|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  | 🞂Table888mmu  The Memory Management Unit |
|  |  |
|  | **robfinch@finitron.ca** 🞂🞂5/18/2014 |

Table of Contents

[Preface 5](#_Toc389958230)

[Who This Book is For 5](#_Toc389958231)

[What this Book is About 5](#_Toc389958232)

[Motivation 5](#_Toc389958233)

[Testing and Debugging 5](#_Toc389958234)

[Additional Registers 6](#_Toc389958235)

[Page Table Address (SPR 04 or CR3) 6](#_Toc389958236)

[Control Register Zero (SPR 05 or CR0) 6](#_Toc389958237)

[Clock Register (SPR 06) 6](#_Toc389958238)

[Fault PC (SPR 08) and Fault CS (SPR 09) 7](#_Toc389958239)

[Design Choices 7](#_Toc389958240)

[Segmentation and Paging 7](#_Toc389958241)

[Segmentation Overview 7](#_Toc389958242)

[Address Formation: 8](#_Toc389958243)

[Paging Overview 8](#_Toc389958244)

[Segmentation 9](#_Toc389958245)

[Number of Registers: 9](#_Toc389958246)

[Segment Usage Conventions 10](#_Toc389958247)

[Composition 11](#_Toc389958248)

[Selector Format 11](#_Toc389958249)

[Register Access 11](#_Toc389958250)

[Moving Register Values 12](#_Toc389958251)

[Merge Format: 12](#_Toc389958252)

[Subroutine Calls 13](#_Toc389958253)

[Call Gates 13](#_Toc389958254)

[Task Gates 14](#_Toc389958255)

[System Calls 15](#_Toc389958256)

[Returning From Subroutines 15](#_Toc389958257)

[Returning from Interrupt Routines 17](#_Toc389958258)

[Hardware Interrupts 17](#_Toc389958259)

[Interrupt Descriptor Table Entries 18](#_Toc389958260)

[Paging 19](#_Toc389958261)

[Registers: 20](#_Toc389958262)

[Moving Register Values 20](#_Toc389958263)

[Page Tables 20](#_Toc389958264)

[The Page Table Address Register 20](#_Toc389958265)

[The Stack 21](#_Toc389958266)

[Stack Overflow 21](#_Toc389958267)

[Stack Underflow 21](#_Toc389958268)

[Exception Handling 21](#_Toc389958269)

[Implementing the Processor 21](#_Toc389958270)

[Convenience Tasks 22](#_Toc389958271)

[next\_state(); 22](#_Toc389958272)

[wb\_xxxx(); 22](#_Toc389958273)

[Implementing Processor Reset 22](#_Toc389958274)

[Implementing the IFETCH stage 23](#_Toc389958275)

[Implementing the Program Counter 23](#_Toc389958276)

[Implementing Hardware Interrupts 25](#_Toc389958277)

[Register Read Access 26](#_Toc389958278)

[Implementing the DECODE stage 27](#_Toc389958279)

[Implementing Register Operands 27](#_Toc389958280)

[Implementing the EXECUTE Stage 27](#_Toc389958281)

[Implementing Branches 27](#_Toc389958282)

[Implementing the JMP Instruction 28](#_Toc389958283)

[Implementing the JSR Instruction 28](#_Toc389958284)

[Implementing the JSR (address,Rn) and JMP (address,Rn) Instructions 29](#_Toc389958285)

[Implementing the Memory Stage 30](#_Toc389958286)

[Implementing Loads 30](#_Toc389958287)

[Implementing the Stack Pointer 30](#_Toc389958288)

[Implementing Stack PUSH / POP operations 31](#_Toc389958289)

[Implementing the Writeback Stage 31](#_Toc389958290)

[Implementing Register Updates 31](#_Toc389958291)

[Instruction Set Description 32](#_Toc389958292)

[BFCHG – Bitfield Change 33](#_Toc389958293)

[BFCLR – Bitfield Clear 33](#_Toc389958294)

[BFEXT – Bitfield Extract 33](#_Toc389958295)

[BFEXTU – Bitfield Extract Unsigned 33](#_Toc389958296)

