Gisselquist
Technology, LLC

ZIPCPU SPECIFICATION

Dan Gisselquist, Ph.D.
dgisselq (at) ieee.org

November 5, 2022

Revision History

Rev.	Date	Author	Description
3.0	6/18/2022	Gisselquist	New ZipDMA, debug interface, multi-bus support and bus width independence
2.01	10/19/2019	Gisselquist	Fixed CIS OpCode Table
2.0	1/18/2017	Gisselquist	Switched from 32–bit to 8–bit bytes.
1.1	11/28/2016	Gisselquist	Moved the ZipSystem address to `0xff000000` base.
1.0	11/4/2016	Gisselquist	Major rewrite, includes compiler info
0.91	7/16/2016	Gisselquist	Described three more CC bits
0.9	4/20/2016	Gisselquist	Modified ISA: LDIHI replaced with MPY, MPYU and MPYS replaced with MPYUHI, and MPYSHI respectively. LOCK instruction now permits an intermediate ALU operation.
0.8	1/28/2016	Gisselquist	Reduced complexity early branching
0.7	12/22/2015	Gisselquist	New Instruction Set Architecture
0.6	11/17/2015	Gisselquist	Added graphics to illustrate pipeline discussion.
0.5	9/29/2015	Gisselquist	Added pipelined memory access discussion.
0.4	9/19/2015	Gisselquist	Added DMA controller, improved stall information, and self–assessment info.
0.3	8/22/2015	Gisselquist	First completed draft
0.2	8/19/2015	Gisselquist	Still Draft, more complete
0.1	8/17/2015	Gisselquist	Incomplete First Draft

Contents

Figures

Tables

Preface

Many people have asked me why I am building the ZipCPU. ARM processors are cheap and effective and ASIC processors will always have a better performance than an FPGA soft-core processor. Xilinx makes and markets Microblaze, Altera Nios, and both have better toolsets than the ZipCPU. The RISC–V ISA is now well known among soft-core circles. So, why build a new processor?

The easiest, most obvious answer is the simple one: Because I can.

There's more to it though. There's a lot of things that I would like to do with a processor, and I want to be able to do them in a vendor independent fashion. First and foremost, I would like to be able to both simulate this processor and place it inside an FPGA. Without paying royalties, gate level ARM simulations are out of the question. I would also like to be able to generate Verilog code, both for the processor and the system it sits within, that can run equivalently on both Xilinx, Altera, and Lattice chips, and that can be easily ported from one manufacturer's logic architecture to another. Even more, before purchasing a chip or a board, I would like to know that my soft core works. That means that I'd like to build a test bench to test components with, and Verilator is my chosen simulation tool. This forces me to use all Verilog, and it prevents me from using any proprietary cores. For this reason, ARM, Microblaze and Nios are out of the question.

Why not OpenRISC? Because the ZipCPU has different goals. OpenRISC is designed to be a full featured CPU. The ZipCPU was designed to be a simple, resource friendly, CPU. The result is that it is easy to get a ZipCPU program running on bare hardware for a special purpose application– such as what FPGAs were designed for, although getting a full featured operating system on the ZipCPU remains one of my eventual goals. Further, the OpenRISC ISA is very complex, defining over 200 instructions (even though most of them have has never been fully implemented . . .). The ZipCPU on the other hand has only a small handful of instructions, and all but the Floating Point instructions have already been fully implemented.

My final reason is that I'm building the ZipCPU as a learning experience. The ZipCPU has allowed me to learn a lot about how CPUs work on a very micro level. For the first time, I am beginning to understand many of the Computer Architecture lessons from years ago. Even at its third version, the ZipCPU has continued to teach me more advanced topics such as verification and regression testing.

To summarize: Because I can, because it is open source, because it is light weight, and as an exercise in learning.

Since building the ZipCPU initially in 2015, I've continued to work with it and update it for many of the reasons above. Now, however, I have a couple of new reasons for using the ZipCPU as well. For example, the ZipCPU has now formed the basis for many contracts and as a portion of the deliverables within those contracts. The ZipCPU has also been a focus for learning about Verilog simulation environments, formal verification, and how various bus structures work. For example, starting with version 3.0, the ZipCPU now supports both AXI4 and AXI4-lite interfaces.

Finally, let me say that even though the ZipCPU project has been unfunded, the work that I have placed into it has been paid back nicely. While working on the ZipCPU, I have learned how to build a back end for GCC. I later received a contract for building a special GCC work-around to (post build) "fix" a RISC–V processor that had made it into silicon, but with some instructions not working. Also, the ZipCPU's instruction fetch routines have formed the basis for a scatter gather

DMA implementation, for a scripted SONAR transmitter, and will likely form the basis in the near future for a small (I2C+SPI) telemetry system. These facts now witness to the truth of the Proverb, "In all labour there is profit: but the talk of the lips tendeth only to penury." (Prov 14:23)

Dan Gisselquist, Ph.D.

1. Introduction

The ZipCPU is a soft–core CPU. It has been designed to be small in area, with only a minimal instruction set.

The basic philosophy of the ZipCPU is driven from a desire to minimize logic, while maintaining the ability to be a fully pipelined, 32–bit CPU capable of running a modern operating system (sans MMU). It should run nicely in a small area, while maintaining compatibility with many FPGA architectures.

You might think of the ZipCPU as a poor man's alternative to the larger architectures prevalent today.

2. Key Features

Some of the Key Features of the ZipCPU are listed below:

- 32–bit Architecture: All registers are 32-bits, addresses are 32-bits, instructions are 32-bits wide, etc.
- A RISC CPU. There is no microcode for executing instructions. Where possible, all instructions have been designed to be completed in a single clock cycle.
- A Load/Store architecture. Only load and store instructions can access memory.
- There are no I/O instructions. Attached peripherals are memory mapped, and external to the CPU.
- Big Endian
- A compressed instruction subset provides some amount of instruction compression for the most commonly used instructions.
- A Von-Neumann architecture. Both instructions and data share a common addressing space.
- A pipelined architecture, having stages for **Prefetch**, **Decode**, **Read-Operand**, a combined stage containing the **ALU**, **Memory**, and **Divide** units, and then the final **Write-back** stage. See Fig. 2.1 for a diagram of this structure.
- Completely open source, licensed and released under the GPL v3.
- The ZipCPU has several memory controllers. These support Wishbone, AXI4–lite, and AXI4 bus structures, and may or may not provide a cache of user configurable size depending upon its configuration.
- Bus-width agile. The ZipCPU's memory controllers can handle any bus width of at least 32-bits. Bus width configuration is parameterizable.
- There is an (optional) external interface for debugging the CPU. This allows an external debugger the ability to start, and stop the CPU, read and adjust its registers, or to step through any piece of software one instruction at a time.
- The ZipCPU is also highly configurable, having optional support for clock gating, an external trace port, profiler, multi-tasking, multiple multiply configurations, divide instruction support, early branching, and more.

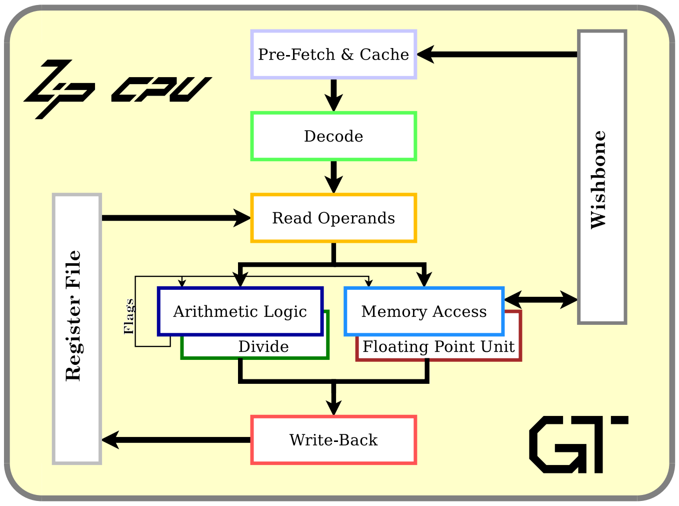

Figure 2.1: ZipCPU internal pipeline architecture

3. CPU Architecture

This chapter describes the general architecture of the ZipCPU. It starts with a description of the ZipCPU's instruction set. From there, it moves on to discuss how the ZipCPU handles interrupts, its pipeline, and then the various memory controller options available to it.

3.1 Instruction Set Architecture

The general form of (most) ZipCPU instructions is shown in Tbl. 3.1. In other words, the ZipCPU will apply some operation, `OP`, to a register plus an immediate, `Rb+#Imm`, and a second register, `Ra`, while leaving the result in the same second register, `Ra`. If the condition `C` is present, the instruction will only complete if the condition holds.

Unconditional ALU operations will set the condition code flags based upon applying the operation to `Rb+#Imm` and `Ra`. Any overflow or carry conditions due to adding `Rb` and the immediate together will be lost.

If the source register, `Rb`, is the program counter register, `PC`, than the immediate will be multiplied by four prior to adding it's value to `PC` to generate the `Rb` value.

The next several sections will go into detail describing the register set, instruction encoding, available operations, condition codes, and more.

3.1.1 Operating Modes

Before introducing the register set, it's important to know that the ZipCPU supports two separate operating modes, a supervisor mode and a user mode. These modes are connected to interrupt handling: when operating in user mode, interrupts are always enabled.[1] When operating in supervisor mode, interrupts are always disabled.

The CPU boots into supervisor mode. The supervisor program can then cause the CPU to switch to user mode by executing a special `RTU` (return to userspace) instruction. When the CPU then encounters either an interrupt or a fault, the CPU will return to supervisor mode *at the instruction where it left off.* This also means that the ZipCPU does not support any interrupt vectors.

[1] Interrupts may still be disabled in the interrupt controller.

```
OP.C   Rb+#Imm,Ra
```

Table 3.1: The form of a generic ZipCPU instruction

Supervisor Register Set #'s 0-15			User Register Set #'s 16-31	
sR0(LR)	sR8		uR0(LR)	uR8
sR1	sR9		uR1	uR9
sR2	sR10		uR2	uR10
sR3	sR11		uR3	uR11
sR4	sR12(FP)		uR4	uR12(FP)
sR5	sSP		uR5	uSP
sR6	sCC		uR6	uCC
sR7	sPC		uR7	uPC
Interrupts Disabled			Interrupts Enabled	

Figure 3.1: ZipCPU Register File

3.1.2 Register Set

The ZipCPU has two sets of sixteen 32-bit registers, one for supervisor mode and the other for user mode. These registers are shown in Fig. 3.1. Any switch from supervisor mode to user mode or back will also cause a sudden shift from one register set to the other. A special form of the `MOV` instruction exists to allow the supervisor access to user registers, but otherwise the two register sets don't interact at all. This effectively means that the compiler knows nothing about the second register set.

Registers in either set may be referenced as `R0` through `R15`. When running in supervisor mode, `MOV` operand references to `uR0` through `uR15` will be understood by the assembler as references to user registers. We'll discuss this further in subsection. 3.1.10. In all other cases, any register reference refers to the currently active set.

Because the register sets are maintained when not in use, switching operating modes has the consequence that the CPU maintains the appearance of continuing where it left off when it last switched modes. Similarly, the supervisor may adjust the user register set to perform a task swap if and when desired.

Two registers in each set are special within the hardware. These are the Program Counter (PC, or R15), and the status register (CC, or R14). When using the compressed instruction set representation, offsets to R13 are optimized within the instruction set to facilitate stack pointer accesses. All other registers are identical in their hardware functionality.

By convention, the tool suite assigns a special meaning to three other registers. As mentioned above, the compiler reserves `R13` for the stack pointer. This also has the mnemonic `SP`. By convention, `R0` is used to maintain the return address of a subroutine, sometimes called the link register or `LR`. Finally, if the compiler requires a frame pointer, then `R12` (or `FP`) is available to it for that purpose.

3.1.3 The Status Register, CC

As mentioned above, the status register (CC) is special. The bit fields within this register have special meaning, and so it really requires its own section. These special bit-fields are shown in Fig. 3.2, and occupy the lower sixteen bits of the status register.

Bit #	Access	Description
31. . . 23	R	Reserved
22. . . 16	R/W	Reserved
15	W	Clear D-Cache command bit, always reads zero
14	W	Clear I-Cache command bit, always reads zero
13	R	Set if processing the first half of a compressed instruction
12	R	(Reserved for) Floating Point Exception
11	R	Division by Zero Exception
10	R	Bus-Error Flag
9	R	Trap Flag (or user interrupt). Cleared on any return to user mode.
8	R	Illegal Instruction Flag
7	R/W	Break–Enable (sCC), or user break (uCC) encountered
6	R/W	Step
5	R/W	User mode / Global Interrupt Enable (GIE) bit
4	R/W	Sleep. When GIE is also set, the CPU waits for an interrupt.
3	R/W	Overflow flag. The last ALU operation produced an arithmetic overflow.
2	R/W	Negative. The sign bit was set as a result of the last ALU instruction.
1	R/W	Carry. The last ALU operation set the carry bit.
0	R/W	Zero. The last ALU operation produced a zero.

Table 3.2: Condition Code Register Bit Assignment

Of the condition codes, the bottom four bits are the current flags from the last ALU instruction that set flags. These are: Zero (Z), Carry (C), Negative (N), and Overflow (V). These flags maintain their usual definition from other CPUs that use them, for all but the shift right instructions. For example, if the result of the last ALU operation is zero, the 'Z' flag will be set. If the result of the last ALU operation sets the most significant bit, then the 'N' flag will be set. Carry is set according to unsigned operation overflow, and the overflow bit is set according signed arithmetic overflow.

Local and arithmetic shift operations also use the carry bit to capture the last bit shifted off the register. This functionality isn't well supported by the compiler, but can be used in assembly to implement extended shift instructions.

We'll walk through the next many bits of the status register in order from least significant to most significant.

4. Bit 4 is the sleep bit. When set, the CPU will enter into a sleep mode. The CPU will wake up and exit from sleep mode on any interrupt if interrupts are enabled.

   This means that this bit can be used to implement two functionalities. There's the `WAIT` for interrupt instruction, whereby the CPU simply sleeps as discussed above in user mode. This bit can also be used to `HALT` the CPU should it ever be set while the CPU is in supervisor mode. (The `WAIT` instruction will place the CPU in user mode, independent of the mode it was in when executed.)

