|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  | 🞂Table888seg  The segmented processor |
|  |  |
|  | **robfinch@finitron.ca** 🞂🞂5/18/2014 |

Table of Contents

[Preface 11](#_Toc388887106)

[Who This Book is For 11](#_Toc388887107)

[Motivation 11](#_Toc388887108)

[Qualifications: 12](#_Toc388887109)

[Choosing an Implementation Language 12](#_Toc388887110)

[Support Tools 13](#_Toc388887111)

[Documenting the Design 13](#_Toc388887112)

[Building the System 13](#_Toc388887113)

[Compilers for the Target Architecture 14](#_Toc388887114)

[Testing and Debugging 14](#_Toc388887115)

[Test Benches 15](#_Toc388887116)

[Bootstrap Code vs the “Real Code” 15](#_Toc388887117)

[Disabling Interrupts 15](#_Toc388887118)

[IRQ Live Indicator 16](#_Toc388887119)

[Disable Caching 16](#_Toc388887120)

[Stuck on a Bug ? 16](#_Toc388887121)

[The Rare Chance 16](#_Toc388887122)

[Design Choices 17](#_Toc388887123)

[RISC vs CISC 17](#_Toc388887124)

[Little Endian vs big Endian 17](#_Toc388887125)

[Deciding on the Degree of Pipelining 17](#_Toc388887126)

[Choosing a Bus Standard 18](#_Toc388887127)

[Choosing an ISA 18](#_Toc388887128)

[Readability 19](#_Toc388887129)

[Planning for the future 19](#_Toc388887130)

[Opcode / Instruction Size: 19](#_Toc388887131)

[Data Size 20](#_Toc388887132)

[Number of Registers: 20](#_Toc388887133)

[Register Access 21](#_Toc388887134)

[Segment Registers 22](#_Toc388887135)

[Other Registers 22](#_Toc388887136)

[Moving Register Values 22](#_Toc388887137)

[Register Usage 23](#_Toc388887138)

[Handling Immediate Values 23](#_Toc388887139)

[SETHI 25](#_Toc388887140)

[IMMxx 25](#_Toc388887141)

[LW Table 26](#_Toc388887142)

[Half-Operand Instructions 26](#_Toc388887143)

[The Branch Set 27](#_Toc388887144)

[Branch Targets 28](#_Toc388887145)

[Branch Instruction Format: 29](#_Toc388887146)

[Branch Prediction 29](#_Toc388887147)

[Looping Constructs 29](#_Toc388887148)

[Other Control Flow Instructions 29](#_Toc388887149)

[Subroutine Calls 29](#_Toc388887150)

[System Calls 30](#_Toc388887151)

[Returning From Subroutines 31](#_Toc388887152)

[Returning from Interrupt Routines 31](#_Toc388887153)

[Jumps 31](#_Toc388887154)

[Conditional Moves 32](#_Toc388887155)

[Predicated Instruction Execution 32](#_Toc388887156)

[Comparison Results: 33](#_Toc388887157)

[Arithmetic Operations 33](#_Toc388887158)

[Immediate Operate Functions 34](#_Toc388887159)

[Single Register Functions (The R table) 34](#_Toc388887160)

[Logical Operations 35](#_Toc388887161)

[Immediate Operate Functions 35](#_Toc388887162)

[Dual Register Functions (the RR table) 35](#_Toc388887163)

[Single register functions (The R table) 35](#_Toc388887164)

[Shift Instructions 36](#_Toc388887165)

[Other Instructions Reserved for Future Implementations 36](#_Toc388887166)

[Exception Handling 37](#_Toc388887167)

[Hardware Interrupts 37](#_Toc388887168)

[Getting and Putting Data 38](#_Toc388887169)

[Load / Store Multiple 39](#_Toc388887170)

[Load / Store Instructions 40](#_Toc388887171)

[The Stack 40](#_Toc388887172)

[Data Caching 41](#_Toc388887173)

[Address Modes: 41](#_Toc388887174)

[Support for Semaphores 42](#_Toc388887175)

[Pipeline Design 44](#_Toc388887176)

[Processor Stages / States 44](#_Toc388887177)

[RESET 44](#_Toc388887178)

[IFETCH 44](#_Toc388887179)

[DECODE 45](#_Toc388887180)

[EXECUTE 45](#_Toc388887181)