[BFINS – Bitfield Insert 33](#_Toc389958297)

[BFSET – Bitfield Set 34](#_Toc389958298)

[IMMx – Immediate Prefix 35](#_Toc389958299)

[Instruction Formats 35](#_Toc389958300)

[Operation 35](#_Toc389958301)

[JMP – Jump 36](#_Toc389958302)

[Instruction Formats 36](#_Toc389958303)

[Operation 36](#_Toc389958304)

[JSR – Jump to Subroutine 37](#_Toc389958305)

[Instruction Formats 37](#_Toc389958306)

[Operation 37](#_Toc389958307)

[LAR – Load Access Rights 38](#_Toc389958308)

[Instruction Formats 38](#_Toc389958309)

[Operation 38](#_Toc389958310)

[LSL – Load Segment Limit 39](#_Toc389958311)

[Instruction Formats 39](#_Toc389958312)

[Operation 39](#_Toc389958313)

[MFSEG – Move from Segment Register 40](#_Toc389958314)

[Instruction Formats 40](#_Toc389958315)

[Operation 40](#_Toc389958316)

[MTSEG – Move to Segment Register 41](#_Toc389958317)

[Instruction Formats 41](#_Toc389958318)

[Operation 41](#_Toc389958319)

[Glossary 42](#_Toc389958320)

[FPGA: 42](#_Toc389958321)

[HDL 42](#_Toc389958322)

[Instruction Bundle: 42](#_Toc389958323)

[ISA: 42](#_Toc389958324)

[Program Counter: 42](#_Toc389958325)

[SIMD: 42](#_Toc389958326)

[Stack Pointer 42](#_Toc389958327)

[Resources: 43](#_Toc389958328)

[Reference Material 43](#_Toc389958329)

[Major Opcode Table 45](#_Toc389958330)

[Func Table for RR instructions 46](#_Toc389958331)

[Func Table for R instructions 47](#_Toc389958332)

[01 Func Table 48](#_Toc389958333)

[02 Func Table 49](#_Toc389958334)

# 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.

## What this Book is About

This book is about the memory management model used in the Table888m processor. The processor includes both segmentation and paging units.

## 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. When one’s worked with enough software one can really appreciate the presence of memory protection. A protected memory system removes a lot of the worry about what program interfered with the piece of software one’s trying to work on.

# Testing and Debugging

The advice provided in Table888 works just as well for getting the segmented processor up and running. First, get the processor running in a non-segmented, non-protected mode (Get Rid of Complexity). It helps to have an address range available that doesn’t respond to segmentation or paging for bootstrapping the processor.

# Additional Registers

There are several more special purpose registers in the advanced version of the Table888 processor. These registers are accessible with the MFSPR and MTSPR instructions.

## Page Table Address (SPR 04 or CR3)

This register is described fully in the section on paged memory management. This register contains the base address of the page table in memory.

|  |  |  |
| --- | --- | --- |
| Address63..12 | ~9 | TS3 |

## Control Register Zero (SPR 05 or CR0)

This register contains bits to enable paged memory management and protected mode.

|  |  |  |
| --- | --- | --- |
| 63 | 62 1 | 0 |
| Pg | ~62 | Pe |

Pg: Paging enable bit 1=enabled, 0 = disabled

PE: Protected Mode enable: 1 = enabled, 0 = disabled.

## Clock Register (SPR 06)

The clock register controls clock gating to the processor to allow lower power consumption. Gating is controlled with a bit pattern which is fed to a clock enable gate. The pattern is 50 bits long, allowed clock control (or power control) in 2% increments. For example loading the register with h2AAAAAAAAAAAA will cause every other clock to be gated off, reducing the effective operating frequency of the core in half. Loading the register with a zero will stop the clock completely. However, a non-maskable interrupt or reset will reload the clock register with all ones, causing the processor to operate at maximum frequency.

|  |  |
| --- | --- |
| 63 50 | 49 0 |
| ~14 | clock gating pattern49..0 |

## Fault PC (SPR 08) and Fault CS (SPR 09)

This pair of registers contains the address at which a fault occurred.

|  |  |
| --- | --- |
| 63 60 | 59 0 |
| ~4 | Program Counter59..0 |

|  |  |
| --- | --- |
| 63 24 | 23 0 |
| ~40 | Code Selector23..0 |

## SRAND1 (SPR16) and SRAND2 (SPR17)

This pair of registers contains the seed for the random number generator. They should be set to a non-zero value to generate random numbers. Using the same seeds results in the same sequence of random numbers generated.

