store byte | 8 | STB |  |
| 1 | CACHE |  | 9 | CACHE |  |
| 2 | STW | store wyde | 10 | STW |  |
| 3 |  |  | 11 |  |  |
| 4 | STT | store tetra | 12 | STT |  |
| 5 |  |  | 13 |  |  |
| 6 | STO | store octet | 14 | STO |  |
| 7 |  |  | 15 |  |  |