[Memory Stage: 45](#_Toc388887182)

[LOAD1 45](#_Toc388887183)

[LOAD2 45](#_Toc388887184)

[LOAD3 45](#_Toc388887185)

[LOAD4 45](#_Toc388887186)

[STORE1 45](#_Toc388887187)

[STORE2 45](#_Toc388887188)

[STORE3 45](#_Toc388887189)

[STORE4 46](#_Toc388887190)

[Instruction Fetch: 46](#_Toc388887191)

[Instruction Cache 46](#_Toc388887192)

[Decode 47](#_Toc388887193)

[Register File Access 47](#_Toc388887194)

[Execute 47](#_Toc388887195)

[Nice-to-Have Hardware Features 48](#_Toc388887196)

[Implementing the Processor 49](#_Toc388887197)

[Convenience Tasks 49](#_Toc388887198)

[next\_state(); 49](#_Toc388887199)

[wb\_xxxx(); 49](#_Toc388887200)

[Implementing Processor Reset 49](#_Toc388887201)

[Implementing the IFETCH stage 50](#_Toc388887202)

[Implementing the Program Counter 50](#_Toc388887203)

[Implementing the Instruction Cache 52](#_Toc388887204)

[Implementing Uncached Instruction Access 57](#_Toc388887205)

[Implementing Hardware Interrupts 58](#_Toc388887206)

[Register Read Access 59](#_Toc388887207)

[Implementing the DECODE stage 60](#_Toc388887208)

[Implementing Register Operands 60](#_Toc388887209)

[Implementing Immediates 60](#_Toc388887210)

[Implementing Target Register Selection 63](#_Toc388887211)

[Implementing the EXECUTE Stage 63](#_Toc388887212)

[Implementing Branches 63](#_Toc388887213)

[Implementing the JMP Instruction 64](#_Toc388887214)

[Implementing the JSR Instruction 65](#_Toc388887215)

[Implementing the JSR (address,Rn) and JMP (address,Rn) Instructions 66](#_Toc388887216)

[Implementing the CMP Instruction 66](#_Toc388887217)

[Implementing Arithmetic and Logical Instructions 67](#_Toc388887218)

[Implementing Multiply and Divide 68](#_Toc388887219)

[Implementing Shift Operations 72](#_Toc388887220)

[Implementing the Memory Stage 72](#_Toc388887221)

[Implementing Loads 72](#_Toc388887222)

[Implementing Stores 74](#_Toc388887223)

[Implementing the Stack Pointer 76](#_Toc388887224)

[Implementing Stack PUSH / POP operations 78](#_Toc388887225)

[Implementing the Writeback Stage 79](#_Toc388887226)

[Implementing Register Updates 79](#_Toc388887227)

[Instruction Set Description 80](#_Toc388887228)

[ADD - addition 81](#_Toc388887229)

[Instruction Formats 81](#_Toc388887230)

[Operation 81](#_Toc388887231)

[AND – bitwise logical ‘and’ 82](#_Toc388887232)

[Instruction Formats 82](#_Toc388887233)

[Operation 82](#_Toc388887234)

[ANDN – bitwise logical ‘and’ with complement 83](#_Toc388887235)

[Instruction Formats 83](#_Toc388887236)

[Operation 83](#_Toc388887237)

[ASR – Arithmetic Shift Right 84](#_Toc388887238)

[Instruction Formats 84](#_Toc388887239)

[Operation 84](#_Toc388887240)

[Bcc – Branches 85](#_Toc388887241)

[Instruction Formats 85](#_Toc388887242)

[Operation 85](#_Toc388887243)

[BRK – Breakpoint 86](#_Toc388887244)

[Instruction Formats 86](#_Toc388887245)

[Operation 86](#_Toc388887246)

[BSR – Branch to Subroutine 87](#_Toc388887247)

[Instruction Formats 87](#_Toc388887248)

[Operation 87](#_Toc388887249)

[CLI – Clear Interrupt Mask 88](#_Toc388887250)

[Instruction Formats 88](#_Toc388887251)

[Operation 88](#_Toc388887252)

[CMP - Comparison 89](#_Toc388887253)

[Instruction Formats 89](#_Toc388887254)

[Operation 89](#_Toc388887255)

[COM – bitwise ones complement 90](#_Toc388887256)