## RAND (SPR 18)

This register contains a random value. The value is the latest value generated by a ‘GRAN’ (generate random number) instruction.

# 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 implement both. Several commercial processors implement both segmentation and paging. Although segmentation has fallen out of favor somewhat it is still used. Typically the segmentation part of the cpu has a handful of segment registers loaded with a flat memory model, then is for the most part ignored.

### Segmentation Overview

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. Table888mmu includes a set of sixteen segment registers.

### Address Formation:

The virtual address is added to a segment base register in order to form a final address. Note that there is no shift associated with the segment addition in this case.

|  |
| --- |
| Virtual Address |
| + |
| segment base register |
| = |
| Segmented address |

### Paging Overview

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

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

# Segmentation

## 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. Table888 uses an array of sixteen segment registers. Segments registers are typically initialized to a flat memory model then forgotten about.

A number of segment registers are dedicated to specific uses and take the place of dedicated registers found in other architectures.

## Segment Usage Conventions

Segment register #15 is the code segment (CS) register. All program counter addresses are formed with the code segment register unless the upper 48 bits of the address are zero in which case the code segment is ignored. If the program counter is used in a load / store instruction, then the code segment register is used as the base by default.

Segment register #14 is the stack segment (SS) register by convention. The stack segment is used for operations involving the stack, PUSH, POP, JSR, or RTS operations. If the base pointer or stack pointer registers are specified in a load / store instruction then the default is to use the stack segment as a base.

Segment register #1 is the data segment (DS) by convention.

Segment register #13 is the volatile data segment (VDS). Addresses formed using this segment register bypass the data cache.

Segment register #12 is task state segment register. This segment register points the task state save area. If the task register is used by a load / store instruction then the task state segment is used by default for address formation. It’s 80x86 equivalent is the task register.

Segment register #11 is Table888’s equivalent to the LDT register in the 80x86 series processors.

#### Segment Registers

|  |  |  |  |
| --- | --- | --- | --- |
| Num |  | Long name | Comment |
| 0 | NS | NULL segment | by convention contains zero |
| 1 | DS | data segment | by convention |
| 2 | TS | thread storage | by convention |
| 3 | BSS | BSS segment | by convention |
| 4 | RS | read only segment | by convention |
| 5 | ES | extra segment | by convention |
| 6 | seg6 |  | unassigned |
| 7 | seg7 |  | unassigned |
| 8 | seg8 |  | unassigned |
| 9 | seg9 |  | unassigned |
| 10 | seg10 |  | unassigned |
| 11 | LDT | local descriptor table | used by hardware to look up local descriptors |
| 12 | TSS | task state segment | associated with tr |
| 13 | VDS | volatile data segment | bypasses the cache |
| 14 | SS | Stack segment | by convention (associated with sp, bp) |
| 15 | CS | Code segment | always used for code addressing |

## Composition

Segment registers are comprised of a program visible portion and an invisible portion. The visible portion of the segment register is a 24 bit selector. The invisible portion is a 128 bit register containing bounds and access rights information. The selector is used to access a segment description entry in the global or local segment descriptor tables. The invisible portion of a segment register is cached in the processor to improve performance.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | |  | Visible portion of segment register -> | | RPL4 | TI | Index19 | |
|  | |  |  |  |  | |  |  |
| ACR8 | | | Limit56..0 | | | | | |
| DPL4 | Base59..0 | | | | | | | |

Selector Format:

|  |  |  |
| --- | --- | --- |
| RPL4 | TI | Index19 |

RPL: request privilege level

TI: table indicator 0=global, 1=local

Index: index into the global or local descriptor table

Whenever the selector value is loaded into a segment register, the entire segment descriptor from the global or local table is loaded into hidden registers. The descriptor controls the accessibility and the location of the segment in memory. The format of a data descriptor is shown below:

### Register Access

Segment register access takes place in parallel with normal general purpose register access.