   If the `OPT_CLKGATE` parameter is set, the CPU will also turn off its clock once it finishes entering sleep mode.

5. Bit 5 is a global interrupt enable bit (GIE). When this bit is set, interrupts will be enabled, otherwise they are disabled. When interrupts are disabled, the CPU will be in supervisor mode, otherwise it is in user mode. This bit also forms the fifth bit of any register address, controlling which register set the CPU reads from by default. Thus, to execute a context switch from supervisor mode to user mode, all one needs to do is to set this bit. Then, on a subsequent interrupt or CPU exception, this bit will be automatically cleared and the CPU will return to supervisor mode.

   Logic within the CPU will prevent user mode software from setting the sleep register and clearing the GIE register at the same time, with clearing the GIE register taking precedence. This keeps the user from halting the CPU, thus restricting the halt instruction to supervisor mode only.

   Whenever read, the supervisor CC register will always have this bit cleared, whereas the user CC register will always read this bit set.

6. Bit 6 is a step bit in the user's CC register, and zero in the supervisor's CC register. It can only be read or set from supervisor mode. When set, any switch to user mode will limit processing in user mode to a single instruction before returning to supervisor mode.

   There are two exceptions to this single instruction rule: a usermode program executing a compressed instruction will complete the full two-instruction sequence, and any usermode program executing a `LOCK` instruction will complete an additional three instruction atomic sequence before concluding the step instruction.

   This functionality was added to allow one software program to debug a second program running in user space.

While the CPU can also be stepped in supervisor mode, the supervisor's step bit is not used in that process. Rather, that is accomplished via the CPU debug port.

7. Bit 7 is a break bit.

   In the user register set, this bit is a status bit that will be set if the user mode program encounters a `BREAK` instruction. It will automatically be cleared upon any release from interrupt.

   In the supervisor register set, this becomes the break enable control bit. This bit determines if a usermode `BREAK` instruction should generate an external break (break enabled), or just return processing to the supervisor mode.

   A break in supervisor mode will also cause an "external break", independent of the break enable bit.

   External breaks are handled by the CPU's wrapper. If the CPU is set to `START_HALTED`, such breaks will halt the CPU. Otherwise, they will cause the CPU to reboot.

   This functionality was added to enable a (potential) external debugger to set and manage software breakpoints.

8. Bit 8 is an illegal instruction bit. When the CPU attempts to execute either a non-existent instruction, or an instruction from an address that produced a bus error when read, this bit will get set. If set in user mode, the CPU will switch to supervisor mode. If an illegal instruction is encountered in supervisor mode, the CPU will issue an external break as described by the break handling section above.

9. Bit 9 is a trap bit. This bit is shared between both user and supervisor `CC` registers. It may be set by a usermode process to request a switch to supervisor mode–a soft interrupt if you will. It is then cleared upon any subsequent return to user space.

   This bit allows a supervisor mode process to determine, following any switch to supervisor mode, if the switch was caused by a user request.

10. Bit 10 is a bus error flag. This bit will be set following any bus error return response to either a load or store instruction.

    If such a bus error is encountered in user mode, then this bit will be set in the user's CC register and the CPU will switch to supervisor mode. It will be cleared by any subsequent return to user mode instruction.

    If the bus error is instead encountered in supervisor mode, then this bit will be set in the supervisor's CC register and the CPU will generate an external break.

    Bus errors encountered by the instruction fetch pipeline are returned as illegal instructions, and therefore will not affect this bit.

11. Bit 11 is a division by zero exception flag. This operates in a fashion similar to the bus error flag. If the user attempts to use the divide instruction with a zero denominator, the system will switch to supervisor mode and set this bit in the user CC register. The bit is automatically cleared upon any return to user mode, although it can also be manually cleared by the supervisor. In a similar fashion, if the supervisor attempts to execute a divide by zero, the CPU will issue an external break and set this division by zero exception flag in the supervisor's CC register for the debugger to inspect. This bit will automatically be cleared

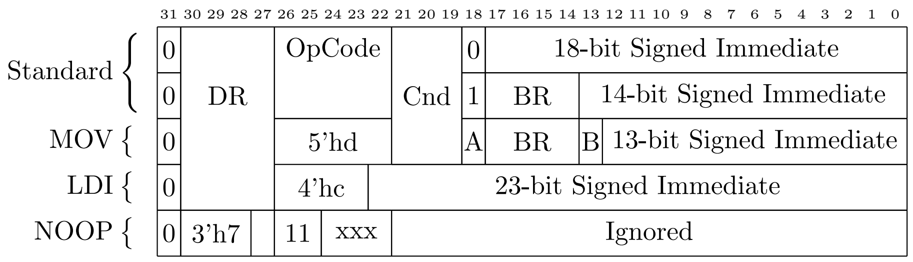

Figure 3.2: Zip Instruction Set Format

upon any CPU reset, or it may be manually cleared by the external debugger writing to this register.

12. Bit 12 is reserved for a hardware accelerated floating point error. This will operate in a similar fashion to both the bus error and the division by zero flags, only it will be set upon a (yet to be determined) floating point error.

13. Bit 13 is the compressed instruction set phase register. This bit will be set on the first instruction of any compressed instruction set pair. It can be used to capture whether or not a CPU fault occurred following the first instruction in a compressed instruction set pair. This is a status bit only.

    The CPU (currently) has no ability to restart an operation in the middle of a compressed instruction set pair.

14. Bit 14 is a clear instruction cache bit. The supervisor may write a one to this bit in order to cause the CPU instruction cache to be cleared. The bit always reads as a zero.

    Writing to this bit from user mode has no effect.

15. Bit 15 is a clear data cache bit. The supervisor may write a one to this bit in order to cause the CPU data cache to be cleared. The bit always reads as a zero.

    Writing to this bit from user space has no effect.

The upper 16-bits of this register a reserved.

3.1.4 Instruction Format

In general, ZipCPU instructions fit in one of the formats shown in Fig. 3.2. The basic format is that some operation, defined by the OpCode, is applied if a condition, Cnd, is true in order to produce a result which is placed in the destination register (DR). The destination register also forms the "A" operand of any instruction. The "B" operand is formed from either an 18–bit signed immediate, or a 14–bit signed immediate plus the value contained within a second register.

There are a couple of exceptions to this general instruction model. The first is the `MOV` instruction, which steals bits 13 and 18 to allow supervisor access to user registers. In supervisor mode, these are set to one to reference user registers, zero otherwise. They are ignored in user mode. The second exception is the load 23–bit signed immediate instruction (`LDI`). This instruction accepts no conditions and uses only a 4-bit opcode. The third exception is the `NOOP` instruction group, encoding the `BREAK`, `LOCK`, `SIM`, and `NOOP` instructions. These instructions ignore their register and immediate settings. Further, the immediate bits used by these opcodes are available for simulation or debug facilities, but otherwise ignored by the CPU. Finally, there is an (optional) compressed instruction format that we'll cover later.

3.1.5 Instruction OpCodes

32 possible instructions can be generated from a 5–bit opcode field. Tbl. 3.3. shows how these 32–values have been allocated to implement 29 instructions. An additional six instruction opcodes are reserved for a (potential, future, optional) single precision floating point accelerator.

3.1.6 Conditional Instructions

Most, although not quite all, instructions may be conditionally executed. The 23–bit load immediate instruction, together with the special instructions, `NOOP`, `SIM`, `BREAK`, and `LOCK`, are the exceptions to this rule. All other instructions may be conditionally executed.

From the four condition code flags, eight conditions are defined, as shown in Tbl. 3.4. There are no condition codes for either less than or equal, or for greater than, whether signed or unsigned. In a similar fashion, there is no condition code for not V. Ways of handling non–supported conditions are discussed in Sec. 3.1.7.

With the exception of `CMP` and `TST` instructions, conditionally executed instructions will not further adjust the condition codes. This allows conditional instruction sequences to be strung together. Conditional `CMP` or `TST` instructions will adjust conditions whenever they are executed. In this way, multiple conditions may be evaluated without branches, creating a sort of logical and–but only if all the conditions are the same. For example, to do something if `R0` is one and `R1` is two, one might implement the assembly shown in Tbl. 3.5. This ability to generate double conditions is used heavily by the compiler when comparing 64-bit numbers together.

The real utility of conditionally executed instructions is that, unlike conditional branches, conditionally executed instructions will not clear the pipeline if they are not executed.

3.1.7 Modifying Conditions

A quick look at the list of conditions supported by the ZipCPU and listed in Tbl. 3.4 reveals that the ZipCPU does not have a full set of conditions. Tbl. 3.6, therefore, shows examples of how these unsupported conditions can be created simply by adjusting the compare instruction, for no extra cost in clocks. Care needs to be taken to ensure that adding one to any immediates, as shown above, does not overflow the size of the immediate field.

Many of these alternate conditions are chosen automatically by the ZipCPU compiler.

3.1.8 Operand B

Many instruction forms have a 19-bit source "Operand B", or OpB for short, associated with them. This "Operand B" is shown in Fig. 3.2 as part of the standard instruction format. For all but the `MOV` instruction, an Operand B is either equal to a register plus a 14–bit signed immediate offset, or

OpCode		A-Reg	Instruction	Sets CC
5'h00	`SUB`	Subtract		Y
5'h01	`AND`	Bitwise And		
5'h02	`ADD`	Add two numbers		
5'h03	`OR`	Bitwise Or		
5'h04	`XOR`	Bitwise Exclusive Or		
5'h05	`LSR`	Logical Shift Right		
5'h06	`LSL`	Logical Shift Left		
5'h07	`ASR`	Arithmetic Shift Right		
5'h08	`BREV`	Bit Reverse B operand into result		N
5'h09	`LDILO`	Load Immediate Low		
5'h0a	`MPYUHI`	Upper 32 of 64 bits from an unsigned 32x32 multiply		Y
5'h0b	`MPYSHI`	Upper 32 of 64 bits from a signed 32x32 multiply		
5'h0c	`MPY`	Lower 32 of 64 bits from a 32x32 bit multiply		
5'h0d	`MOV`	Move OpB into Ra		N
5'h0e	`DIVU`	R0-R13	Divide, unsigned	Y
5'h0f	`DIVS`	R0-R13	Divide, signed	
5'h10	`CMP`	Compare (Ra-OpB) to zero		Y
5'h11	`TST`	Test (AND w/o setting result)		
5'h12	`LW`	Load a 32-bit word from memory (OpB) into Ra		N
5'h13	`SW`	Store a 32-bit word from Ra into memory at (OpB)		
5'h14	`LH`	Load 16-bits from memory (opB) into Ra, clear upper 16 bits		
5'h15	`SH`	Store the lower 16-bits of Ra into memory at (OpB)		
5'h16	`LB`	Load 8-bits from memory (OpB) into Ra, clear upper 24 bits		
5'h17	`SB`	Store the lower 8-bits of Ra into memory at (OpB)		
5'h18/9	`LDI`	Load 23–bit signed immediate		N
5'h1a	`FPADD`	R0-R13	(Reserved for) Floating point add	Y
5'h1b	`FPSUB`	R0-R13	(Reserved for) Floating point subtract	
5'h1c	`FPMPY`	R0-R13	(Reserved for) Floating point multiply	
5'h1d	`FPDIV`	R0-R13	(Reserved for) Floating point divide	
5'h1e	`FPI2F`	R0-R13	(Reserved for) Convert integer to floating point	
5'h1f	`FPF2I`	R0-R13	(Reserved for) Convert floating point to integer	
5'h1c	`BREAK`	None(15)	Debugger break point	N
5'h1d	`LOCK`	None(15)	Begin an atomic access sequence	
5'h1e	`SIM`	None(15)	Simulation–only instruction	
5'h1f	`NOOP`	None(15)		

Table 3.3: ZipCPU OpCodes

Code	Mnemonic	Condition
3'h0	None	Always execute the instruction
3'h1	`.Z`	Zero. Only execute when 'Z' is set
3'h2	`.LT`	Less than. Only execute when 'N' is set
3'h3	`.C`	Carry set (Also known as less-than unsigned)
3'h4	`.V`	Overflow. Only execute when 'V' is set
3'h5	`.NZ`	Not zero. Only execute when 'Z' is clear
3'h6	`.GE`	Greater than or equal. Executes when 'N' is clear
3'h7	`.NC`	Not carry (also known as greater-than or equal, unsigned)

Table 3.4: Conditions for conditional operand execution

```
CMP 1,R0
; Condition codes are now set based upon R0-1
CMP.Z 2,R1
; If R0 ≠ 1, conditions are unchanged, Z is still false.
; If R0 = 1, conditions are now set based upon R1-2.
; Now some instruction could be done based upon the conjunction
; of both of these conditions.
; While we use the example of a SW, it could easily be any other instruction.
SW.Z R0,(R2)
```

Table 3.5: An example of a double conditional

Unsupported condition	Modified	Name

`CMP Imm,Ry`
`BLE label`
	`CMP 1+Imm,Ry`
	`BLT label`
		Less-than or equal (signed, `Z` or `N` set)

`CMP Rx,Ry`
`BLE label`
	`CMP Rx,Ry`
	`BLT label`
	`BZ label`
		Less-than or equal (signed, `Z` or `N` set)

`CMP Imm,Ry`
`BGT label`
	`CMP 1+Imm,Ry`
	`BGE label`
		Greater-than (immediate)

`CMP Rx,Ry`
`BGT label`
	`CMP Ry,Rx`
	`BLT label`
		Greater-than (register)

`CMP Imm,Ry`
`BLEU label`
	`CMP 1+Imm,Ry`
	`BC label`
		Less-than or equal, unsigned immediate

`CMP Rx,Ry`
`BLEU label`
	`CMP Ry,Rx`
	`BNC label`
		Less-than or equal unsigned register

`CMP Imm,Ry`
`BGTU label`
	`CMP 1+Imm,Ry`
	`BNC label`
		Greater-than unsigned (immediate)

`CMP Rx,Ry`
`BGTU label`
	`CMP Ry,Rx`
	`BC label`
		Greater-than unsigned

Table 3.6: Modifying conditions

18	17 16 15 14	13 12 11 10 9 8 7 6 5 4 3 2 1 0
0	18-bit Signed Immediate	
1	Reg	14-bit Signed Immediate

Table 3.7: Bit allocation for Operand B

an 18–bit signed immediate offset by itself. This value is encoded as shown in Tbl. 3.7. This format represents a deviation from many other RISC architectures that use `R0` to represent zero, such as OpenRISC and RISC-V. Here, instead, we use a bit within the instruction to note whether or not an immediate is used. The result is that ZipCPU instructions can encode larger immediates within their instruction space.