[Instruction Formats 90](#_Toc388887257)

[Operation 90](#_Toc388887258)

[DIV - Division 91](#_Toc388887259)

[Instruction Formats 91](#_Toc388887260)

[Operation 91](#_Toc388887261)

[EOR – bitwise logical exclusive ‘or’ 92](#_Toc388887262)

[Instruction Formats 92](#_Toc388887263)

[Operation 92](#_Toc388887264)

[ENOR – complement bitwise logical exclusive ‘or’ 93](#_Toc388887265)

[Instruction Formats 93](#_Toc388887266)

[Operation 93](#_Toc388887267)

[IMMx – Immediate Prefix 94](#_Toc388887268)

[Instruction Formats 94](#_Toc388887269)

[Operation 94](#_Toc388887270)

[JMP – Jump 95](#_Toc388887271)

[Instruction Formats 95](#_Toc388887272)

[Operation 95](#_Toc388887273)

[JSR – Jump to Subroutine 96](#_Toc388887274)

[Instruction Formats 96](#_Toc388887275)

[Operation 96](#_Toc388887276)

[LB – Load Byte with Sign Extend 97](#_Toc388887277)

[Instruction Formats 97](#_Toc388887278)

[Operation 97](#_Toc388887279)

[LBU – Load Byte with Zero Extend 98](#_Toc388887280)

[Instruction Formats 98](#_Toc388887281)

[Operation 98](#_Toc388887282)

[LC – Load Character with Sign Extend 99](#_Toc388887283)

[Instruction Formats 99](#_Toc388887284)

[Operation 99](#_Toc388887285)

[LCU – Load Character with Zero Extend 100](#_Toc388887286)

[Instruction Formats 100](#_Toc388887287)

[Operation 100](#_Toc388887288)

[LDI – Load Immediate 101](#_Toc388887289)

[Instruction Formats 101](#_Toc388887290)

[Operation 101](#_Toc388887291)

[LH – Load Half-Word with Sign Extend 102](#_Toc388887292)

[Instruction Formats 102](#_Toc388887293)

[Operation 102](#_Toc388887294)

[LHU – Load Half-Word with Zero Extend 103](#_Toc388887295)

[Instruction Formats 103](#_Toc388887296)

[Operation 103](#_Toc388887297)

[LW – Load Word 104](#_Toc388887298)

[Instruction Formats 104](#_Toc388887299)

[Operation 104](#_Toc388887300)

[MFSPR – Move from Special Register 105](#_Toc388887301)

[Instruction Formats 105](#_Toc388887302)

[Operation 105](#_Toc388887303)

[MOD – Signed Modulus 106](#_Toc388887304)

[Instruction Formats 106](#_Toc388887305)

[Operation 106](#_Toc388887306)

[MODU – Unsigned Modulus 107](#_Toc388887307)

[Instruction Formats 107](#_Toc388887308)

[Operation 107](#_Toc388887309)

[MOV – Move Register to Register 108](#_Toc388887310)

[Instruction Formats 108](#_Toc388887311)

[Operation 108](#_Toc388887312)

[MTSPR – Move to Special Register 109](#_Toc388887313)

[Instruction Formats 109](#_Toc388887314)

[Operation 109](#_Toc388887315)

[MUL – Signed Multiply 110](#_Toc388887316)

[Instruction Formats 110](#_Toc388887317)

[Operation 110](#_Toc388887318)

[MULU – Unsigned Multiply 111](#_Toc388887319)

[Instruction Formats 111](#_Toc388887320)

[Operation 111](#_Toc388887321)

[NAND – Complement Bitwise Logical ‘And’ 112](#_Toc388887322)

[Instruction Formats 112](#_Toc388887323)

[Operation 112](#_Toc388887324)

[NEG – Negate 113](#_Toc388887325)

[Instruction Formats 113](#_Toc388887326)

[Operation 113](#_Toc388887327)

[NOP – No Operation 114](#_Toc388887328)

[Instruction Formats 114](#_Toc388887329)

[Operation 114](#_Toc388887330)

[NOR – Complement Bitwise Logical ‘Or’ 115](#_Toc388887331)

[Instruction Formats 115](#_Toc388887332)

[Operation 115](#_Toc388887333)

[NOT – Not 116](#_Toc388887334)

