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

Table of Contents

[Preface 4](#_Toc391604878)

[Who This Book is For 4](#_Toc391604879)

[What this Book is About 4](#_Toc391604880)

[Motivation 4](#_Toc391604881)

[Testing and Debugging 4](#_Toc391604882)

[Register Usage 5](#_Toc391604883)

[Additional Registers 5](#_Toc391604884)

[Page Table Address (SPR 04 or CR3) 5](#_Toc391604885)

[Control Register Zero (SPR 05 or CR0) 6](#_Toc391604886)

[Clock Register (SPR 06) 6](#_Toc391604887)

[Fault PC (SPR 08) and Fault CS (SPR 09) 7](#_Toc391604888)

[SRAND1 (SPR16) and SRAND2 (SPR17) 7](#_Toc391604889)

[RAND (SPR 18) 7](#_Toc391604890)

[Design Choices 7](#_Toc391604891)

[Segmentation and Paging 7](#_Toc391604892)

[Segmentation Overview 7](#_Toc391604893)

[Address Formation: 8](#_Toc391604894)

[Paging Overview 8](#_Toc391604895)

[Segmentation 9](#_Toc391604896)

[Number of Registers: 9](#_Toc391604897)

[Segment Usage Conventions 10](#_Toc391604898)

[Composition 10](#_Toc391604899)

[Register Access 10](#_Toc391604900)

[Moving Register Values 11](#_Toc391604901)

[System Calls 11](#_Toc391604902)

[Returning from Interrupt Routines 11](#_Toc391604903)

[Hardware Interrupts 12](#_Toc391604904)

[Interrupt Vector Table Entries 12](#_Toc391604905)

[Paging 13](#_Toc391604906)

[Registers: 14](#_Toc391604907)

[Moving Register Values 14](#_Toc391604908)

[Page Tables 14](#_Toc391604909)

[The Page Table Address Register 14](#_Toc391604910)

[The Stack 15](#_Toc391604911)

[Stack Overflow 15](#_Toc391604912)

[Stack Underflow 15](#_Toc391604913)

[Exception Handling 15](#_Toc391604914)

[Implementing the Processor 15](#_Toc391604915)

[Convenience Tasks 16](#_Toc391604916)

[next\_state(); 16](#_Toc391604917)

[wb\_xxxx(); 16](#_Toc391604918)

[Implementing Processor Reset 16](#_Toc391604919)

[Instruction Set Description 16](#_Toc391604920)

[BFCHG – Bitfield Change 17](#_Toc391604921)

[BFCLR – Bitfield Clear 17](#_Toc391604922)

[BFEXT – Bitfield Extract 17](#_Toc391604923)

[BFEXTU – Bitfield Extract Unsigned 17](#_Toc391604924)

[BFINS – Bitfield Insert 18](#_Toc391604925)

[BFSET – Bitfield Set 18](#_Toc391604926)

[GRAN – Generate Random Number 19](#_Toc391604927)

[Instruction Formats 19](#_Toc391604928)

[Operation 19](#_Toc391604929)

[Glossary 20](#_Toc391604930)

[FPGA: 20](#_Toc391604931)

[HDL 20](#_Toc391604932)

[Instruction Bundle: 20](#_Toc391604933)

[ISA: 20](#_Toc391604934)

[Program Counter: 20](#_Toc391604935)

[SIMD: 20](#_Toc391604936)

[Stack Pointer 20](#_Toc391604937)

[Resources: 21](#_Toc391604938)

[Reference Material 21](#_Toc391604939)

[Major Opcode Table 23](#_Toc391604940)

[Func Table for RR instructions 24](#_Toc391604941)

[Func Table for R instructions 25](#_Toc391604942)

[01 Func Table 26](#_Toc391604943)

[02 Func Table 27](#_Toc391604944)

# 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 Table888 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.

# Register Usage

|  |  |  |  |
| --- | --- | --- | --- |
| Regno | Name | Usage |  |
| 00 | r0 | This register is always zero, hardware enforced. |  |
| 01 | r1 | first return value / parameter register, by software convention |  |
| 02 | r2 | second return value / parameter register by software convention |  |
| 03-10 |  | compiler temporaries |  |
| 11-18 |  | register variables |  |
| 19-21 |  | register variables for code pointers |  |
| … |  |  |  |
| 244 | fl0 | Flags, this register is used as a flags register by software convention. |  |
| … |  |  |  |
| 250 | LR | link register – software convention |  |
| 251 | CLR | catch exception link register – software convention |  |
| 252 | tr | task register – software convention |  |
| 253 | bp | Base Pointer – this register is reserved for use as a stack frame base pointer by software convention |  |
| 254 | pc | Program Counter - This register reflects the program counter value as is read only, enforced by the hardware. |  |
| 255 | sp | Stack Pointer - This register is used by hardware to implement the stack pointer |  |

# 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 9 6 1 | | | | | | | | 0 |
| Pg | Ce |  |  |  |  |  | BXWR |  | Pe |

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

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

Ce: cache enable 1 = enabled, 0 = disabled

BXWR: triple mode redundancy bits

X: TMR on execute

W: TMR on writes

R: TMR on reads

## 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 0 |
| Program Counter59..0 |

|  |  |  |
| --- | --- | --- |
| 63 9 | 8 4 | 3 0 |
| Code Segment63..9 | ~5 | PL4 |

## 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 uses three 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). This unit is also sometimes called an Address Translation Cache (ATC). A paging system tends to have more overhead associated with it compared to a segmented system.