### Moving Register Values

Segment registers may be only moved to or from general purpose registers. There are no other instructions for manipulating a segment register. MTSEG is used to move a general purpose register to the segment register. MTSEG causes the hidden portion of the segment register to be loaded from memory. MFSEG moves a segment register into a general purpose register. MFSEG does not trigger a memory operation.

## Merge Format:

It would be wasteful and time consuming to store the selector and program counter offset as separate words on the stack when a subroutine is called, when it isn’t needed most of the time. It would take two bus cycles for every subroutine call when really only a single cycle is needed. So usually the selector and offset are merged together into a single 64 bit format. Most of the time a program will consume far less than 40 bits of address space, meaning the upper bits of the program counter are zero. This fact can be used to merge the 24 bit selector into the upper address bits while storing a return address on the stack. When a subroutine return is done, the selector and program counter are split apart. In order to be able to use more than 40 bits for the program counter, a pair of far call, far return instructions could be used to store the selector and offset separately on the stack. Alternately, a bit flag can be stored on the stack to indicate that two words were used to store the selector and offset separately. Table888 uses the alternate mechanism of storing a flag on the stack with the return address.

Return address, long format:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| w1 | Code Segment Selector24 | | ~8 | ~24 | |
| w0 | ~4 | Progam Counter59..40 | Progam Counter39..2 | | 11 |

Return address, short format with selector:

|  |  |  |  |
| --- | --- | --- | --- |
| w0 | Code Segment Selector24 | Progam Counter39..2 | 01 |

Return address, short format without selector:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| w0 | ~4 | Progam Counter59..40 | Progam Counter39..2 | 00 |

Sometimes software is written to manipulate the return address from the stack. Table888’s mechanism for storing the return address makes this difficult to do. Fortunately, this is the rare case, not the usual one.

### Subroutine Calls

Calling a subroutine may cause the selector and current program counter to be merged together and stored on the stack, as mentioned above.

In addition to a subroutine call which does not change the code segment, there is a new prefix instruction provided in order to change the code segment. The format of this prefix and following subroutine call are shown below:

With Selector Prefix:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| RPL4 | TI | Index19 | Addr8 | 618h | JSP |
| Address32 | | | | 518h | JSR address |

The prefix also works with the jump instruction.

When calling a subroutine using indirect addressing, the format of the program counter address stored in memory is assumed to be a merged format.

## Call Gates

Call gates are used to switch to a higher level of privilege. Call gates in Table888 do not touch the stack. No parameters are copied between stacks. Parameters are passed through the gate entirely in registers. It is left up to software to switch stacks if required. Stack switching can be done relatively fast by software using existing instructions as the following example shows.

|  |
| --- |
| SomeCallgateFn:  lw r251,[sp] ; get return address  ; First, save the current stack pointer and segment  shr r250,r251,#61 ; get originating privilege level  ; The stack pointer will be invalid for a little bit, so disable interrupts  sei  sw sp,TCB\_SP0Save[tr+r250\*8] ; save original stack pointer  mfseg sp,ss  sw sp,TCB\_SS0Save[tr+r250\*8]  ; 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\_SS0Save[tr+r250\*8]  mtseg ss,sp  lw sp,TCB\_SP0Save[tr+r250\*8] ; get stack pointer according to privilege level  cli  push r251  ;...  ;now use the stack  ;...  ; Save the current stack back, if needed (may not be necessary)  mfseg r250,cs ; get current privilege level  shr r250,r250,#61  sw sp,TCB\_SP0Save[tr+r250\*8] ; store the stack pointer in the tss.  mfseg r249,ss  sw r249,TCB\_SS0Save[tr+r250\*8] ; store stack segment in tss  ; load back the original stack  pop r251  shr r250,r251,#61 ; get originating privilege level  sei  lw sp,TCB\_SS0Save[tr+r250\*8] ; get back segment register  mtseg ss,sp  lw sp,TCB\_SP0Save[tr+r250\*8] ; get appropriate stack pointer from tss  cli  rts |

Call gates have fallen into disuse in the 80x86 processors, in favor of sysenter/sysexit instructions which execute much faster.