[Instruction Formats 116](#_Toc388887335)

[Operation 116](#_Toc388887336)

[OR – bitwise logical or 117](#_Toc388887337)

[Instruction Formats 117](#_Toc388887338)

[Operation 117](#_Toc388887339)

[ORN – Bitwise Logical Or with Complement 118](#_Toc388887340)

[Instruction Formats 118](#_Toc388887341)

[Operation 118](#_Toc388887342)

[PHP – Push Processor Status 119](#_Toc388887343)

[Instruction Formats 119](#_Toc388887344)

[Operation 119](#_Toc388887345)

[PLP – Pull Processor Status 120](#_Toc388887346)

[Instruction Formats 120](#_Toc388887347)

[Operation 120](#_Toc388887348)

[POP – Pop Register 121](#_Toc388887349)

[Instruction Formats 121](#_Toc388887350)

[Operation 121](#_Toc388887351)

[PUSH – Push Register 122](#_Toc388887352)

[Instruction Formats 122](#_Toc388887353)

[Operation 122](#_Toc388887354)

[ROL – Rotate Left 123](#_Toc388887355)

[Instruction Formats 123](#_Toc388887356)

[Operation 123](#_Toc388887357)

[ROR – Rotate Right 124](#_Toc388887358)

[Instruction Formats 124](#_Toc388887359)

[Operation 124](#_Toc388887360)

[RTI – Return From Interrupt 125](#_Toc388887361)

[Instruction Formats 125](#_Toc388887362)

[Operation 125](#_Toc388887363)

[RTS – Return From Subroutine 126](#_Toc388887364)

[Instruction Formats 126](#_Toc388887365)

[Operation 126](#_Toc388887366)

[SB – Store Byte 127](#_Toc388887367)

[Instruction Formats 127](#_Toc388887368)

[Operation 127](#_Toc388887369)

[SC – Store Character 128](#_Toc388887370)

[Instruction Formats 128](#_Toc388887371)

[Operation 128](#_Toc388887372)

[SH – Store Half-Word 129](#_Toc388887373)

[Instruction Formats 129](#_Toc388887374)

[Operation 129](#_Toc388887375)

[SHL – Shift Left 130](#_Toc388887376)

[Instruction Formats 130](#_Toc388887377)

[Operation 130](#_Toc388887378)

[SHR – Shift Right 131](#_Toc388887379)

[Instruction Formats 131](#_Toc388887380)

[Operation 131](#_Toc388887381)

[SW – Store Word 132](#_Toc388887382)

[Instruction Formats 132](#_Toc388887383)

[Operation 132](#_Toc388887384)

[SXB – Sign Extend Byte 133](#_Toc388887385)

[Instruction Formats 133](#_Toc388887386)

[Operation 133](#_Toc388887387)

[SXC – Sign Extend Character 134](#_Toc388887388)

[Instruction Formats 134](#_Toc388887389)

[Operation 134](#_Toc388887390)

[SXH – Sign Extend Half-Word 135](#_Toc388887391)

[Instruction Formats 135](#_Toc388887392)

[Operation 135](#_Toc388887393)

[Glossary 137](#_Toc388887394)

[FPGA: 137](#_Toc388887395)

[HDL 137](#_Toc388887396)

[Instruction Bundle: 137](#_Toc388887397)

[ISA: 137](#_Toc388887398)

[Program Counter: 137](#_Toc388887399)

[SIMD: 137](#_Toc388887400)

[Stack Pointer 137](#_Toc388887401)

[Resources: 138](#_Toc388887402)

[Reference Material 138](#_Toc388887403)

[Major Opcode Table 140](#_Toc388887404)

[Func Table for RR instructions 141](#_Toc388887405)

[Func Table for R instructions 142](#_Toc388887406)

[01 Func Table 143](#_Toc388887407)

[02 Func Table 144](#_Toc388887408)

# Preface

## Who This Book is For

This book is for the FPGA enthusiast who’s already had a look at the Table888 processor. It’s advisable that one have a fairly good background in digital electronics and computer systems before trying to read this book. This book uses examples in the Verilog language, it would be helpful to have some understanding of HDL languages. Finally, this book contains a lot about memory management, some previous knowledge would also be helpful. If you’re into electronics and computers as a hobby FPGA’s can be a lot of fun. This book attempts to be ‘hands-on’ in nature and provides sample program code.