# 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 three segment registers. Segments registers are typically initialized to a flat memory model then forgotten about.

## Segment Usage Conventions

All program counter addresses are formed with the code segment. If the program counter is referenced in a load / store instruction then the assembler will select the code segment as the base segment register for the instruction.

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 set by the assembler is to use the stack segment as a base.

Load / Store operations use the data segment (DS) by default. This may be overridden using a segment prefix instruction in assembler code.

#### Segment Registers

|  |  |  |  |
| --- | --- | --- | --- |
| Spr Num |  | Long name | Comment |
| 20h | CS | Code segment | always used for code addressing |
| 21h | DS | data segment | by convention |
| 22h | SS | Stack segment | by convention (associated with sp, bp) |

## Composition

The low order four bits of the code segment register contain the processor’s current operating privilege level.

|  |  |  |
| --- | --- | --- |
| 63 9 | 8 4 | 3 0 |
| Code Segment63..9 | ~5 | PL4 |

|  |  |
| --- | --- |
| 63 4 | 3 0 |
| Data / Stack Segment63..4 | PL4 |

### 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. MTSPR is used to move a general purpose register to the segment register. MFSPR moves a segment register into a general purpose register.

Note that the only way to change the code segment is by executing an interrupt. An interrupt or the break instruction will load the code segment from a value in the interrupt vector table. The other instruction which loads the code segment register is the RTI instruction. Unlike some other architectures there are no far call / far return instructions.

### 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 specified in the interrupt vector 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 vector table, which identifies an interrupt or trap. The entire program counter is set to the value found in the table. The code segment register is also set from this table. The BRK instruction stores the program counter, code segment and status register on the stack in a long address format:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| w1 | Code Segment63..9 | ~5 | PL4 | |
| w0 | Progam Counter63..2 | | | I |

### Returning from Interrupt Routines

Table888’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 Vector Table Entries

The interrupt vector table contains two types of vectors (interrupt, or trap). Interrupt vector table entries are sixteen bytes in size with the following format:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Code Segment63..9 | ~5 | PL4 | | |
| Offset63.2 | | | ~ | T |

PL: privilege level

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.

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 and is as follows:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| w1 | Code Segment63..9 | ~5 | PL4 | |
| w0 | Progam Counter63..2 | | | I |

# 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

# Protection Mechanism

The advanced Table888 processor features protection mechanisms implemented with a sixteen level privilege system. Protection of data follows a simple rule. The current privilege level must be greater (numerically lower) or equal to that of the data to be accessed. The current privilege level the processor is operating at is contained in the code segment register. The data privilege level is contained in the page table entry associated with the data. Protection for code follows the rule that code can only be executed at the same or higher (numerically lower) level of privilege.

# 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 code segment register to be loaded with zero. Other than that the segment registers are not initialized. They must be initialized by software before loads and stores will work properly. The registers have to be initialized or they would contain random values, making it impossible to boot.

The processor has to start executing instructions somewhere, so the PC needs to be set at reset. It is set to 40’h000000FFF0.

# 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.

# 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- | SEQ | SNE |  |  |  |  |  |  | SGT | SLE | SGE | SLT | SHI | SLS | SHS | SLO |
| 4- | BEQ | BNE | BVS | BVC | BMI | BPL | BRA | BRN | BGT | BLE | BGE | BLT | BHI | BLS | BHS | BLO |
| 5- | JMP | JSR | JMP (,x) | JSR (,x) | JMP d(Rn) | JSR d(Rn) | BSR |  | BRZ | BRNZ | DBNZ |  |  |  |  |  |
| 6- | RTS |  | LINK |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 7- |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 8- | LB | LBU | LC | LCU | LH | LHU | LW | LWS | LBx | LBUx | LCx | LCUx | LHx | LHUx | LWx | LEAx |
| 9- |  |  | LEA |  |  |  |  |  |  |  |  |  |  |  |  |  |
| A- | SB | SC | SH | SW | CINV | SWS | PUSH | POP | SBx | SCx | SHx | SWx | CINVx |  |  |  |
| B- |  |  |  |  | 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- |  |  |  | SWAP | MOV | NEG | COM | NOT | SXB | SXC | SXH |  |  |  | MOVS |  |
| 1- | LSL | LAR |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 2- |  |  |  |  |  |  |  |  |  |  | CPUID |  |  |  |  |  |
| 3- | SEI | CLI | PHP | PLP | ICON | ICOFF |  |  |  |  |  |  |  |  |  |  |
| 4- | RTI | STP | UNLINK |  |  |  |  |  | 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 |