Call Gate Descriptor

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| P | ~ | ~ | 1 | X | C | R | A | ~27 | ~5 | Selector23..0 |
| DPL4 | | | | Offset59..0 | | | | | | |

## Task Gates

Where are the task gates ? Task gates on the 80x86 series have proven to be too much of a solution and too inflexible for practical usage. There are no operating systems that make use of them. I tried for a while a number of years ago to implement a simple task switcher using task gates. I always found myself wanting to ‘wedge into’ the task switch mechanism. Hopefully learning from the experience of others, Table888 does not implement task gates.

Table888 provides a fast way to save groups of registers via the store multiple registers and load multiple registers instructions. These instructions are components that could support an operating system.

### System Calls

System calls in Table888 made use of the BRK instruction. The BRK instruction jumped into an interrupt table based on a vector supplied in the instruction. In the memory managed version of Table888 the BRK instruction executes an interrupt or trap gate specified in the interrupt descriptor table. From the calling program the instruction looks the same, but behind the scenes it executes differently. The format of the BRK instruction remains the same:

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

The vector number is used as an index into the interrupt descriptor table, which identifies an interrupt or trap gate. The entire program counter is set to the value found in the gate. The BRK instruction stores both the program counter and status register on the stack in a long address format:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| w1 | Code Segment Selector24 | | ~8 | Status Reg32 | |
| w0 | ~4 | Progam Counter59..40 | Progam Counter39..2 | | 11 |

A trap or interrupt gate in the interrupt descriptor table only allows for a 28 bit program offset. This limits interrupt or trap routines to residing within the first 256 MB region of the selected address space. This limitation can be overcome by specifying a selector that uses a call gate. The issue here is minimizing the size of the interrupt descriptor table. It consumes cache space when interrupt descriptors are loaded into the cache. So the table is setup for the common case where it is likely that less than 28 bits would be required for the address.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| P | ~ | ~ | 0 | 0 | 1 | 1 | T | Interrupt Code Segment Selector24 |
| DPL4 | | | | Offset28 | | | | |

T = 1: Trap Gate

### 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. Table888seg stores the return address on the stack in one of three different formats. The subroutine call instructions decide which format is needed. The goal here is to minimize stack memory usage in the usual case and maximize performance. Usually one is calling a local routine with a program counter less than 40 bits in size. In that case there is no need to store the selector. Loading the selector would cause the code segment register to be reloaded, incurring a penalty of several additional clock cycles. This is too costly to do for every return operation, especially when it’s not needed. Hence a return address format without a selector is needed. This type of thing is handled with multiple call and return instructions in the 80x86 processors. While it’s possible to have multiple call and return instructions that specify ‘near’ and ‘far’ addresses the author finds that undesirable as it complicates the whole process from writing software to building it. One of the reasons why the ‘flat’ memory model gets chosen is that it avoids the complication of multiple instructions to perform calls and returns. Programmers just aren’t fond of it; it’s too complex a solution from a software perspective. Table888 uses a single return instruction that can recognize different formats of return addresses on the stack. There are two short formats, and one long one. One of the short formats stores only the program counter, the other short format stores the program counter with the selector in the same stack word. The long format stores the program counter and selector using separate words on the stack. The formats are distinguished between by the low order two significant bits of word zero stored on the stack.

Return address, long format:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| w1 | Code Segment Selector24 | | ~8 | ~24 | |
| w0 | ~4 | Progam Counter59..40 | Progam Counter39..2 | | 11 |

Return address, short format with selector:

|  |  |  |  |
| --- | --- | --- | --- |
| w0 | Code Segment Selector24 | Progam Counter39..2 | 01 |

Return address, short format without selector:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| w0 | ~4 | Progam Counter59..40 | Progam Counter39..2 | 00 |

The astute reader will note that this mechanism trashes the two low order bits of the return address. This isn’t a problem. In table888 the two low order bits are the same as the next two bits. These can be reset by the return instruction.