In those cases where a fourteen or eighteen bit immediate doesn't make sense, such as for `LDILO`, the extra bits associated with the immediate are simply ignored. (This rule does not apply to the shift instructions, `ASR`, `LSR`, and `LSL`–which all use all of their immediate bits.)

3.1.9 Address Modes

Load and store instructions use the OpB field for their address, whether source or destination. As a result, the ZipCPU can support both register plus immediate addressing, as well as a limited amount of immediate addressing.

3.1.10 Move Operands

The `MOV` instruction is the exception to operand B encoding, with the purpose of providing the supervisor access to user mode registers while in supervisor mode. The two bits, shown as `A` and `B` in Fig. 3.2 above, are designed to contain the high order bit of the 5–bit register index. If the `B` bit is a '1', the source operand comes from the user register set. If the `A` bit is a '1', the destination operand is in the user register set. A zero bit indicates the current register set.

This encoding has been chosen to keep the compiler simple. For the most part, the extra bits are quietly set to zero. Special assembly instructions, or particular compiler built–in instructions, can be used to get access to these cross register set move instructions.

Further, the `MOV` instruction lacks the full OpB capability to use a register or a register plus immediate as a source, since a load immediate instruction, `LDI`, already exists. As a result, all moves come from a register plus a potential offset.

This also creates a situation where a `MOV` instruction may be used like a 3-operand `ADD` instruction. Because the `MOV` instruction doesn't affect the condition codes, the compiler may use this instruction during address calculation.

3.1.11 Multiply Operations

The ZipCPU supports three separate 32x32-bit multiply instructions: `MPY`, `MPYUHI`, and `MPYSHI`. The first of these produces the low 32-bits of a 32x32-bit multiply result. The second two produce the upper 32-bits. `MPYUHI` produces the upper 32-bits assuming the multiply was unsigned, whereas

OPT_MULTIPLY	Implementation
0	No multiply support
1	Single clock multiply. This implementation neither registers the inputs to the multiplier, nor the output prior to the ALU output result.
2	Two clock multiply. This implementation registers the inputs to the multiply unit, but the outputs are not registered prior to the ALU mux.
3	Three clock multiply. This implementation registers the inputs to the multiply unit as well as the outputs immediately following the multiply and prior to the final ALU register. This is the workhorse of most FPGA DSP implementations, although it does require a DSP or DSP combination capable of implementing a 32x32 multiply in a single clock cycle.
4	Four clock multiply. This implementation splits the multiply into multiple sixteen bit multiplies in a FOIL (first, outer, inner, last) binomial multiplication fashion. The first clock registers the inputs. The second clock calculates all FOIL values, and the third clock adds the results together to form a full 64-bit multiply result. This method is appropriate for DSP implementations that can handle 16x16 bit multiplies but cannot be combined to implement a 32x32 bit multiply.
5+	Slow multiply. This implementation uses (roughly) 32-cycles to achieve a full multiply result. This is the one implementation that does not use hardware multiply (DSP) support.

Table 3.8: Multiply implementation choices

`MPYSHI` produces the same bits for signed multiplication. Each multiply instruction is independent of every other in execution, although the compiler is likely to use them as though they were dependent.

In an effort to maintain a fast clock speed, all three of these multiplies have been slowed down in logic. Thus, depending upon the setting of the `OPT_MULTIPLY` parameter, the multiply instructions will either 1) cause an ILLEGAL instruction error (`OPT_MULTIPLY=0`, or no multiply support), or take `OPT_MULTIPLY` additional clock cycles to complete.

Several multiplication implementations exist, as shown in Tbl. 3.8.

3.1.12 Divide Unit

The ZipCPU also has an optional divide unit which can be built alongside the ALU. This divide unit provides the ZipCPU with another two instructions that cannot be executed in a single cycle: `DIVS`, or signed divide, and `DIVU`, the unsigned divide. These are both 32–bit divide instructions, dividing one 32–bit number by another. In this case, the Operand B field, whether it be register or register plus immediate, constitutes the denominator, whereas the numerator is given by the other register.

As with the multiply, the divide instructions are also multi–clock instructions. While the divide is running, the ALU, any memory loads, and the floating point unit (if installed) will all be idle. Once the divide completes, other units may be activated.

Should the divisor be zero, the divide will result in a division by zero exception. Upon exception, the divide by zero bit will be set in the appropriate CC register. In the case of a user mode divide by

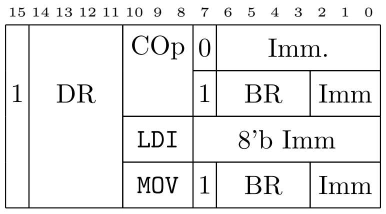

Figure 3.3: ZipCPU Compressed Instruction Set (CIS) Format

COp		Instruction
3'h0	`SUB`	Subtract
3'h1	`AND`	Bitwise And
3'h2	`ADD`	Add two numbers
3'h3	`CMP`	Compare
3'h4	`LW`	Load 32-bit word
3'h5	`SW`	Store 32-bit word
3'h6	`LDI`	Load immediate
3'h7	`MOV`	Move

Table 3.9: CIS OpCodes

zero, this will be cleared by any return to user mode command. The supervisor bit may be cleared either by either a reboot or by a write from the external debugger.

3.1.13 Compressed Instructions

The ZipCPU also supports a compressed instruction set (CIS), as outlined in Fig. 3.3, when enabled via `OPT_CIS`. This compressed instruction set packs two instructions per word. Words must still be aligned, and jumping into the middle of a compressed instruction is not (currently) allowed. Interrupts, therefore, are disabled between the two instructions. Further, the CIS only permits the encoding of 8 of the 32 opcodes available in the ISA. These eight compressed opcodes are listed in Tbl. 3.9.

A final feature of the compressed instruction set has to do with load and store instructions. All CIS load and store instructions use the form `Rb+#Imm`. The instruction encoding that would otherwise be for an `#Imm` alone has been made into a shorthand for using the stack pointer as `Rb` with an offset. Hence the compressed instruction set allows loads and stores to offsets of the Stack Pointer of -128 octets on up to 127 octets. In practice, this gives the compressed load and store instructions, when referencing the stack, thirty–two words that they can reference.

This compressed instruction set is somewhat similar to other architectures that have a thumb instruction set, with the difference that the ZipCPU can intermix regular and compressed instructions at will. When using the CIS, instructions are still issued one at a time, however interrupts are disabled between instruction halves in order to prevent the CPU from stopping and then needing to

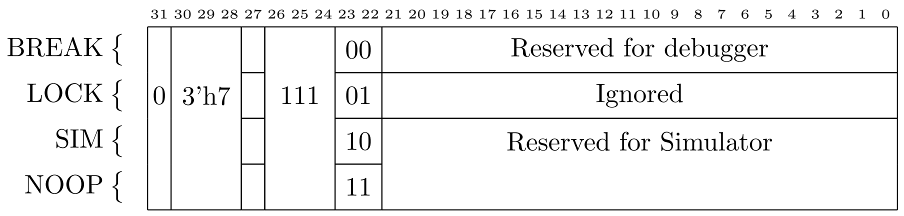

Figure 3.4: NOOP/Break/LOCK Instruction Format

re-start mid-instruction. Further, it is the silent job of the assembler to generate CIS instructions in an opportunistic fashion–unless this feature has been disabled on the command line.

The disassembler represents CIS instructions by placing a vertical bar between the two components, while still leaving them on the same line.

Compressed instructions do not support conditional execution.

3.1.14 BREAK, Bus LOCK, SIM, and NOOP Instructions

Four instructions within the opcode list in Tbl. 3.3, have been reserved for special operations. These are the `BREAK`, bus `LOCK`, `SIM`, and `NOOP` instructions. These are encoded according to Fig. 3.4.

The `BREAK` instruction is useful for creating a debug instruction that will halt the CPU without executing. If in user mode, depending upon the setting of the break enable bit, it will either switch to supervisor mode or halt the CPU–depending upon where the user wishes to do his debugging. The lower 22 bits of this instruction are reserved for the debugger's use.

The `LOCK` instruction forms the basis of the ZipCPU's atomic operation support. The `LOCK` instruction is the first instruction of a four instruction sequence that executes with interrupts disabled. This sequence is typically characterized by a `LOCK` instruction, followed by a load instruction, an ALU operation, and then a store instruction. Using this four instruction sequence, the ZipCPU can perfom atomic ALU operations such as adds, subtracts, bit-wise OR, bit-wise AND, and exclusive OR operations. It can also be used to implement atomic exchanges, test and set instructions, or compare and swap instructions. Since interrupts are disabled during `LOCK` instructions, the sequence can also be used for a short series of instructions in user mode that need to execute with interrupts disabled.

The `SIM` and `NOOP` instructions need a touch more explaining. These instructions have one meaning when run in simulation, and a separate meaning when run in hardware. From the CPU's standpoint, the `SIM` instruction is designed to be an illegal instruction in hardware (i.e. when the `OPT_SIM` parameter is clear), and a `NOOP` instruction when executed in simulation. When executed in hardware, the lower 22–bits of these instructions are ignored.

In simulation, however, those lower 22–bits often have a meaning specifying a simulation only instructions.

Both `SIM` and `NOOP` instructions, though, contain 22–bits that can be used by a simulator if present. The encoding of these 22-bits is identical, so that programs that run in a simulator may run on actual hardware as well (using the `NOOP` encoding), or they may complain that they were unintended to run on actual hardware, such as if the `SIM` encoding were used. Particular encodings

	31	30	29	28	27	26	25	24	23	22	21 20	19 18 17 16 15 14 13 12 11 10 9 8	7 6 5 4	3 2 1 0
xEXIT	0	1	1	1		1	1	1	1	S		12'h01	Rsrvd	
xDUMP	0	1	1	1		1	1	1	1	S		12'h02	8'hff	
xDUMP Rx	0	1	1	1		1	1	1	1	S		12'h002	0	Reg
xDUMP uRx	0	1	1	1		1	1	1	1	S		12'h02	1	uReg
xOUT Rx	0	1	1	1		1	1	1	1	S		12'h02	2	Reg
xOUT uRx	0	1	1	1		1	1	1	1	S		12'h02	3	uReg
xOUT #Imm	0	1	1	1		1	1	1	1	S		12'h004	Imm	

Figure 3.5: NOOP/SIM Sub-Instruction Format

allow for exiting the simulation with a known exit code, x`EXIT`, dumping either one or all registers, x`DUMP`, or simpling sending a character to the simulator's standard output stream, x`OUT`–where x is either `N` for the `NOOP` version of the instruction, or `S` for the `SIM` version of the opcode.

The various `NOOP` and `SIM` encodings are listed in Fig. 3.5.

3.1.15 Floating Point

Although the ZipCPU does not (yet) have a floating point unit, the current instruction set reserves six opcodes for floating point operations. It also reserves a bit in the CC register for treating floating point exceptions like divide by zero errors.

This should allow a 32–bit floating point accelerator to be included within the CPU, and to allow some amount of native support for 32–bit floating point operations. 64–bit floating point instructions will still either need to be emulated in software, or else they will need an external floating point peripheral.

Until this FPU is built and integrated, or even afterwards if the floating point unit is not installed by option, floating point instructions will trigger an illegal instruction exception, which may be trapped and then implemented in software.

3.1.16 Derived Instructions

The ZipCPU supports many other common instructions by construction, although not all of them are single cycle instructions. Tables 3.10, 3.11, 3.12 and 3.13 show how many of these other instructions may be implemented on the ZipCPU. Many of these instructions will have assembly equivalents, such as the branch instructions, to facilitate working with the CPU.

3.2 Interrupt Handling

The ZipCPU does not maintain any interrupt vector tables. If an interrupt takes place, the CPU simply switches to from user to supervisor (interrupt) mode. Since getting to user mode in the first

Mapped	Actual	Notes

`ABS Rx`	`TST -1,Rx`
	`NEG.LT Rx`
		Absolute value instruction. This depends upon the derived `NEG` instruction below, and so this expands into three instructions total.

`ADD Ra,Rx`
`ADDC Rb,Ry`
	`Add Ra,Rx`
	`ADD.C $1,Ry`
	`Add Rb,Ry`
		Add with carry. This capability does not extend easily past 64 bits.

`BRA.`x `+/-$Addr`	`ADD.`x `$Addr+PC,PC`	Branch or jump on condition x. Works for 18–bit signed address offsets.
`BZ $Addr`	`Add.Z $Addr+PC,PC`	Branch on zero. Also known as branch on equals.
`BNZ $Addr`	`Add.NZ $Addr+PC,PC`	Branch on not-zero. Also known as branch on not-equals.
`BLT $Addr`	`Add.LT $Addr+PC,PC`	Branch on less than.
`BGE $Addr`	`Add.GE $Addr+PC,PC`	Branch on greater than or equal to.
`BC $Addr`	`Add.C $Addr+PC,PC`	Branch on carry, also known as branch on less-than unsigned.
`BNC $Addr`	`Add.NC $Addr+PC,PC`	Branch on not carry.
`BV $Addr`	`Add.V $Addr+PC,PC`	Branch on overflow.

`BUSY`
`CLR Rx`
	`ADD $-1,PC`
	`LDI $0,Rx`
		Execute an infinite loop.
		Clears Rx, leaving the flags untouched. This instruction can be compressed, but cannot be conditional.

`CLR.NZ Rx`	`BREV.NZ $0,Rx`	Clears Rx, leaving the flags untouched. This instruction can be executed conditionally. The assembler will quietly choose between `LDI` and `BREV` depending upon the existence of the condition.
`HALT`	`Or $SLEEP,CC`	This only works when issued in interrupt/supervisor mode. In user mode this is simply a wait until interrupt instruction.
`JMP R6+$Offset`	`MOV $Offset(R6),PC`	Only works for 15–bit aligned offsets. Other offsets may require adding the offset first to R6 before jumping.

Table 3.10: Derived Instructions

Mapped	Actual	Notes

`LJMP $Addr`	`LW (PC),PC`
	`Address`
		Although this only works for an unconditional jump, and it only makes sense in an environment with a unified instruction and data address space, this instruction combination makes for a nice combination that can be adjusted by a linker at a later time.