## Motivation

Continuing in the flavor of Table888. This book reflects my desire to add memory management capability to the Table888 processor including memory protection mechanisms.

## Qualifications:

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

# Testing and Debugging

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

## Test Benches

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

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

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

which turns on all the LEDs on the board.

## Bootstrap Code vs the “Real Code”

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

## Disabling Interrupts

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

## Disable Caching

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

# Design Choices

Segmentation and paging. Segmentation and paging are the two main choices for memory management beyond some simpler mechanisms like bank switching. The goal for Table888 is to eventually implement both. Although segmentation has fallen out of favor somewhat it is still used.

As part of the memory management portion of a cpu segment registers are often provided. There are usually multiple segment registers in order to support multiple segments which are typically part of a program. Common program segment are: the code segment, the data segment, the uninitialized data segment and the stack segment. There are often other segments as well. 80x88 is famous for its segment registers, but other processors like IBM’s PowerPC also use them as well. Segment registers are a fairly easy to understand , and a low cost memory management approach. The memory address 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. Once a design seems to be working well, I tend to add segment registers to the design as a first step at providing memory management features. Table888seg includes a set of sixteen segment registers.

## Planning for the future

## Number of Registers:

The number of segment registers that are useful seems not to be quantified as closely as the number of general purpose registers. However, four registers was deemed not enough for the 80x86 architecture and two more segment registers were added. Also a couple of additional registers in the 80x86 design were added to support the segmentation architecture and they act a lot like segment registers. These include the task register and the local descriptor table register. So we have about eight segment registers in the 80x86 architecture. Table888seg uses an array of sixteen segment registers. Segments registers are typically initialized to a flat memory model then forgotten about.

### Register Access

### Segment Registers

### Other Registers

### Moving Register Values

### Register Usage

## Other Control Flow Instructions

### Subroutine Calls

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

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

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

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

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

### System Calls

System calls are used to call the system. They are often called software interrupts or traps. The 80x88 uses the name ‘int’. 6809 calls this a ‘SWI’ for software interrupt. In 6502 parlance it’s the BRK instruction. They are called TRAPs on the 680x0 series. All these instructions do much the same thing. They are almost like a jump to subroutine instruction with an implied address. The system call instruction usually saves more machine state on the stack than a subroutine call would. These instructions may also switch the processor operating mode into a more protected level. Table888 calls this a break (BRK) instruction. The format of the BRK instruction is shown below:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ~19 | Vector9 | 04 | 008 | BRK |

The program counter register low order 13 bit are loaded directly from the BRK instruction. The high order program counter bits come from the vbr register.

### Returning From Subroutines

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

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

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

### Returning from Interrupt Routines

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

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

### Jumps

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

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

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

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

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

## Comparison Results:

## Arithmetic Operations

## Other Instructions Reserved for Future Implementations

## Exception Handling

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

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

## Hardware Interrupts

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

## Getting and Putting Data

# Implementing the Processor

This section describes the details of implementing the processor.

## Convenience Tasks

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

### next\_state();

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

### wb\_xxxx();

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

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

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

## Implementing Processor Reset

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

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

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

The processor has to start executing instructions somewhere, so the PC needs to be set at reset. It is set to 32’h0000FFF0.

## Implementing the IFETCH stage

### Implementing the Program Counter

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

|  |
| --- |
| function [31:0] pc\_inc;  input [31:0] pc;  begin  case(pc[3:0])  4'd0: pc\_inc = {pc[31:4],4'd5};  4'd5: pc\_inc = {pc[31:4],4'd10};  4'd10: pc\_inc = {pc[31:4],4'd0} + 32'd16;  default: pc\_inc = 32'h0000FFB0;  endcase  end  endfunction |

Note that if the program counter becomes unaligned it is automatically set to the alignment fault vector (32’h0000FFB0).The least significant four bits of the program counter should always be one of 0h, 5h or Ah. There isn’t much that can be done for a program where the program counter is out of alignment. There’s no telling what happened to the program. Jumping directly to the alignment fault vector doesn’t allow the processor to stack information as it would for other exceptions.

At processor reset the program counter is forced to 32’h0000FFF0. There should be a jump or branch instruction to boot code located there.