Also note that interrupts and traps always use the long address format because they are required to store the status register on the stack as well as a full 60 bit program counter.

The problem with this approach is that the stack frame isn’t predictable. A call instruction may have stored the return address in the long format or short format. This adds some complexity to accessing arguments on the stack for the called procedure. Subroutine arguments are often accessed relative to the base pointer register, which is setup as part of a function prologue. This prologue code will have to adjust the base pointer. However, that is really only required if the long address format is actually used. The author expects that it wouldn’t be used for some time. One would have to be using code addressing of greater than 40 bits before the long address format of the return address is relevant.

### Returning from Interrupt Routines

Table888seg’s instruction for returning from an interrupt remains the same as for Table888. From a software perspective the instruction hasn’t changed. However from a hardware perspective it has. The RTI instruction now loads the code segment from the stack in addition to the status register and program counter.

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 |

## Hardware Interrupts

### Interrupt Descriptor Table Entries

The interrupt descriptor table contains only two types of gates (interrupt, or trap). Interrupt descriptor table entries are eight bytes in size with the following format:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| P | ~ | ~ | 0 | 0 | 1 | 1 | T | Interrupt Code Segment Selector24 |
| DPL4 | | | | Offset28 | | | | |

T = 1: Trap Gate

T = 0: Interrupt Gate

The difference between a trap and an interrupt is that an interrupt automatically masks further interrupts from happening. A trap does not affect the interrupt flag.

Note that the interrupt routine must be located within the first 256MB region of the segment because only a 28 bit offset is allowed for.

The return from interrupt (RTI) instruction uses a long format to store both the program counter and status register on the stack. This format includes the code segment selector and is as follows:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| w1 | Code Segment Selector24 | | ~8 | Status Reg32 | |
| w0 | ~4 | Progam Counter59..40 | Progam Counter39..2 | | 11 |

# Paging

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

The paging system is capable of mapping the entire 64 bit address space. A multi-level system of page directories and subdirectories is used. In most cases the address space mapped will be less than a full 64 bit address space. The paging system accommodates this by using a smaller directory hierarchy. For instance, if an application is less than 1GB in size, a two level page system is used. If the application can fit within 2MB only a single level is required. The depth of the directory system is controllable on an application basis. The page memory management unit takes care of walking the page tables in hardware in order to find a translation. Translations are stored in a translation look-aside buffer (TLB) which is a translation cache, so that the page tables don’t have to be walked for every translation.



## Registers:

The primary register that controls paging is the page table address register (PTA) or as it is alternately called control register number three. (CR3). This register contains the base address of the root page table in memory. Once the PTA (page table address) is set, the processor knows where to begin looking up virtual to physical address translations.

### Moving Register Values

Control registers for paging may be only moved to or from general purpose registers. There are no other instructions for manipulating a control register. MTSPR is used to move a general purpose register to the control register. MFSPR moves a control register into a general purpose register.

## Page Tables

Page tables are the central piece of a paging system. Page tables are 4kB in size and contain 512, 8 byte entries. The page table format is the same for both page directories, and page mapping tables.

## The Page Table Address Register

This register contains the base address of the page table in memory. The page table must be aligned on a 4kB boundary. The lowest three bits of the register identify the translation space.

|  |  |  |
| --- | --- | --- |
| Address63..12 | ~9 | TS3 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| TS3 | Translation Space | Address Processed | | | | | |
| 0 | 2MB |  | | | | | Address20..12 |
| 1 | 1GB |  | | | | Address29..12 | |
| 2 | 512 GB |  | | | Address38..12 | | |
| 3 | 256 TB |  | | Address47..12 | | | |
| 4 | 128 XB |  | Address56..12 | | | | |
| 5 | 2^64 | Address63..12 | | | | | |
| 6 | not used |  | | | | | |
| 7 | not used |  | | | | | |

Page Directory Entries (PDE’s)