`LJMP.`x `$Addr`	`LW.`x `4(PC),PC`
	`ADD 4,PC`
	`Address`
		Implements a conditional long jump.

`LJSR $Addr`	`MOV $8+PC,R0`
	`LW (PC),PC`
	`Address`
		Long jump-to-subroutine. This is similar to the LJMP instruction, save that it stores the return address in `R0`.

`JSR PC+$Offset`	`MOV $4+PC,R0`
	`ADD $Offset,PC`
		This is similar to the jump and link instructions from other architectures, save only that it requires a specific link instruction, seen here as the `MOV` instruction on the left.

`LDI $val,Rx`	`BREV REV(`*val*`)&0x0ffff,Rx`
	`LDILO (`*val*`&0x0ffff),Rx`
		Since there's not enough instruction space to load a complete immediate value into any register, fully loading a register with a 32-bit value requires two cycles. The `LDILO` (load immediate low) instruction has been created to facilitate this together with `BREV`.
		This is also the appropriate means for setting a register value to an arbitrary 32–bit value in a post–assembly link operation.

`NEG Rx`	`XOR $-1,Rx`
	`ADD $1,Rx`
		Negates Rx

`NEG.C Rx`	`MOV.C $-1+Rx,Rx`
	`XOR.C $-1,Rx`
		Conditionally negates Rx

`NOT Rx`	`XOR $-1,Rx`	One's complement

Table 3.11: Derived Instructions, continued

`POP Rx`	`LW $(SP),Rx`
	`ADD $4,SP`
		The compiler avoids the need for this instruction and the similar `PUSH` instruction when setting up the stack by coalescing all the stack address modifications into a single instruction at the beginning of any stack frame.

`PUSH Rx`	`SUB $4,SP`
	`SW Rx,$(SP)`
		Note that for pipelined operation, it helps to coalesce all the `SUB`'s into one command, and place the `SW`'s right after each other.

`RET`	`MOV R0,PC`	This depends upon the return address either remaining in `R0` from a prior `JSR` instruction, or otherwise it needs to be restored prior to the return call.

`SEXB Rx`	`LSL 24,Rx`
	`ASR 24,Rx`
		Signed extend an 8–bit value into a full word.

`SEXH Rx`	`LSL 16,Rx`
	`ASR 16,Rx`
		Sign extend a 16–bit value into a full word.

`STEP`	`OR $Step\|$GIE,CC`	Steps a user mode process by one instruction

`SUBR Rx,Ry`	`XOR -1,Ry`
	`ADD 1+Rx,Ry`
		Ry is set to Rx-Ry, rather than the normal subtract which sets Ry to Ry-Rx.

`SUB Ra,Rx`
`SUBC Rb,Ry`
	`SUB Ra,Rx`
	`SUB.C $1,Ry`
	`SUB Rb,Ry`
		Subtract with carry. Note that the overflow flag may not be set correctly after this operation.

`TRAP #X`	`LDI $x,R1`
	`AND ~$GIE,CC`
		This works because whenever a user lowers the $GIE flag, it sets a TRAP bit within the uCC register. Therefore, upon entering the supervisor state, the CPU only need check this bit to know that it got there via a TRAP. The trap could be made conditional by making the LDI and the AND conditional. In that case, the assembler would quietly turn the LDI instruction into a `BREV`/`LDILO` pair, but the effect would be the same.

`TS Rx,Ry,(Rz)`	`LDI 1,Rx`
	`LOCK`
	`LB (Rz),Ry`
	`TEST Ry`
	`SB.Z Rx,(Rz)`
		A test and set instruction. The `LOCK` instruction insures that the next three instructions lock the bus between the instructions, so no one else can use it. Thus guarantees that the operation is atomic.

Table 3.12: Derived Instructions, continued

`TST Rx`	`TST $-1,Rx`	Set the condition codes based upon Rx without changing Rx. Equivalent to a CMP $0,Rx.
`WAIT`	`Or $GIE | $SLEEP,CC`	Wait until the next interrupt, then jump to supervisor/interrupt mode.

Table 3.13: Derived Instructions, continued

place required a return to userspace instruction, `RTU`, once the interrupt takes place the supervisor just simply starts executing code immediately after that `RTU` instruction.

Since the CPU may return from userspace after either an interrupt (hardware generated), a trap (software generated), or an exception (a fault of some type), it is up to the supervisor code that handles the transition to determine which of the three has taken place.

3.3 Memory Architecture

Having now described the CPU registers, instructions, and instruction formats, we now turn our attention to how the CPU interacts with the rest of the world. Specifically, we shall discuss how the bus is implemented, and the memory model assumed by the CPU.

3.3.1 Bus Standards

The ZipCPU (currently) has the ability to operate using one of three bus types: Wishbone (B4, pipelined), AXI-Lite, or AXI (full).

When using Wishbone, several choices have been made to simplify this bus. First, all unnecessary ancillary information has been removed. This includes the retry, tag, lock, cycle type indicator, and burst indicator signals. Second, we insist that all accesses be pipelined. As a result, Wishbone transactions complete whenever either the ERR line goes high or the last ACK has been received.

Further, the ZipCPU is big endian in how it uses the bus.

This becomes a problem when using either AXI-Lite and AXI (full) bus standards, since these standards are specifically little endian. In general, this isn't a problem for the instruction fetch since all instructions are 32-bit words–the words are just ordered so that the MSB stays the MSB regardless of byte order. Things become more difficult for data accesses. Even though the ZipCPU data bus components can naturally handle the AXI bus in the required little-endian fashion, the tool chain doesn't yet fully support this. Where this difference becomes a problem is when accessing peripherals. The AXI specification requires that any big-endian CPU re-order all of its bytes to access a 32-bit peripheral. This would force the CPU to need to rearrange all of the bytes from big to little endian byte order in any 32-bit peripheral access, and would therefore likewise require a change to all 32-bit peripherals so that they would reorder their bytes back to their natural order.

To avoid needing to rearrange all bus accesses in software, the ZipCPU's various AXI memory components have been written with a `SWAP_WSTRB` option. When using `SWAP_WSTRB`, bytes within a 32-bit word are left in their natural order contrary to the AXI specification. 32-bit writes maintain their MSB to LSB order, from left to write, as do 16-bit writes. This is contrary to the AXI specification. However, it allows the ZipCPU to interact with 32-bit peripherals using the ZipCPU's natural byte-order. Sadly, this means that, when interacting with a memory type of peripheral–specifically when interacting with DMAs of any type, then either all components must be adjusted to use this (non-)standard, or the CPU must re–order bytes within 32-bit words in software.

3.3.2 Memory Model

The memory model of the ZipCPU is that of a uniform 32–bit address space. The CPU knows nothing about which addresses reference on–chip or off-chip memory, or even which addresses reference peripherals (outside of the data cache). There are two exceptions to this memory model. The first exception is that the data cache needs to be able to know what addresses can be cached and which ones cannot be cached. The bus compositor must therefore create an *iscachable.v* module that the data cache can reference to know what addresses can be cached. This module examines addresses, and returns a bit indicating if values at the given address can be cached. The second exception applies to the ZipSystem CPU wrapper. When using this wrapper, memory addresses where the most significant 8–bits of 32 are set are reserved for processor local peripherals. These local peripherals will be discussed more in Sec. 7. Other bus wrappers will forward these addresses directly to memory.

The prefetch cache currently has no means of detecting whether instruction memory gets changed outside of the CPU. As a result, any DMA operation should be followed by a manual clearing the instruction and data caches. This may be necessary when loading programs into previously used memory, or when creating self–modifying code.

Should the memory management unit (MMU) be integrated into the ZipCPU, the MMU configuration will replace the *iscachable* module and tell the ZipCPU wich addresses may be cached and which not.

This topic is discussed further in the linker section, Sec. 5.6.1 of the ABI chapter, Chap. 5.

3.4 Debug Interface

The ZipCPU supports an external debug port. This port has a minimum of 64 word address locations. Using this interface, it is possible to both control the CPU, as well as read register values and current status from the CPU.

While a more detailed discussion will be reserved for Sec. 6.1, here we'll just discuss how it is put together. The debug interface allows a controller access to the CPU reset signal, a halt control signal, and a clear cache request signal. By raising the reset line, the CPU will be caused to clear it's cache, to clear any internal exception or error conditions, and then to start execution at the `RESET_ADDRESS`. This will cause the CPU to reboot, while only forcing changes to the CC and PC registers. In a similar fashion, the debug interface allows you to control the `cpu_halt` line into the CPU. Holding this line high will hold the CPU in an externally halted state. Toggling the line low for one clock allows one to step the CPU by one instruction. Lowering the line causes the CPU to go. The final control wire, controlled by the debug interface, will force the CPU to clear its cache. All of these control wires are set or cleared from the external debug control register. This control register occupies the zero address of the ZipCPU's debug register space, and aliases to the next 31 addresses.

The other 32-word addresses are allocated to the various ZipCPU registers, starting with the supervisor registers. This means that a debugger can first halt the CPU and then examine or even modify its full register set, before telling the CPU to continue.

One of the big differences between version 2 and version 3 of the ZipCPU are the address allocations for these registers. In previous versions of the ZipCPU, reading CPU register state required writing the register's address to the control register before reading the register's value back. This proved to be problematic when trying to debug the ZipCPU over a slow link. By creating a separate address for each register, burst read requests may be issued by the debugger for the entire register state. This can greatly speed up interactions between the debugger and the CPU.

Finally, without halting the CPU, the debug controller can read from any single register, and it can see if the CPU is still actively running, whether it is in user or supervisor modes, and whether or not it is sleeping. This alone is useful for detecting deadlocks or other difficult problems.

4. Operation

This chapter will explore how to perform common tasks with the ZipCPU, offering examples in both C and assembly for those tasks.

4.1 CRT0

Of course, the one task that every CPU must do is start the CPU for other tasks. The ZipCPU is no different. This is the one ZipCPU task that must take place in assembly, since no assumptions can be made about the state of the ZipCPU upon entry. In particular, the stack pointer, SP, needs to be loaded with a valid memory location before any higher level language can work. Once that has taken place, it is then possible to call other higher level routines.

Table. 4.1 presents an example of one such initialization routine that first sets up the stack, then calls a bootloader routine to potentially copy program memory from ROM to RAM and zero out any global memory space. Upon completion, the initialization routine then calls main. Should main ever return, this routine will call exit. Finally, once exit completes, a short routine following halts the CPU.

4.2 System High

The easiest and simplest way to run the ZipCPU is just to leave it in its supervisor mode, herein called “System High.” In this mode, the CPU runs your program in supervisor mode from reset to power down, and is never interrupted. You will need to poll the interrupt controller to determine when any external condition has become active. This mode is incredibly useful, and can handle many microcontroller–type tasks.

Even better, in system high mode, all of the user registers are available to the system high program as variables. Accessing these registers can be done in a single clock cycle, which would move them to the active register set or move them back. While this may seem like a load or store instruction, none of these register accesses will suffer from memory delays.

While supervisor mode tasks cannot be interrupted, they can wait for interrupts via the `WAIT` instruction. This instruction can be accessed from C using the `zip_wait()` built–in function. This will place the ZipCPU into an idle/sleep mode to wait for interrupts. Because the supervisor puts the CPU to sleep, rather than the user, no user context needs to be set up.

```
; By starting our loader in the .start section, we guarantee through our
; linker script that these are the very first instructions the CPU sees.
    .section .start
    .global _start
; _start is to be placed at our reboot/reset address, so it will be
; called upon any reboot.
_start:
    ; The most important step: creating a stack pointer. The value
    ; _top_of_stack is created by the linker based upon the linker script.
    LDI _top_of_stack,SP
    ; We then call the bootloader to load our code into memory.
    MOV _after_bootloader(PC),R0
    BRA bootloader
_after_bootloader:
    ; Clear the cache, so any DMA operations will be recognized.
    OR 0xc000,CC
    ; Set argc to zero
    CLR R1
    ; Point argv to NULL
    MOV _argv(PC),R2
    ; A pointer to the environment (often NULL)
    LDI __env,R3
    ; Finally, we call the main function.
    JSR main
; Call the C-library exit function
; If main falls through, then the user hasn't done so, so call it here.
; exit() should not return.
_graceful_kernel_exit:
    JSR exit
; The library exit() function should call _hw_shutdown() on completion.
; Any ongoing hardware operations should have ended before now
    .global _hw_shutdown
_hw_shutdown:
    NEXIT
; Finally, we halt the CPU
_kernel_is_dead:
    HALT
    ; Just in case ...
    BRA _kernel_is_dead ; Provide a dummy value for an empty argv list
_argv:
    .WORD 0,0
```

Table 4.1: Setting up a stack frame and starting the CPU

```
#define EINT(A) (0x80008000|(A<<16)) // Enable interrupt A
#define DINT(A) (A<<16)              // Just disable the interrupts in A
#define DISABLEALL 0x7fff0000        // Disable all interrupts
#define CLEARPIC 0x7fff7fff          // Clears and disables all interrupts
#define SYSINT_TMA 0x10              // The Timer–A interrupt mask

void timer_delay(int nclocks) {
    // Clear the PIC. We want to exit from here on timer counts alone
    zip->pic = DISABLEALL|SYSINT_TMA;
    if (nclocks > 10) {
        // Set our timer to count down the given number of counts
        zip->z_tma = nclocks;
        zip->z_pic = EINT(SYSINT_TMA);
        zip_wait();
        zip->z_pic = CLEARPIC;
    } // else anything less has likely already passed
}
```

Table 4.2: Waiting on a timer

4.3 A Programmable Delay

One common task in microcontrollers, whether in a user task or supervisor task, is to wait for a programmable amount of time. Using the ZipSystem, there are several peripherals that can be used to create such a delay. It can be done with any one of the three timers, the ZipJiffies peripheral, or even an off-chip ZipCounter.

Here, in Tbl. 4.2, we present one means of waiting for a programmable amount of time using a timer. If exact timing is important, you may wish to calibrate the method by subtracting from the counts number the counts it takes to actually do the routine. Otherwise, the timer is guaranteed to at least `counts` ticks.

Notice that the routine clears the PIC early on. While one might expect that this could be done in the instruction immediately before `zip_rtu()`, this isn't the case. The reason is a race condition created by the fact that the write to the PIC might complete after the `zip_rtu()` instruction. (Remember, the ZipCPU doesn't wait for write completion before issuing its next instruction.) As a result, you might find yourself with a zero delay simply because the timer had tripped some time earlier. An alternative way of dealing with this is to read from the PIC after writing to it.