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

|  |
| --- |
| if (!hwi)  pc <= pc\_inc(pc); |

If a hardware interrupt is taking place, the program counter is not incremented. We want the program counter to remain pointed to the interrupted instruction (which hasn’t had a chance to execute yet), and not the next instruction.

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

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

Note there is a two cycle delay incurred after any operation that changes the program counter. this is to allow time for the altered program counter address to percolate through the instruction cache.

### Implementing the Instruction Cache

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

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

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

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

#### The Cache Ram

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

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

Because a synchronous RAM is used, there is a one cycle delay from the time it is addressed by the program counter, until it outputs data.

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

|  |
| --- |
| module icache\_ram(wclk, wr, wa, i, rclk, pc, insn, debug\_bits);  input wclk;  input wr;  input [12:0] wa;  input [31:0] i;  input rclk;  input [12:0] pc;  output reg [39:0] insn;  output reg [1:0] debug\_bits;  wire [31:0] o1,o2,o3,o4;  syncram\_512x32\_1rw1r u1 (wclk, wr && wa[3:2]==2'b00, wa[12:4], i, rclk, pc[12:4], o1);  syncram\_512x32\_1rw1r u2 (wclk, wr && wa[3:2]==2'b01, wa[12:4], i, rclk, pc[12:4], o2);  syncram\_512x32\_1rw1r u3 (wclk, wr && wa[3:2]==2'b10, wa[12:4], i, rclk, pc[12:4], o3);  syncram\_512x32\_1rw1r u4 (wclk, wr && wa[3:2]==2'b11, wa[12:4], i, rclk, pc[12:4], o4);  wire [127:0] bundle = {o4,o3,o2,o1};  always @(posedge rclk)  case(pc[3:0])  4'h0: insn <= bundle[ 39: 0];  4'h5: insn <= bundle[ 79:40];  4'hA: insn <= bundle[119:80];  default: insn <= 40'h0000FFB0\_50; // JMP Alignment fault  endcase  always @(posedge rclk)  case(pc[3:0])  4'h0: debug\_bits <= bundle[121:120];  4'h5: debug\_bits <= bundle[123:122];  4'hA: debug\_bits <= bundle[125:124];  default: debug\_bits <= 2'b00;  endcase  endmodule |

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

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

The tag ram is fed a cache reset signal during processor reset in order to invalidate the cache.

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

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

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

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

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

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

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

#### Implementing Cache Invalidates

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

### Implementing Uncached Instruction Access

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

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

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

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

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

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

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

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

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

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

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

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

### Implementing Hardware Interrupts

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

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

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

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

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

Maskable interrupts use a vector table to determine the address of the interrupt subroutine. The vector number, used in the calculation of the address, is loaded by the processor from the vect\_i signal. The vect\_i signal is supplied externally by the interrupt controller when an interrupt occurs. The vect\_i signal is a private bus. Often an interrupt controller communicates the interrupt vector using the regular bus, however it is simple just to use a private bus. The break (BRK) instruction is used for processing the interrupt.

### Register Read Access

One of the things occurring at the IFETCH stage is addressing of registers for read access. In order to be able to read registers as soon as possible, the register read fields are all set in fixed positions in the ISA. Because of this the register read register numbers field can be a simple bit-select directly from the instruction register. Note there is no intervening logic to slow things down:

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

Another item to take place is bypassing register zero to zero, and setting up read of the stack pointer and program counter. These are handled by three multiplexors, one for each read port. One of the multiplexors is shown below:

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

The register file itself is implemented as an array of 64 bit registers.

|  |
| --- |
| reg [63:0] regfile [255:0]; |

## Implementing the DECODE stage

### Implementing Register Operands

Register file values from the register file are latched into operand registers ‘a’, ‘b’, and ’c’ at the start of the DECODE stage.