The page directory entries identify where in memory further subdirectories or translation pages are.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Address63..12 | ~2 | a1 | d1 | p4 | c1 | r1 | x1 | w1 |

Page Table Entries (PTE’s)

Page table entries map the virtual address to a physical one.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Address63..12 | ~2 | a1 | d1 | p4 | c1 | r1 | x1 | w1 |

a1: accessed

d1: dirty

p4: privilege level

c1: cacheable

r1: readable

x1: executable

w1: writeable

## The Stack

### Stack Overflow

Stack overflow may occur when a program needs to push more data onto the stack than is available. It’s not necessarily a program error. Table888 checks for stack overflow just before the stack would overflow, to allow the fault handler to be able to re-execute a pending stack operation. Four words are allowed for the fault handler’s stack space, in order to store the return program counter and status register.

### Stack Underflow

Stack underflow is almost always the result of a program error. Stack underflow occurs when there are more stack pop operations than there are pushes. An example would be removing too many parameters from the stack after a function call. Since it’s a software error about the only thing that can be done is to abort the current program. In this case the stack doesn’t need to expand, and the fault handler doesn’t require more stack space.

## Exception Handling

# 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

RESET causes the segment registers to be loaded with a flat memory model. The first fifteen registers are setup as data segments, segment register number fifteen is setup as a code segment. All are set to the highest privilege level. If the segment registers weren’t initialized it would be necessary to provide a bypass function so that the segment registers could be bypassed during startup. The registers have to be initialized or they would contain random values, making it impossible to boot. Note that although the segment registers are setup, the descriptor tables in memory are not. So a segment register can’t be loaded or it would be loaded with an invalid value. That means it’s not possible to jump to another code segment, or load a segment register with the mtseg instruction during initialization.

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.

# Instruction Set Description

A description of changes and additions to the Table888 processor instruction set follows.

### BFCHG – Bitfield Change

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 39 36 35 30 | | 29 24 | 23 16 | 15 8 | 7 0 |
| Func | me | mb | Rt | Ra | Opcode |
| 34 | me6 | mb6 | Rt8 | Ra8 | 03h8 |

#### Description:

Inverts the bitfield in Ra located between the mask begin (mb) and mask end (me) bits and stores the result in the target register.

### BFCLR – Bitfield Clear

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 39 36 35 30 | | 29 24 | 23 16 | 15 8 | 7 0 |
| Func | me | mb | Rt | Ra | Opcode |
| 24 | me6 | mb6 | Rt8 | Ra8 | 03h8 |

#### Description:

Sets the bits to zero of the bitfield in Ra located between the mask begin (mb) and mask end (me) bits and stores the result in the target register.

### BFEXT – Bitfield Extract

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 39 36 35 30 | | 29 24 | 23 16 | 15 8 | 7 0 |
| Func | me | mb | Rt | Ra | Opcode |
| 54 | me6 | mb6 | Rt8 | Ra8 | 03h8 |

#### Description:

Extracts a bitfield from register Ra located between the mask begin (mb) and mask end (me) bits and places the sign extended result into the target register.

### BFEXTU – Bitfield Extract Unsigned

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 39 36 35 30 | | 29 24 | 23 16 | 15 8 | 7 0 |
| Func | me | mb | Rt | Ra | Opcode |
| 44 | me6 | mb6 | Rt8 | Ra8 | 03h8 |

#### Description:

Extracts a bitfield from register Ra located between the mask begin (mb) and mask end (me) bits and places the zero extended result into the target register.

### BFINS – Bitfield Insert

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 39 36 35 30 | | 29 24 | 23 16 | 15 8 | 7 0 |
| Func | me | mb | Rt | Ra | Opcode |
| 04 | me6 | mb6 | Rt8 | Ra8 | 03h8 |

#### Description:

Inserts a bitfield into the target register located between the mask begin (mb) and mask end (me) bits from the low order bits of Ra.

### BFSET – Bitfield Set

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 39 36 35 30 | | 29 24 | 23 16 | 15 8 | 7 0 |
| Func | me | mb | Rt | Ra | Opcode |
| 14 | me6 | mb6 | Rt8 | Ra8 | 03h8 |

#### Description:

Sets the bits to one of the bitfield in Ra located between the mask begin (mb) and mask end (me) bits and stores the result in the target register.