The routine is also careful not to clear any other interrupts beyond the timer interrupt, lest some other condition trip that the user was also waiting on.

4.4 Traditional Interrupt Handling

Although the ZipCPU does not have a traditional interrupt vector architecture, with interrupt vector addresses kept somewhere in memory, it is still possible to create the more traditional interrupt

```
while(true) {
    zip_rtu();
    if (zip_ucc() & CC_TRAPBIT) { // Here, we allow users to install ISRs, or
        // whatever else they may wish to do in supervisor mode.
        ...
    } else (zip_ucc() & (CC_BUSERR|CC_FPUERR|CC_DIVERR)) {
        // Here we handle any faults that the CPU may have encountered
        // The easiest solution is often to print a trace and reboot
        // the CPU.
        _start();
    } else {
        // At this point, we know an interrupt has taken place: Ask the programmable
        // interrupt controller (PIC) which interrupts are enabled and which are active.
        int picv = zip->pic;
        // Turn off all active interrupts
        int active = (picv >> 16) & picv & 0x07fff;
        zip->pic = (active<<16);
        // We build a mask of interrupts to re-enable in picv.
        picv = 0;
        for(int i=0,msk=1; i<15; i++, msk<<=1) {
            if ((active & msk)&&(isr_table[i])) {
                // Here we call our interrupt service routine.
                (isr_table[i])();
        }
    }
}
```

Table 4.3: Traditional Interrupt handling

approach via software. In this mode, the programmable interrupt controller is used together with the supervisor state to create the illusion of more traditional interrupt handling.

To set this up, upon reboot the supervisor task:

1. Creates a (single) user context, a user stack, and sets the user program counter to the entry of the user task
2. Creates a task table of ISR entries
3. Enables the master interrupt enable via the interrupt controller, albeit without enabling any of the fifteen potential underlying interrupts.
4. Switches to user mode, as the first part of the while loop in Tbl. 4.3.

We can work through the interrupt handling process by examining Tbl. 4.3. First, remember, the CPU is always running either the user or the supervisor context. Once the supervisor switches

```
idle_task:
    ; Wait for the next interrupt, then switch to supervisor task
    WAIT
    ; When WAIT completes, the CPU will switch to supervisor mode.
    ; If the supervisor then re-enables this task, it will be because
    ; the supervisor wishes to wait for an interrupt again. For
    ; this reason, we loop back to the top.
    BRA idle_task
```

Table 4.4: Example Idle Task in Assembly

to user mode, control does not return until either an interrupt, a trap, or an exception has taken place. Therefore, if neither the trap bit nor any of the exception bits have been set, then we know an interrupt has taken place.

It is also possible that an interrupt will occur coincident with a trap or exception. If this is the case, the subsequent `zip_rtu()` instruction will return immediately, since the interrupt has yet to be cleared.

As Sec. 7.1 discusses, the top of the PIC register stores which interrupts are enabled, and the bottom stores which have tripped. (Interrupts may trip without being enabled, they just will not generate an interrupt to the CPU.) Our first step is to query the register to find out our interrupt state, and then to disable any interrupts that have tripped. To do that, we write a one to the enable half of the register while also clearing bit fifteen–creating a disable interrupt command.

Using the bit mask of interrupts that have tripped, we walk through all fifteen possible interrupts. If there is an ISR installed, we simply call it here.

There you have it: the ZipCPU, with its non-traditional interrupt architecture, can still process interrupts in a very traditional fashion.

4.5 Idle Task

One task every operating system needs is the idle task, the task that takes place when nothing else can run. On the ZipCPU, this task is quite simple, and it is shown in assembly in Tbl. 4.4, or equivalently in C in Tbl. 4.5.

When this task runs, the CPU will fill up all of the pipeline stages up the ALU. The `WAIT` instruction, upon leaving the ALU, places the CPU into a sleep state where nothing more moves. Then, once an interrupt takes place, control passes to the supervisor task to handle the interrupt. When control passes back to this task, it will be on the next instruction. Since that next instruction sends us back to the top of the task, the idle task thus does nothing but wait for an interrupt.

This should be the lowest priority task, the task that runs when nothing else can. Running this task will reduce power consumption, even stopping the clock if `OPT_CLKGATE` is sset.

```
void idle_task(void) {
    while(true) { // Never exit
    // Wait for the next interrupt, then switch to supervisor task
    zip_wait();
    //
    // When we come back, it's because the supervisor wishes to
    // wait for an interrupt again, so go back to the top.
    }
}
```

Table 4.5: Example Idle Task in C

4.6 Context Switch

Fundamental to any multiprocessing system is the ability to switch from one task to the next. In the ZipSystem, this is accomplished in one of a couple of ways. The first step is that an interrupt, trap, or exception takes place. This will pull the CPU out of user mode and into supervisor mode. At this point, the CPU needs to execute the following tasks:

1. Check for the reason, why did we return from user mode? Did the user execute a trap instruction, or did some other user exception such as a break, bus error, division by zero error, or floating point exception occur. That is, if the user process needs attending then we may not wish to adjust the context, check interrupts, or call the scheduler. Tbl. 4.6 shows the rudiments of this code, while showing nothing of how the actual trap would be implemented.

   You may also wish to note that the instruction before the first instruction in our context swap *must be* a return to userspace instruction. Remember, the supervisor process is re–entered where it left off. This is different from many other processors that enter interrupt mode at some vector or other. In this case, we always enter supervisor mode right where we last left.

2. Capture user accounting counters. If the operating system is keeping track of system usage via the accounting counters, the user counters need to be copied and accumulated into some master user-task counter at this point.

3. Preserve the old context. This involves recording all of the user registers to some supervisor memory structure, such as is shown in Tbl. 4.7. Since this task is so fundamental, the ZipCPU compiler back end provides the `zip_save_context(void *)` function.

4. Reset the watchdog timer. If you are using the watchdog timer, it should be reset on a context swap, to know that things are still working.

5. Interrupt handling. How you handle interrupts on the ZipCPU are up to you. You can activate a sleeping task if you like, or for smaller faster interrupt routines, such as copying a character to or from a serial port or providing a sample to an audio port, you might choose to do the task within the kernel main loop. The difference may depend upon how you have your hardware set up, how fast the kernel main loop is, and how tight your timing requirements are.

```
while(true) {
    // The instruction before the context switch processing must
    // be the RTU instruction that enacted user mode in the first
    // place. We show it here just for reference.
    zip_rtu();

    if (zip_ucc() & (CC_FAULT)) {
        // The user program has experienced an unrecoverable fault and must die.
        // Do something here to kill the task, recover any resources
        // it was using, and report/record the problem.
        ...
    } else if (zip_ucc() & (CC_TRAPBIT)) {
        // Handle any user request
        zip_restore_context(userregs);
        // If the request ID is in uR1, that is now userregs[1]
        switch(userregs[1]) {
        case x: // Perform some user requested function
            break;
        }
    }

}
```

Table 4.6: Checking for whether the user task needs our attention

```
save_context:
    SUB 4,SP        ; Function prologue: create a stack
    SW R5,(SP)      ; frame and save R5. (R1-R4 are assumed
    MOV uR0,R2      ; to be used and in need of saving. Then
    MOV uR1,R3      ; copy the user registers, four at a time to
    MOV uR2,R4      ; supervisor registers, where they can be
    MOV uR3,R5      ; stored, while exploiting memory pipelining
    SW R2,(R1)      ; Exploit memory pipelining:
    SW R3,4(R1)     ; All instructions write to same base memory
    SW R4,8(R1)     ; All offsets increment by one
    SW R5,12(R1)
    ...; Need to repeat for all user registers
    MOV uR12,R2     ; Finish copying ...
    MOV uSP,R3
    MOV uCC,R4
    MOV uPC,R5
    SW R2,48(R1)    ; and saving the last registers.
    SW R3,52(R1)    ; Note that even the special user registers
    SW R4,56(R1)    ; are saved just like any others.
    SW R5,60(R1)
    LW (SP),R5      ; Restore our one saved register
    ADD 4,SP        ; our stack frame,
    RETN            ; and return
```

Table 4.7: Example Storing User Task Context

```
restore_context:
    SUB 4,SP          ; Set up a stack frame
    SW R5,(SP)        ; and store a local register onto it.

    LW (R1),R2        ; By doing four loads at a time, we are
    LW 4(R1),R3       ; making sure we are using our pipelined
    LW 8(R1),R4       ; memory capability.
    LW 12(R1),R5
    MOV R2,uR1        ; Once the registers are loaded, copy them
    MOV R3,uR2        ; into the user registers that they need to
    MOV R4,uR3        ; be placed within.
    MOV R5,uR4
    . . . ; Need to repeat for all user registers
    LW 48(R1),R2      ; Now for our last four registers ...
    LW 52(R5),R3
    LW 56(R5),R4
    LW 60(R5),R5
    MOV R2,uR12       ; These are the special purpose ones, restored
    MOV R3,uSP        ; just like any others.
    MOV R4,uCC
    MOV R5,uPC
    LW (SP),R5        ; Restore our saved register,
    ADD 4,SP          ; and the stack frame,
    RETN              ; and return to where we were called from.
```

Table 4.8: Example Restoring User Task Context

6. Calling the scheduler. This needs to be done to pick the next task to switch to. The next task may be an interrupt handler, or it may be a normal user task. From a priority standpoint, it would make sense that the interrupt handlers all have a higher priority than the user tasks, and that once they have been called the user tasks may then be called again. If no task is ready to run, run the idle task to wait for an interrupt.

   This suggests a minimum of four task priorities:

   (a) Interrupt handlers, executed with their interrupts disabled
   (b) Device drivers, executed with interrupts re-enabled
   (c) User tasks
   (d) The idle task, executed when nothing else is able to execute

7. Restore the new tasks context. Given that the scheduler has returned a task that can be run at this time, the user registers need to be read from the memory at the user context pointer and then placed into the user registers. An example of this is shown in Tbl. 4.8, Because this is such an important task, the ZipCPU GCC provides a built–in function, `zip_restore_context(void *)`, which can be used for this task.

8. Clear the userspace accounting registers. In order to keep track of per process system usage, these registers need to be cleared before reactivating the userspace process. That way, upon the next interrupt, we'll know how many clocks the userspace program has encountered, and how many instructions it was able to issue in those many clocks.

9. Return back to the top of our loop in order to execute `zip_rtu()` again.

5. Tool Suite and Application Binary Interface

This chapter discusses not the CPU itself, but rather how the GCC and binutils toolchains have been configured to support the ZipCPU.

5.1 Executable File Format

ZipCPU executable files are stored in the Executable and Linkable Format (ELF). The ZipCPU loader will use this file to load the executable into flash, or alternatively into whatever memory the program will be executed from.

The ZipCPU described by this specification uses the 16-bits `16'hdad1` to identify itself against other CPUs. This is not an officially registered number, and may change in the future.

The ZipCPU does not (yet) have a dynamic linker/loader. All linking is currently static, and must be done prior to the Zip loader.

5.2 Stack

Register `R13` (also known as the `SP` register) is the stack register. The compiler generates code that grows the stack from high addresses to lower addresses. That means that the stack will usually start out set to a very large value, such as one past the last RAM address, and it will grow to lower and lower values–hopefully never mixing with the heap. Memory at the current stack position is assumed to be allocated.

When creating a stack frame for a function, the compiler will subtract the size of the stack frame from the stack register. It will then store any registers used by the function, from `R5` to `R12` (including the link register `R0`) onto offsets given by the stack pointer plus a constant. If a frame pointer is used, the compiler uses `R12` (or `FP`) for this purpose. The frame pointer is set by moving the stack pointer plus an offset into `FP`. This `MOV` instruction effectively limits the size of any individual stack frame to $2^{12} - 1$ octets.

Once a subroutine is complete, the frame is unwound. If the frame pointer, `FP` was used, then `FP` is copied directly to the stack pointer, `SP`. Registers are restored, starting with `R0` all the way to `R12` (`FP`). This also restores, and obliterates, the subroutine frame pointer. Once complete, a value is added to the stack pointer to return it to its original value, and a jump is made to the value located within `R0`.

5.3 Relocations

The ZipCPU binutils back end supports several types of relocations, although the two most common are the 32–bit relocations for register load and long jump.

The first of these is for loading an arbitrary 32–bit value into a register. Such instructions are broken into a pair of `BREV` and `LDILO` instructions, and once the value of the parameter is known their immediate values can be filled in.

The second type of 32–bit relocation is for jumps to arbitrary addresses. These jumps are supported by the `LW (PC),PC` instruction, followed by the 32–bit address to be filled in later by the linker. If the jump is conditional, then a conditional `LW.`x `4(PC),PC` instruction is used, followed by a `ADD 4,PC` and then the 32–bit relocation value.

If a branch distance is known and within reach, then it will be implemented with an `ADD #,PC` instruction, possibly conditional, as necessary.

While other relocations are supported, they tend not to be used nearly as much as these two.

5.4 Call format

One unique feature of the ZipCPU is that it has no native JSR instruction. The assembler attempts to minimize this problem by replacing a `JSR` *address* instruction with a `MOV #(PC),R0` followed by a jump to the requested address. In this case, the offset to the PC for the `MOV` instruction is determined by whether or not the jump can be accomplished with a local branch or a long jump.

While this works well in practice, this implementation prevents such things as `JSR`'s followed by `BRA`'s from being combined together.

Finally, GCC will place first five operands passed to the subroutine into registers R1–R5, starting with R1. Any additional operands are placed upon the stack.

5.5 Built-ins

The ZipCPU ABI supports the a number of built in functions. The compiler maps these functions directly to assembly language equivalents, essentially providing the C programmer with access to several assembly language instructions. These are:

1. `zip_bitrev(int)` reverses the bits in the given integer, returning the result. This utilizes the internal `BREV` instruction, and is designed to be used with FFT's as necessary.

2. `zip_busy()` executes an `ADD -4,PC` function, essentially forcing the CPU into a very tight infinite loop.

3. `zip_cc()` returns the value of the current CC register. This may be used within both user and supervisor code to determine in which mode the CPU is within.

4. `zip_halt()` executes an `OR $SLEEP,CC` instruction to place the processor to sleep. If the processor is in supervisor mode, this halts the processor.

5. `zip_rtu()` executes an `OR $GIE,CC` instruction. This will place the CPU into user mode, and has no effect if the CPU is already in user mode.