|  |
| --- |
| a <= rfoa;  b <= rfob;  c <= rfoc; |

### Implementing Immediates

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

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

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

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

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

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

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

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

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

### Implementing Target Register Selection

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

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

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

## Implementing the EXECUTE Stage

### Implementing Branches

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

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

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

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

### Implementing the JMP Instruction

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

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

### Implementing the JSR Instruction

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

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

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

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

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

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

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

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

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

### Implementing the CMP Instruction

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

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

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

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

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

### Implementing Arithmetic and Logical Instructions

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

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

### Implementing Multiply and Divide

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

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

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

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

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

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

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

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

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

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

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

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

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

### Implementing Shift Operations

## Implementing the Memory Stage

### Implementing Loads

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

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

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

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

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

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

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

### Implementing Stores

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

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

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

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

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

### Implementing the Stack Pointer

The stack pointer needs to be updated at the same time that a register load from the stack is taking place. This would require two writes ports on the register file. It isn’t cost effective to implement two write ports on the entire register file in order to support a single register. In order to implement two writes at the same time, the stack pointer has it’s own register separate from the general register file. Reading this register is handled by the same multiplexor that puts the zero constant on the register output when R0 is read. Note that this multiplexor is repeated three times, once for each read port.

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

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

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

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

### Implementing Stack PUSH / POP operations

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

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

## Implementing the Writeback Stage

### Implementing Register Updates

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

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

# Instruction Set Description

A description of the instruction set follows.

## IMMx – Immediate Prefix

IMM1 #i32

IMM2 #i32

Instruction Formats:

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

Operation:

IMM1: constant buffer = sign extend (immediate32)

IMM2: constant buffer[63:32] = immediate32

Notes:

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

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

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

## JMP – Jump

JMP abs

JMP (abs,Rn)

JMP d(Rn)

Instruction Formats:

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

Operation:

#### Absolute Address Form

PC = Address32

#### Memory Indexed Indirect Form

PC = memory[address + Rn]

#### Register Indirect with Displacement Form

PC = displacement + Rn

Notes:

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

## JSR – Jump to Subroutine

JSR abs

JSR (abs,Rn)

JSR d(Rn)

Instruction Formats:

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

With Selector Prefix:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| RPL3 | TI | Index20 | Addr8 | 618h | JSP |
| Address32 | | | | 518h | JSR address |

Operation:

#### Absolute Address Form

SP = SP - 8

memory[SP] = PC

PC = Address32

#### Memory Indexed Indirect Form

SP = SP - 8

memory[SP] = PC

PC = memory[address + Rn]

#### Register Indirect with Displacement Form

SP = SP - 8

memory[SP] = PC

PC = displacement + Rn

Notes:

The addressing may be extended up to 61 bits with a selector prefix instruction.

If the selector points to a jump gate then the address portion of the instruction is ignored. Jumping to a gate routine can be done more compactly with the JGR instruction.

## MFSEG – Move from Segment Register

MFSEG Rt, Seg

Instruction Formats:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0D8h | ~8 | Rt8 | ~4 | Sg4 | 018h | MTSEG Seg,Ra |

Operation:

#### Register-Register Form

Rt= Seg << 40

Notes:

The segment register is moved to the general purpose register. Note that the selector value from the segment register is placed in the upper 24 bits of the register. Unlike the MTSEG instruction this instruction does not cause protection checks, or loading of the segment.

## MSO – Merge Segment with Offset

MSO Rt, Ra, Seg

Instruction Formats:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 298 | Rt8 | ~4 | Sg4 | Ra8 | 028 | MSO Rt,Ra,Seg |

Operation:

Rt = (Ra & $FFFFFFFFFF) | (selectorof(Sg) << 24)

Notes:

Use this instruction to merge a pointer with it’s segment selector into a single register. This is typically used to compact a pointer for storage in memory.

|  |
| --- |
| ; First, save the current stack pointer and segment  mso sp,sp,ss ; build stack pointer incl. selector  shr r250,r251,#61 ; get originating privilege level  sw sp,TCB\_SP0Save[tr+r250\*8] ; save original stack pointer |

This instruction is equivalent to the following two instructions: mfseg, followed by or r,r,r2.

## MTSEG – Move to Segment Register

MTSEG Seg, Ra

Instruction Formats:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0C8h | ~8 | ~4 | Sg4 | Ra8 | 018h | MTSEG Seg,Ra |

Operation:

#### Register-Register Form

Seg = Ra >> 40

load\_segment(Seg)

Notes:

The general purpose register is moved to the segment register. The segment descriptor is loaded from either the local or global descriptor table. Note that the segment selector value should be in the upper 24 bits of the register.

When the segment is loaded the processor performs protection checks prior to loading the segment register. Exceptions that may occur are segment not present, privilege violation, and segment type violation.