## GRAN – Generate Random Number

GRAN Rt

Instruction Formats:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 14h |  | Rt8 | ~8 | 01h | GRAN Rt |

Operation:

#### Register Form

Rt = random number

Notes:

Execution of the GRAN instruction generates a new random number according to George Marsaglia’s multiply method. The random number seed registers must be set to non-zero values before a number can be generated. The number returned by the GRAN instruction is the previous random number. The newly generated number is available in the RAND special purpose register.

## IMMx – Immediate Prefix

IMM1 #i32

IMM2 #i28

Instruction Formats:

|  |  |  |  |
| --- | --- | --- | --- |
| Sg/Cn4 | Constant28 | FDh | IMM1 |
| Sg4 | Constant28 | 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, when used by itself, sign extends an immediate constant found in a 28 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-four bits using just the IMM1 prefix. The IMM1 prefix also supplies a segment override for the following instruction. However, if used in combination with the IMM2 instruction, the segment override is not provided by the IMM1 instruction.

The IMM2 prefix loads a 28 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.

The IMM2 prefix also supplies a segment register override for the following instruction.

## 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) |

With Selector Prefix:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| RPL4 | TI | Index19 | Addr8 | 618h | JSP |
| Address32 | | | | 508h | JMP address |

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. Transfer to a different code segment at the same privilege level is possible by using the jump selector prefix. The prefix indicates the target selector and adds eight additional address bits.

For indirect addresses, if the top 24 bits are non-zero they are treated as a selector value. If the top 24 bits are zero then the jump stays within the current code segment. It is possible to jump indirectly to a different code segment at the same privilege level.

## 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:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| RPL4 | TI | Index19 | 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 60 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.

## LAR – Load Access Rights

LAR Rt, Ra

Instruction Formats:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 11h |  | Rt8 | Ra8 | 01h | LAR Rt, Ra |

Operation:

#### Register Form

Rt = load\_segment(Ra).access\_rights

Notes:

The access rights to the segment identified by the selector contained in register Ra are loaded to the target register. The access rights are only loaded if the caller has sufficient privilege, otherwise a zero is loaded into the target register.

## LSL – Load Segment Limit

LAR Rt, Ra

Instruction Formats:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 10h |  | Rt8 | Ra8 | 01h | LSL Rt, Ra |

Operation:

#### Register Form

Rt = load\_segment(Ra).access\_rights

Notes:

The segment limit of the segment identified by the selector contained in register Ra are loaded to the target register. The limit is only loaded if the caller has sufficient privilege, otherwise a zero is loaded into the target register.

## 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.

## 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.

# 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} | {bitfield} | 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 |  |  | BMS | BMC | BMF | BMT |  |  |  |  |  |  |  |  |
| C- |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| D- |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| E- |  |  |  |  |  |  |  |  |  |  | NOP |  |  |  |  |  |
| F- |  |  |  |  |  |  |  |  |  |  |  |  |  | IMM1 | IMM2 |  |

# Func Table for RR instructions

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | -0 | -1 | -2 | -3 | -4 | -5 | -6 | -7 | -8 | -9 | -A | -B | -C | -D | -E | -F |
| 0- |  |  |  |  | ADD | SUB | CMP | MUL | DIV | MOD |  |  |  |  |  |  |
| 1- |  |  |  |  | ADDU | SUBU |  | MULU | DIVU | MODU |  |  |  |  |  |  |
| 2- | AND | OR | EOR | ANDN | NAND | NOR | ENOR | ORN |  |  |  |  |  |  |  |  |
| 3- | 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 |  |  |
| 1- | LSL | LAR |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 2- |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 3- | SEI | CLI | PHP | PLP | ICON | ICOFF |  |  |  |  |  |  |  |  |  |  |
| 4- | RTI |  |  |  |  |  |  |  | MTSPR | MFSPR |  |  |  |  |  |  |
| 5- |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 6- |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 7- |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 8- | VERR | VERW | VERX |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 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 |