6. `zip_syscall()` executes a `CLR CC` instruction to return the CPU to supervisor mode. This essentially executes a trap, setting the trap bit for the supervisor to examine.

   What this instruction does not do is arrange for the trap arguments to be placed into the registers `R1` through `R5`. If necessary, a function call may be made to an assembly routine that executes the trap if necessary to place the arguments in their proper places.

7. `zip_wait()` executes an `OR $SLEEP|$GIE,CC` instruction. Unlike `zip_halt()`, this `zip_wait()` instruction places the CPU into a wait state regardless of whether or not the CPU is in supervisor mode or not. When this instruction completes, it will leave the CPU in supervisor mode upon an interrupt having taken place.

8. `zip_restore_context(context *)` inserts the 32 assembly instructions necessary to copy all sixteen user registers to a memory region pointed to by the given context pointer, starting with `uR0` on up to `uPC`.

9. `zip_save_context(context *)` inserts the 32 assembly instructions necessary to copy all sixteen user registers to a memory region pointed to by the given context pointer argument, starting with `uR0` on up to `uPC`.

10. `zip_ucc()`, returns the value of the user CC register.

5.6 Linker Scripts

The ZipCPU makes no assumptions about its memory layout. The result, though, is that the memory layout of a given project is board specific. This is accomplished via a board specific linker script. This section will discuss some of the specifics of a ZipCPU linker script.

Because the ZipCPU uses a modified binutils package as part of its tool chain, the format for this linker script is defined by the GNU LD project within binutils. Further details on that format may be found within the GNU LD documentation within the binutils package.

This discussion will focus on those parts of the script specific to the ZipCPU.

5.6.1 Memory Types

Of the FPGA boards that the ZipCPU has been applied to, most of them have some combination of three types of memory: flash, block RAM, and (possibly DDR) Synchronous Dynamic RAM (SDRAM). Of these three, only the flash is non–volatile. The block RAM is the fastest, and the SDRAM the largest. While other memory types are available, such as files on an external media such as an SD card or a network drive, these three types have so far been sufficient for our purposes.

To support these memories, the linker script has three memory lines identifying where each memory exists on the bus, the size of the memory, and any protections associated with it. For example,

$$\text{blkram (wx) : ORIGIN = 0x0008000, LENGTH = 0x0008000}$$

specifies that there is a region of memory, called blkram, that can be read and written, and that programs can execute from. This section starts at address `0x8000` and extends for another `0x8000` bytes. The other memories are defined in a similar manner, with names `flash` and `sdram`.

Following the memory section, three specific symbols are defined:

- `_rom`, defines the beginning of a ROM (i.e. flash) memory area. This is the area where software is placed prior to startup. It may be set to NULL if the program is already loaded in RAM, and doesn't need to be copied to RAM prior to starting.
- `_kram`. Some devices have a faster RAM (i.e. block RAM) than others. `_kram` defines the location of this RAM. If not used, then this value may be left at NULL.
- `_ram`. This defines the beginning of regular RAM mmeory. If both `_rom` and `_ram` are defined, then the CRT0 routine will copy any softare from `_rom` to `_ram` on startup.

These symbols are used to make the bootloader's task easier.

5.6.2 The Entry Function

The ZipCPU has, as a parameter, a `RESET_ADDRESS`. It is important that this address contain a valid instruction (or more), since this is the first instruction the ZipCPU will execute. Traditionally, this address is also the first address in instruction memory as well.

To make this happen, the ZipCPU defines two additional segments: the `.start` and the `.boot` segments. The `.start` segment is to have nothing in it but the very initial startup code. This code typically needs to run from flash (or other ROM). It should be placed at the `RESET_ADDRESS`. This is the purpose of the `.start` section–making sure the `RESET_ADDRESS` has this function.

The `.boot` section has a similar purpose. This section includes anything associated with the bootloader. It is a special section because, when loading from flash, the bootloader *cannot* be placed in RAM, but must be placed in flash–since it is the code that loads things from flash into RAM.

It may also make sense to place any code executed once only within flash as well. Such code may run slower than the main system code, but by leaving it in flash it can be kept from consuming any (potentially precious) higher speed RAM. To do this, place this other code into the `.boot` section.

You may also find that large data structures that are best left in flash can also be placed into this `.boot` section as well for that purpose.

5.6.3 Bootloader Tags

The bootloader needs to know a couple things from the linker script. It needs to know what code/data to copy to block RAM from the flash, what code/data to copy to SDRAM, and finally what initial data area needs to be zeroed. Four additional pointers, set within a linker script, can define these regions.

1. `_kram_start`

   This is the first location in flash containing data that the bootloader needs to move.

2. `_kram_end`

   This is a pointer to one past the last location in block RAM to place things into. If this pointer is equal to `_kram_start`, then no information is placed into _kram.

3. `_ram_image_start`

   This should be equal to one past the last _kram address, if _kram is used, or alternatively the first address in `_rom` containing data to be copied to the RAM memory area. By adding the difference between `_ram_image_start` and `_kram` to the flash address in `_kram_start`, the actual source address within the flash of the code/data that needs to be copied into SDRAM can be determined.

4. `_ram_image_end`

   This is the ending address of any code/data to be copied into _ram. The distance between this pointer and `_ram` should be the total amount of data to be placed into the RAM memory area.

5. `_bss_image_end`

   The BSS segment contains data the starts with an initial value of zero. Such data are usually not placed in the executable file, nor are they placed into any flash image. This address points to the last location in _ram used by the BSS segment. The bootloader is responsible then for clearing the RAM between `_ram_image_end` and `_bss_image_end`.

   The bootloader must also be robust enough to handle the cases where 1) there is no SDRAM, 2) there is no block RAM (_kram is NULL), and 3) where there is non requirement to move memory at all (_rom is NULL)—such as when the program is placed into memory and started from there.

5.6.4 Other required linker symbols

Two other symbols need to be defined in the linker script, which are used by the startup code. These are:

1. `_top_of_stack`

   This is the address that the startup code will set the stack pointer to point to. It may be one past the last location of a RAM memory, whether block RAM or SDRAM.

2. `_top_of_heap`

   This is the first location past the end of the `.bss` segment. Equivalently, this is the address of the first unused piece of memory. It is used as the first location from whence to start any dynamic memory subsystem.

All of these symbols need to reference word aligned addresses.

5.7 Loading ZipCPU Programs

There are two basic ways to load a ZipCPU program, depending upon whether or not the ZipCPU is active within the current configuration. If the ZipCPU is not a part of the current FPGA configuration, one need only write the flash and then switch configurations. It will be the CPU's responsibility to place itself in RAM then.

The more practical alternative is a little more involved, and there are several steps to it.

1. Halt the CPU by writing 0x09 to the CPU control register. This both halts and resets the CPU. It then prevents both bus contention, while writing the new instructions to memory, as well as preventing the CPU from running some instructions from one program and other instructions from another.

2. Load the program into memory. For many programs this will involve loading the program into flash, and necessitate having and using a flash controller. The ZipCPU also supports being loaded straight into RAM address as well, as though the bootloader had completed it's task.
3. You may optionally, at this point, clear all of the CPUs registers, to make certain the reboot is clean.
4. Set the sPC register to the starting address.
5. Clear the instruction cache in order to force the CPU to reload its cache upon start.
6. Release the CPU by writing a zero to the CPU debug control register.

5.8 Starting a ZipCPU program

5.8.1 CRT0

Most computers have a section of code, conventionally called `crt0`, which loads a program into memory. On the ZipCPU, this code starts at `_start`. It is responsible for setting the stack pointer, calling the boot loader, and then calling the main entry function, `entry()`.

Because `_start` *must* be the first symbol in a program, and because that first symbol is located at the boot address for the CPU, the `_start` is placed into the `.start` segment. It is the only routine placed there.

On those CPU's that don't have enough logic space for a debugger, it may be useful to place a routine to dump any registers, stack values and/or kernel traces to an output routine at this time. That way, on any kernel fault, the kernel can be brought back up with a debug trace. This works because rebooting the CPU doesn't reset any register values save the `sCC` and `sPC`.

5.8.2 The Bootloader

As discussed in Sec. 5.6.3, the bootloader must be placed into flash if it is used. It can be a small C program (it need not be assembly, like `_start`), and it only needs to copy memory. First, it copies any memory from flash to block RAM. Second, it copies any necessary memory from flash to SDRAM. Then, it zeros any memory necessary in SDRAM (or block RAM, if there is no SDRAM).

These memory copies may be done with the DMA, or they may be done one–at–a time for a performance penalty.

6. Debug Register Addressing

This chapter covers the definitions and locations of the ZipCPU's registers when accessed by the debugging port. The ZipSystem is special, having access to an extra set of registers within the same address space, so we'll cover that separately.

This chapter also marks a significant upgrade from version 2.0 and prior of the CPU. In particular, each CPU register has now been given its own address location in the debug address space. This should make it easier to read all registers at once via a burst read command of some type.

6.1 Debug Port Registers

When the ZipCPU has been built with the `OPT_DBGPORT` parameter set, then the CPU may be accessed and controlled by an external debug port. This port contains at least 64 word addresses, of which 33 are generally assigned. These registers are shown in Tbl. 6.1. The ZipSystem wrapper contains an additional 64 words as well, which we'll get to in a moment.

The foremost register among these is the ZipCPU Control and Status Register. This register has the fields shown in Tbl. 6.2.

Here are some operations you can do with this register:

1. Reset: To reset the CPU, write an `0x08` to the debug control register. If the CPU was configured to immediately start on reset, then the CPU will start immediately. If not, a second write will be required to release the CPU.
2. Reset and halt: To reset the CPU but leave it halted, write a `0x09` to the debug control register.
3. Start a halted CPU: If the CPU is halted, simply write a `0x00` to the debug control register in order to cause it to continue.
4. Stepping the CPU: To step through a single instruction, write a `0x04` to the debug control register. This will step one instruction through the CPU and then halt it again immediately following that instruction.

   The CPU will not step through compound instructions or LOCK instruction sequences, but rather it will step over them as though they were only a single instruction. Hence, any compound instruction will continue until both instructions have been executed (or one trapped). Similarly, a LOCK instruction will step the full four instructions sequence before halting.
5. Halting the CPU. To halt the CPU, simply write a `0x01` to the debug control register. Once the register reads back a `0x03`, the CPU is halted.

Name	Address	Width	Access	Description
ZIPCTRL	0	32	R/W	ZipCPU Control and Status Register
⋮	⋮			(Reserved addresses)
sR0	128	32	R/W	Supervisor Register R0
sR1	132	32	R/W	Supervisor Register R1
⋮	⋮	32	R/W	Other supervisor registers
sSP	180	32	R/W	Supervisor Stack Pointer
sCC	184	32	R/W	Supervisor Condition Code Register
sPC	188	32	R/W	Supervisor Program Counter
uR0	192	32	R/W	User Register R0
uR1	196	32	R/W	User Register R1
⋮	⋮	32	R/W	Other user registers
uSP	244	32	R/W	User Stack Pointer
uCC	248	32	R/W	User Condition Code Register
uPC	252	32	R/W	User Program Counter

Table 6.1: ZipSystem Debug Registers

Debug port reads from these internal register addresses will return the current value from the CPU's internal register set. If the CPU is still running, the value returned may be out of date as soon as it is returned. For this reason, it makes sense to halt the CPU first.

On the other hand, any attempt to write to an internal CPU register will require that the CPU first be halted. This is accomplished in the wrapper processing the request. Such writes will also leave the CPU in a halted state.

6.1.1 Breakpoint Handling

Breakpoints can be handled via the debug control register. Once a breakpoint has been hit, the CPU will halt, raise its external interrupt flag, and set the break bit in the debug control register.

At this point, the debugger may examine and/or modify any registers as necessary.

Once complete, the breakpoint instruction may be replaced with another instruction, the cache cleared, and that instruction may then be stepped through. The breakpoint may then be replaced and the cache cleared again. The CPU may then be started to go until its next breakpoint.

6.2 ZipSystem Registers

The ZipSystem has an additional set of registers which may be accessed by the CPU. These are associated with the additional peripherals the ZipSystem wrapper provides to the CPU. These registers are listed in Tbl. 6.3. These registers addresses allow an external debugger to have access

Bit #	Access	Description
11	RO	The CPU has suffered from a break condition, and has halted itself as a result.
10	RO	True if an interrupt is pending.
9	RO	GIE. If set, the CPU is currently in user mode.
8	RO	Sleep. The CPU is sleeping.
7–6		(Reserved)
5	R/W	Debug catch bit. If set to '1', then the CPU will halt on any external exception. Otherwise, the CPU will reboot on any exception.
4	WO	Clear cache. Set to '1' to clear both the CPU's caches and its pipelines. This is useful if you have just adjusted memory and now need the CPU to be able to read that adjusted memory. As a side effect, setting this bit will also halt the CPU if it wasn't halted before.
3	R/W	Reset. Set to '1' to reset the CPU. If the CPU has been configured to start in a halted state, it will reset and then halt.
2	WO	Step Command. Set to '1' to step the CPU, and then leave it halted. Self clearing.
1	RO	Halt status. If true, the CPU has come to a complete halt.
0	R/W	Halt request. Set to '1' to halt the CPU.

Table 6.2: Debug Control Register Bits

Name	Address	Width	Access	Description
PIC	256	32	R/W	Primary Interrupt Controller
WDT	260	32	R/W	Watchdog Timer
WBUS	264	32	RO	Address of the Last Bus Error
APIC	268	32	R/W	Secondary Interrupt Controller
TMRA	272	32	R/W	Timer A
TMRB	276	32	R/W	Timer B
TMRC	280	32	R/W	Timer C
JIFF	284	32	R/W	Jiffies peripheral
MTASK	288	32	R/W	Master task clock counter
MMSTL	292	32	R/W	Master memory stall counter
MPSTL	296	32	R/W	Master Pre-Fetch Stall counter
MICNT	300	32	R/W	Master instruction counter
UTASK	304	32	R/W	User task clock counter
UMSTL	308	32	R/W	User memory stall counter
UPSTL	312	32	R/W	User Pre-Fetch Stall counter
UICNT	316	32	R/W	User instruction counter
DMACMD	320	32	R/W	DMA command and status register
DMALEN	324	32	R/W	DMA transfer length
DMASRC	328	32	R/W	DMA read address
DMADST	332	32	R/W	DMA write address

Table 6.3: Debug Register Addresses

to the ZipSystem peripherals as well as the CPU register set. These peripherals may be read or written from the debug data port.