## SSO – Split Segment and Offset

SSO Rt, Ra, Seg

Instruction Formats:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 298 | Rt8 | ~4 | Sg4 | Ra8 | 028 | SSO Rt,Ra,Seg |

Operation:

Rt = Ra & $FFFFFFFFFF

Seg = load\_segment(Ra >> 40)

Notes:

Use this instruction to split a pointer into it’s segment and offset as separate registers. The top 24 bits of the register Ra are used as a selector to load a segment register. The lower 40 bits of Ra are used as an offset and transferred to the target register Rt. This is typically used after loading a compacted pointer from memory.

|  |
| --- |
| ; Second, setup a stack at this privilege level  mfseg r250,cs ; get the current privilege level from the cs selector  shr r250,r250,#61  lw sp,TCB\_SP0Save[tr+r250\*8] ; get stack pointer according to privilege level  sso sp,sp,ss |

This instruction is equivalent to the following two instructions: mtseg, followed by and r,r,#$FFFFFFFFFF.

# Glossary

## FPGA:

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

HDL:

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

## Instruction Bundle:

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

## ISA:

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

## Program Counter:

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

## SIMD:

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

Stack Pointer:

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

# Resources:

Source code for the latest release of Table888 and others is available at my github account. <http://github.com/robfinch/Cores/blob/master/Table888/Table888.html>

Visit my website: <http://www.finitron.ca> for a number of examples of working (and not working) HDL code.

comp.arch newsgroup is a newsgroup about computer architecture.

http://OpenCores.org is an organization with lots and lots of examples of HDL code including a number of processors. I’ve studied numerous processor architectures available there.

## Reference Material

Below is a short list of some of the reading material I’ve studied. I’ve downloaded a fair number of documents on computer architecture from the web. Too many to list.

*Computer Architecture A Quantitative Approach, Second Edition, by John L Hennessy & David Patterson, published by Morgan Kaufman Publishers, Inc. San Franciso, California* is a good book on computer architecture. There is a newer edition of the book available.

PowerPC Microprocessor Developer’s Guide, SAMS publishing. 201 West 103rd Street, Indianapolis, Indiana, 46290

80386/80486 Programming Guide by Ross P. Nelson, Microsoft Press

Programming the 286, C. Vieillefond, SYBEX, 2021 Challenger Drive #100, Alameda, CA 94501

Tech. Report UMD-SCA-2000-02 ENEE 446: Digital Computer Design — An Out-of-Order RiSC-16

Programming the 65C816, David Eyes and Ron Lichty, Western Design Centre Inc.

Microprocessor Manuals from Motorola, and Intel,

The SPARC Architecture Manual Version 8, SPARC International Inc, 535 Middlefield Road. Suite210 Menlo Park California, CA 94025

The SPARC Architecture Manual Version 9, SPARC International Inc, Sab Jose California, PTR Prentice Hall, Englewood Cliffs, New Jersey, 07632

The MMIX processor: http://mmix.cs.hm.edu/doc/instructions-en.html

# Major Opcode Table

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

# Func Table for RR instructions

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

# Func Table for R instructions

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

## 01 Func Table

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Func8 |  |  | |  | | Opcode8 |  |
| Call Code32 | | | | | | 00h | BRK |
| Func8 |  | | Rt8 | | Ra8 | 01h | {R} |
| 04h |  | | Rt8 | | Ra8 | 01h | MOV |
| 05h |  | | Rt8 | | Ra8 | 01h | NEG |
| 06h |  | | Rt8 | | Ra8 | 01h | COM |
| 07h |  | | Rt8 | | Ra8 | 01h | NOT |
| 08h |  | | Rt8 | | Ra8 | 01h | SXB |
| 09h |  | | Rt8 | | Ra8 | 01h | SXC |
| 0Ah |  | | Rt8 | | Ra8 | 01h | SXH |
| 30h |  | |  | |  | 01h | SEI |
| 31h |  | |  | |  | 01h | CLI |
| 32h |  | |  | |  | 01h | PHP |
| 33h |  | |  | |  | 01h | PLP |
| 34h |  | |  | |  | 01h | ICON |
| 35h |  | |  | |  | 01h | ICOFF |
| 40h |  | |  | |  | 01h | RTI |
| 48h |  | |  | |  | 01h | MTSPR |
| 49h |  | |  | |  | 01h | MFSPR |

## 02 Func Table

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

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