In this manner, the ZipSystem's full internal state may be read and adjusted, in addition to the CPU's internal register state.

7. ZipSystem Peripherals

The ZipSystem CPU wrapper contains a minimal CPU peripheral set which can be accessed internally. Here in this section, we'll walk through the definition of each of these registers in turn, together with any bit fields that may be associated with them, and how to set those fields.

The registers themselves can be found at the address shown in Fig. 7.1. These registers are all 32-bit registers. Writes of less than 32–bits may have unexpected results. Further, they are located in a reserved location within the CPU's address space. As a result, references to these locations by either a ZipBones or an AXI based system may generate a bus error. When using the AXI bus, a separate AXI-lite peripheral set is available to offer all but the DMA's capability. However, the AXI-lite peripheral set is not guaranteed to be in any particular address location.

Name	Address	Width	Access	Description
PIC	`0xff000000`	32	R/W	Primary Interrupt Controller
WDT	`0xff000004`	32	R/W	Watchdog Timer
WBU	`0xff000008`	32	RO	Address of last bus error
APIC	`0xff00000c`	32	R/W	Secondary Interrupt Controller
TMRA	`0xff000010`	32	R/W	Timer A
TMRB	`0xff000014`	32	R/W	Timer B
TMRC	`0xff000018`	32	R/W	Timer C
JIFF	`0xff00001c`	32	R/W	Jiffies
MTASK	`0xff000020`	32	R/W	Master Task Clock Counter
MMSTL	`0xff000024`	32	R/W	Master Stall Counter
MPSTL	`0xff000028`	32	R/W	Master Pre–Fetch Stall Counter
MICNT	`0xff00002c`	32	R/W	Master Instruction Counter
UTASK	`0xff000030`	32	R/W	User Task Clock Counter
UMSTL	`0xff000034`	32	R/W	User Stall Counter
UPSTL	`0xff000038`	32	R/W	User Pre–Fetch Stall Counter
UICNT	`0xff00003c`	32	R/W	User Instruction Counter
DMACTRL	`0xff000040`	32	R/W	DMA Control Register
DMALEN	`0xff000044`	32	R/W	DMA total transfer length
DMASRC	`0xff000048`	32	R/W	DMA source address
DMADST	`0xff00004c`	32	R/W	DMA destination address

Table 7.1: ZipSystem Internal/Peripheral Registers

Here in this section, we'll walk through the definition of each of these registers/peripherals in turn, together with any bit fields that may be associated with them, and how to set those fields.

7.1 Interrupt Controller(s)

Perhaps the most important peripheral within the ZipSystem is the interrupt controller. While the ZipCPU itself can only handle one interrupt, and has only the one interrupt state: disabled or enabled, the interrupt controller can make things more interesting.

The ZipSystem interrupt controller module supports up to 15 interrupts, all controlled from one register. Further, it has been designed so that individual interrupts can be enabled or disabled individually without having any knowledge of the rest of the controller setting. To enable an interrupt, write to the register with bit 15 set and the respective interrupt enable bit set. No other bits will be affected. To disable an interrupt, write to the register bit 15 clear and the respective interrupt enable bit set. To clear an interrupt, write a '1' to that interrupt's status bit. The interrupt enable pin for the global interrupt enable is set and cleared likewise.

As an example, suppose you wished to enable interrupt #4. You would then write to the register a `0x80108010` to enable interrupt #4, interrupts in general, and to clear any past active state for interrupt #4. When you later wish to disable this interrupt, you would write a `0x00100010` to the register. This both disables the interrupt and clears the active indicator. This does not disable interrupts in general, however. To do that you'd also need to set bit 31. Similarly a subsequent write of `0x80000000` will disable all interrupts as well. (Why? Bit 31 selects the global interrupt enable, and bit 15 is clear meaning that the interrupts will be disabled.)

The ZipSystem hosts two interrupt controllers: a primary and a secondary. The primary interrupt controller is the one that interrupts the CPU. It has six local interrupt lines, the rest coming from external interrupt sources. One of those interrupt lines to the primary controller comes from the secondary interrupt controller. This controller maintains an interrupt state for the process accounting counters, and any other external interrupts that didn't fit into the primary interrupt controller.

As a word of caution, because the interrupt controller is an external peripheral, and because memory writes take place concurrently with any following instructions, any attempt to clear interrupts on one instruction followed by an immediate Return to Userspace (`RTU`) instruction, may not have the effect of having interrupts cleared before the `RTU` instruction executes. This is only relevant if the two instructions take place in immediate succession. As an alternative, reading from the interrupt controller after writing to it will place enough time between these two events for the `RTU` command to successfully complete.

Looking into the bits that define this controller, you can see from Tbl. 7.2, that the ZipCPU Interrupt controller has five different types of bits. The high order bit, or bit–31, is the master interrupt enable bit. When this bit is set, then any time an interrupt occurs the CPU will be interrupted and will switch to supervisor mode, etc.

The CPU also has a global interrupt enable bit defined internally as well. This bit is separate from the master interrupt enable in the programmable interrupt controller. Both the PICs master interrupt enable and the CPU's global interrupt enable bit (turning on user mode) will both need to be set for interrupts to be received and processed.

Bits 30 . . . 16 of the PIC are individual interrupt enable bits. Should the respective interrupt line ever be high while its enable line is also high and while the master enable line is high, an interrupt

Bit #	Access	Description
31	R/W	Master Interrupt Enable
30. . . 16	R/W	Interrupt Enable lines
15	R	Current Master Interrupt State
15	W	Set to ‘1’ when writing to the device in order to enable any interrupt lines whose respective bits are set in bits 16-31, ‘0’ to clear them
15. . . 0	R/W	Input Interrupt states, write ‘1’ to clear

Table 7.2: Interrupt Controller Register Bits

will be generated. Further, interrupts are level triggered. Hence, if the interrupt is cleared while the line feeding the controller remains high, then the interrupt will re–trip. To set one of these interrupt enable bits, one needs to write to the controller while both writing a ‘1’ to this bit and a ‘1’ to bit 15. To clear the bit later, one need only write a ‘1’ to this enable bit, while leaving bit 15 low.

Bits 14. . . 0 are the current state of the interrupt vector. Interrupt lines trip whenever they are high, and remain tripped until the input is lowered and the interrupt is acknowledged. To lower an interrupt line, simply write a one to the active interrupt bit to acknowledge it. Since interrupts are level triggered, this will only clear the line if the incoming interrupt has also been cleared. For this reason, it makes sense to clear the interrupt first in the peripheral generating it, and then in the interrupt controller.

As an example, consider the following scenario where the ZipCPU supports four interrupts, 3. . . 0.

1. The Supervisor will first, while in the interrupts disabled mode, write a `32’h800f800f` to the controller. This will enable the master interrupt line, as well as interrupts 0-3 while also clearing any past status from interrupts 0-3. The supervisor may then switch to the user mode to fully enable interrupts.

2. When an interrupt occurs, the CPU will switch to the supervisor mode. It may then cycle through the interrupt bits to learn which interrupt handler to call.

3. If the interrupt handler expects more interrupts, it will clear its current interrupt line when it is done handling the interrupt in question. To do this, it will write a ‘1’ to the low order interrupt mask, such as writing a `32’h0000_0001`.

4. If the interrupt handler does not expect any more interrupts, it will instead clear the interrupt from the controller by writing a `32’h0001_0001` to the controller.

5. The supervisor should also check for any user exceptions here, but this action has nothing to do with the interrupt control register itself.

6. The CPU may then leave supervisor mode, possibly adjusting whichever task is running, by executing a return to userspace instruction.

Bit #	Access	Description
31	R/W	Auto-Reload
30...0	R/W	Current timer value

Table 7.3: Timer Register Bits

7.1.1 Timer Register

The ZipSystem contains three separate timer registers. Each has an identical functionality, and a single control and status register whose bit definitions are given in Tbl. 7.3. This is a very simple timer. It just counts down to zero and then trips an interrupt. Writing to the current timer value sets the value to count down from, and reading from it returns that value. Writing to the current timer value while also setting the auto–reload bit will send the timer into an auto–reload mode. In this mode, upon setting its interrupt bit for one cycle, the timer will also reset itself back to the value of the timer that was written to it when the auto–reload option was written to it. To clear and stop the timer, just simply write a '32'h00' to this register.

This timer may be used for non–interrupt purposes as well. For example, one might write `0x7fff_ffff` to the timer before beginning some operation. Once the operation is complete, the difference between the starting counter's value and its value on completion would then tell you how many clock cycles were used for that operation.

7.1.2 ZipJiffies

The ZipJiffies peripheral is motivated by the Linux use of 'jiffies' whereby a process can request to be put to sleep until a certain number of 'jiffies' have elapsed. Using this interface, the CPU can read the number of 'jiffies' from the peripheral (it only has the one location in address space), add the sleep length to it, and write the result back to the peripheral. The `zipjiffies` peripheral will record the value written to it only if it is nearer the current counter value than the last current waiting interrupt time. If no other interrupts are waiting, and this time is in the future, the peripheral will be enabled. The processor may then place this sleep request into a list of other sleep requests. Once the timer expires, it would write the next Jiffy request to the peripheral and wake up the process whose timer had expired.

Indeed, the Jiffies register is nothing more than a glorified counter with an interrupt. Unlike the other counters, the internal Jiffies counter can only be read, never set. Writes to the jiffies register create an interrupt time. When the Jiffies register later equals the value written to it, an interrupt will be asserted and the register then continues counting as though no interrupt had taken place.

Finally, if the new value written to the Jiffies register is within the past 2^{31-1} clock ticks, the Jiffies register will immediately cause an interrupt and otherwise ignore the new request.

The purpose of this register is to support absolute alarm intervals within a CPU, and moreover to support them within an operating system.

To set an alarm for a particular process N clocks in advance, read the current Jiffies value, add N, and write it back to the Jiffies register. The O/S must also keep track of values written to the Jiffies register. Thus, when an 'alarm' trips, it should be removed from the list of alarms, the list should be resorted, and the next alarm in terms of Jiffies should be written to the register–possibly for a second time.

Bit #	Access	Description
31. . . 0	R	Current jiffy value
31. . . 0	W	Value/time of next interrupt

Table 7.4: Jiffies Register Bits

Similarly, if you wish to set an alarm on an interval, read the current value from the Jiffies register, and add your interval. Keep track of the initial value. Later, when the process is interrupted, you can add your interval to the previous alarm time to achieve an absolute alarm interval.

In many ways, the ZipJiffies register is simply a counter. It just counts up one on every clock. Reads from this register, as shown in Tbl. 7.4, always return the time value contained in the register.

The register accepts writes as well. Writes to the register set the time of the next Jiffy interrupt. If the next interrupt is between 0 and 2^{31} clocks in the past, the peripheral will immediately create an interrupt. Otherwise, the register will compare the new value against the currently stored interrupt value. The value nearest in time to the current jiffies value will be kept, and so the jiffies register will trip at that value. Prior values are forgotten.

When the Jiffy counter value equals the value in its trigger register, then the jiffies peripheral will trigger an interrupt. At this point, the internal register is cleared. It will create no more interrupts unless a new value is written to it.

7.1.3 Watchdog Timer

The watchdog timer has only two differences from the of the other timers. The first difference is that it is a one–shot timer. There is no watchdog interval mode. Writes to the watchdog timer will therefore count down to zero and stop. The second difference, though, is critical: the interrupt line from the watchdog timer is tied to the reset line of the CPU. Hence writing a '1' to the watchdog timer will always reset the CPU. To stop the Watchdog timer, write a '0' to it. To start it, write any other number to it—as with the other timers.

The general usage of the watchdog timer is to write some amount of time to it, equal to the maximum amount of time through the CPU's core processing loop. If every time through the loop the same maximum amount of time is written, all will be well. If something goes wrong, however, and locks up the system, the watchdog timer will detect that it doesn't get restarted on time. The CPU will then be reset. Any additional fault diagnosis, however, will need to be handled at reset. (The watchdog provides no notification that it has tripped.)

7.1.4 Bus Watchdog

There is an additional watchdog timer on the Wishbone bus of the ZipSystem. This timer, however, is hardware configured and not software configured. The timer is reset at the beginning of any bus transaction, and only counts clocks during such bus transactions. If the bus transaction takes longer than the number of counts the timer allots, it will raise a bus error flag to terminate the transaction. This is useful in the case of any peripherals that are misbehaving. If the bus watchdog terminates a bus transaction, the CPU may then read from the bus watchdog's port to find out which memory location created the problem.

Bit #	Access	Description
31...0	R/W	Current counter value

Table 7.5: Counter Register Bits

7.2 Performance Counters

The ZipCPU also supports several counter peripherals. These are very simply: they just count. The current count value is implemented as a single 32–bit register, always incrementing. The ZipCounter cannot be halted. When it rolls over, it issues an interrupt. Writing a value to the counter just sets the current value, and it starts counting again from that value.

It's that simple.

These counters each contain a single register, as shown in Tbl. 7.5. Writes to this register set the new counter value. Reads return the current counter value. Further, each counter will trigger an interrupt on overflow, to allow the CPU to keep track of as many counts as are necessary.

These counters can be configured to count upwards upon any event. Using this capability, eight counters have been assigned the task of performance counting. Two sets of four registers are available for keeping track of performance. The first set tracks all CPU performance, including both supervisor as well as user CPU statistics. The second set tracks user mode statistics only, and will not count in supervisor mode.

The four counters in each set are configured as follows:

1. The first counter counts clock ticks.
2. The second counter counts the number of clock cycles where the CPU is enabled, but no instruction is ready.
3. The third counter counts CPU stalls following the read operands stage. A *stall* in this case indicates that an instruction is ready to execute, but the necessary execution unit is somehow busy.
4. The fourth and final counter keeps track of instructions issued.

It is envisioned that these counters will be used for process accounting as follows: First, every time a master counter rolls over, the supervisor (Operating System) will record the fact. Second, whenever activating a user task, the Operating System will set the four user counters to zero. When the user task has completed, the Operating System will read the timers back off, to determine how much of the CPU the process had consumed. To keep this accurate, the user counters will only increment when the GIE bit is set to indicate that the processor is in user mode.

In practice, these timers have also worked nicely to keep track of runtime performance during time sensitive operations–such as when running benchmarks.

Bit #	Access	Description
31	R	DMA Active. Write a zero to this bit to begin a transaction.
30	R	Bus error, transaction aborted. This bit is set if a bus error is encountered at any time during the transaction. It may be cleared by writing a one to it. New transactions cannot commence without clearing any prior error condition.
29	R/W	Interrupt triggered. If the transfer is interrupt triggered, then the transfer will not start until the interrupt line is high. Keep this bit clear to start the transfer immediately.
28–24	R/W	Interrupt number. Determines which interrupt will trigger the transfer, should the transfer be interrupt triggered.
22	R/W	Set to '1' to prevent the controller from incrementing the destination address, '0' for normal memory copy.
21–20	R/W	Control the word size of the destination device.
18	R/W	Set to '1' to prevent the controller from incrementing the source address, '0' for normal memory copy.
17–16	R/W	Control the word size of the source device.
11...0	R/W	Intermediate transfer length. Thus, to transfer one byte per interrupt, set this value to 1. To transfer the maximum size, set it to 0.

Table 7.6: DMA Control Register Bits

7.2.1 ZipDMA Controller

As of Version 3 of the ZipCPU, the ZipSystem now has a new DMA controller called the ZipDMA. This is a Wishbone DMA controller designed to handle unaligned transfers in a bus-width independent fashion.[1] This DMA controller may also be used as an independent peripheral if desired.

This ZipDMA controller has four registers. Of these four, the transfer length, source and destination address registers should need no further explanation. They are full 32–bit registers specifying the entire transfer length, the starting address to read from, and the starting address to write to. These registers can be written to any time the DMA is idle, and they can also be read at any time. The control register, however, will need some more explanation.

The bit allocation of the control register is shown in Tbl. 7.6. This control register has been designed so that the common case of memory access need only set the key and the transfer length. Hence, writing a 32'h0000 to the control register will start any memory transfer. On the other hand, if you wished to read a single byte from a serial port (constant address), connected to interrupt zero, and then to place that result into a buffer every time a byte was available, you might wish to write 32'h20070001. (Note that the DMA controller does not use the interrupt controller, and so all interrupts must be self clearing when using the DMA.) As a third example, if you wished to write to 16-bytes to a serial port transmit FIFO anytime it was less than half full, and this half full interrupt line was number 3, then you might wish to issue a 32'h20700010 to this port.

[1]A separate AXI controller is scheduled for development.

2'b00	Full bus width, highest speed DMA transfer.
2'b01	Transfer 32-bits at a time.
2'b10	Transfer 16-bits per beat.
2'b11	Transfer 8-bits per beat.

Table 7.7: ZipDMA Word Size Enumeration

Both source and destination can be configured to support word sizes less than a full bus word in length, as enumerated in Tbl. 7.7. When using word sizes other than the full bus width, then both source and destination addresses, together with the transfer length, will need to be aligned to the word size used. No particular alignment is required, however, when using either 8-bit transfers or the full bus width.

In most cases, the transfer length can be understood as the number of bytes that will be read and written per interrupt. The exception to this is when not using interrupt based transfers. In this case, and because Wishbone can only go in one direction at a time, a state machine within the ZipDMA will break the transfer size up into packets, each a transfer length in size. The packet will first be read into an internal buffer, and then written out. This process will repeat until the entire transfer has completed.

8. Integration

8.1 ZipCPU Parameters

One problem with a simple goal such as being light on logic, is that some architectures have some needs, others have other needs. What constitutes minimum area in some architectures might consume all the available logic in others. As an example, the CMod S6 board built by Digilent uses a very spare Xilinx Spartan 6 LX4 FPGA. This FPGA doesn't have enough look up tables (LUTs) to support pipelined mode, whereas another project running on a XuLA2 LX25 board made by Xess, having a Spartan 6 LX25 on board, has more than enough logic to support a pipelined mode. Even better, the Artix–7 35T has not only enough logic for a pipelined mode, but plenty of RAM to support instruction and data caches as well. Very quickly it becomes clear that LUTs and RAMs can be traded for performance.

To make tailored configurations possible, the ZipCPU has a set of parameters that can be used to configure its logic usage and performance. This section, therefore, will go through each of the ZipCPU's configuration parameters and explain its meaning.

The first several parameters control the ZipCPU's design as a whole, such as whether or not the CPU is pipelined, how the register file is implemented, or how big the caches are.

- `OPT_PIPELINED`: The first performance question is whether or not the ZipCPU will operate on multiple instructions at a time in a pipelined fashion. Since the ZipCPU is fundamentally a pipelined architecture, setting this value to zero will primarily simplify the pipeline stall logic and remove any duplicated registers throughout the pipeline so that only one instruction is ever allowed into the pipeline at a time. While this can be used to lower logic, it doesn't fundamentally affect the pace of a single instruction's execution.

- `OPT_EARLY_BRANCHING`: When enabled, the ZipCPU may branch on an unconditional branch instruction as soon as it is recognized by the instruction decoder. This minimizes the pipeline stalls associated with branching, for a small additional logic cost.

- `OPT_DISTRIBUTED_REGS`: The ZipCPU normally keeps its registers in distributed RAM, where it can read them on the same clock cycle their address becomes available. This doesn't work, however, on an iCE40 FPGA since iCE40 FPGAs don't have distributed RAM. Instead, iCE40 FPGAs require a clock to read from RAM, and they also require that any read of the RAM go immediately and unconditionally into a registered output. This parameter exists to support the iCE40 and similar architectures. Set this parameter to zero to implement the register file in such a way that all register file reads are registered on a clock edge.

- `OPT_LGICACHE`: This specifies the log, based two, of the instruction cache size. A value of one or less will result in no instruction cache. A value of two (for Wishbone) or four or less (for AXI4 and AXI4–lite) will result in a small pipelined memory reader–with no cache. Anything larger will specify the size of the instruction cache size.
- `OPT_LGDCACHE`: Specifies the log, based two, of the data cache size. As with the instruction cache, a value of zero will yield a basic memory controller with no cache. If `OPT_PIPELINED` is true, a pipelined memory controller may be used that allows multiple requests to be outstanding at once. For anything larger than two, this value specifies the log of the data cache size.
- `OPT_LOWPOWER`: The ZipCPU has been designed as a low logic processor. Low logic, however, doesn't necessarily low power. If the `OPT_LOWPOWER` parameter is set, the ZipCPU will attempt to either minimize unused state transitions or else pin their values to zero.

The next several parameters control the instruction set, and whether or not all instructions are implemented or not.

- `OPT_MPY`: Controls the algorithm that will be used to implement a multiply. If left at zero, any multiply instruction will result in an illegal instruction error. Values of 1–4 will generate a multiply algorithm requiring that many cycles. Of those, 1–2 are not well protected from the critical path and may not be usable. `OPT_MPY = 3` is the workhorse for most Xilinx series 7 parts. It involves registering the multiply inputs on the first clock, performing the multiply itself on the second, and the registering and returning those results on a third clock cycle. `OPT_MPY = 4` is similar to `OPT_MPY = 3`, save that a binomial multiply formula is used, requiring an extra clock cycle. This seems to work well on Spartan 6 FPGAs.

  If an FPGA doesn't have DSPs, and therefore no native hardware accelerated multiply support, then any value greater than four will generate a low logic shift–add algorithm that should work on any architecture at the cost of about 33 clock cycles per multiply.
- `OPT_DIV`: If set, the ZipCPU will include a divide unit to support divide instructions. If not, any attempt to issue a divide instruction will result in an illegal instruction error.
- `OPT_SHIFTS`: If set, the ZipCPU will support shift instructions: `LSR` (logical shift right), `ASR` (arithmetic shift right), and `LSL` (logical shift left) for any number of shifts. If clear, these instructions will only ever shift by one bit.

  Unlike divide or multiply instructions, if `OPT_SHIFTS` is clear the CPU will not generate an illegal instruction error. Instead, it will quietly execute the wrong instruction. As a result, there's no current way to trap an unimplemented shift and replace it with software. Worse, GCC support currently requires that this value be set, so turning this option off is quite problematic.

  Shift register support costs a couple hundred LUTs, and hence the reason for this option. A very low logic implementation might wish to keep this parameter clear.

  Note that the ZipCPU GCC compiler port does not support this option.
- `OPT_CIS`: Controls whether or not the instruction decoder includes the logic necessary to handle the compressed instruction set.

- `OPT_LOCK`: Atomic access on the ZipCPU requires the use of a LOCK instruction. If this value is set, the LOCK instruction will be supported. If clear, LOCK instructions will generate illegal instruction errors.

  LOCK instructions should only be necessary in multitasking or interrupt environments.

- `OPT_SIM`: The ZipCPU supports several simulation–only instructions. If this value is set, these instructions will be decoded and processed through the pipeline. If clear, any simulation only instruction will be converted into either a NOOP or an illegal instruction–depending upon the instruction opcode in question.

The final set of parameters control the ZipCPU's wrappers, and hence its environment.

- `OPT_START_HALTED`: The ZipCPU can be configured to immediately start processing instructions on power up, or it can be configured to wait for a command on the debug port before starting. If `OPT_START_HALTED` is set, then the ZipCPU will stay in its halted state upon reset and so wait for a command before starting.

  This `START_HALTED` option is useful for debugging, since it prevents the CPU from doing anything without supervision. Of course, once all pieces of your design are in place and proven, you'll probably want to set this to zero, so that the CPU will then start up immediately upon power up.

- `RESET_DURATION`: Some architectures, such as the iCE40s, require that the CPU stay halted for a particular period of time before first attempting to access memory or other peripherals. This behavior can be controlled by the `RESET_DURATION` parameter, controlling how long the CPU remains in reset before leaving its initial halted state. Set this value to zero to start immediately following a reset.

- `OPT_TRACE_PORT`: The ZipCPU has a trace port that can be useful for debugging the CPU in hardware if necessary. The trace port is a 32-bit output containing internal details of the CPU on a clock by clock basis. For example, it is possible to watch what is happening within the pipeline, what registers are getting set to what values and more by observing the trace port. This port, however, costs area, so it is not enabled by default. When not enabled, the debug trace port output will be clamped at zero.

- `OPT_CLKGATE`: All of the ZipCPU's wrappers now include a clock gating option controlled by this parameter. If set to one, clock gating will be enabled and the ZipCPU will halt its clock any time it is halted or asleep.

  This option is designed for optimizing simulations only, and has not yet been tested on any FPGAs.

- `OPT_DMA`: The ZipSystem wrapper optionally contains a Wishbone DMA. When this option is set, this ZipDMA will be included. If clear, the ZipSystem will be built without an internal DMA.

- `OPT_ACCOUNTING`: The ZipSystem wrapper also includes a set of eight counters which can be used for process accounting. This parameter controls whether or not these accounting timers are inclued into the ZipSystem wrapper or not.

When using either the AXI–Lite or AXI wrappers, these accounting registers have been moved into an external AXI peripheral package.

- `EXTERNAL_INTERRUPTS`: Controls the number of interrupt wires coming into the ZipSystem CPU wrapper. This number must be between one and sixteen, or if the performance counters are disabled, between one and twenty four.

  All other wrappers only allow a single incoming interrupt wire.

- `RESET_ADDRESS`: The `RESET_ADDRESS` parameter controls what address the CPU will attempt to fetch its first instruction from upon any CPU reset. The default value is not likely to be particularly useful, so overriding the default is recommended for every implementation.

- `ADDRESS_WIDTH`: The `ADDRESS_WIDTH` parameter configures the size of the ZipCPU's address space *in bytes* (not words). This parameter can be used to trim down the width of the address registers used by the CPU. For example, although the ZipCPU will support a 32-bit addressing, particular implementations may only implement a smaller subset of these bits. By setting this value to the actual size of the external address space, some logic may be spared within the CPU. The default is also the maximum, a 32–bit address width.

- `BUS_WIDTH`: Specifies the bit width of the bus. The minimum bus width is 32-bits, although both instruction and data bus protocol handlers should be able to accommodate larger bus widths as necessary.

8.2 Clocks

The ZipCPU has now been tested and proven on the Xilinx Spartan 6 FPGA, as well as the Artix–7 FPGA.

Name	Source	Rates (MHz)		Description
		Max	Min	
i_clk	External	100 MHz		System clock, Artix–7/35T
		80 MHz		System clock, Spartan 6

Table 8.1: List of Clocks

On a SPARTAN 6, the clock can run successfully at 80 MHz.

When running on Digilent's Arty board, the clock is limited to 81.25 MHz by the memory interface generated (MIG) core used to access SDRAM.

Others have described running the ZipCPU successfully at 140 MHz on a Kintex–7.

8.3 I/O Ports

This chapter presents and outlines the various I/O lines in and out of the ZipSystem. Since the ZipCPU can only ever be a component of a larger system, connecting these I/O lines is an important part of integration.

Port	Width	Direction	Description
`o_wb_cyc`	1	Output	Indicates an active Wishbone cycle
`o_wb_stb`	1	Output	WB Strobe signal
`o_wb_we`	1	Output	Write enable
`o_wb_addr`	30	Output	Bus address
`o_wb_data`	32	Output	Data on WB write
`o_wb_sel`	4	Output	Select lines
`i_wb_stall`	1	Input	WB bus slave not ready
`i_wb_ack`	1	Input	Slave has completed a R/W cycle
`i_wb_data`	32	Input	Incoming bus data
`i_wb_err`	1	Input	Bus Error indication

Table 8.2: CPU Master Wishbone I/O Ports

Port	Width	Direction	Description
`i_dbg_cyc`	1	Input	Indicates an active Wishbone cycle
`i_dbg_stb`	1	Input	WB Strobe signal
`i_dbg_we`	1	Input	Write enable
`i_dbg_addr`	7	Input	Debug port register address
`i_dbg_data`	32	Input	Data on WB write
`o_dbg_stall`	1	Output	WB bus slave not ready
`o_dbg_ack`	1	Output	Slave has completed a R/W cycle
`o_dbg_data`	32	Output	Incoming bus data

Table 8.3: CPU Debug Wishbone I/O Ports

The I/O ports to the ZipSystem may be grouped into three categories. The first is that of the master wishbone used by the CPU, then the slave wishbone used to command the CPU via a debugger, and then the rest. The first two of these were already discussed in the wishbone chapter. They are listed here for completeness in Tbl. 8.2 and 8.3 respectively.

There are four other basic lines to the CPU: the external clock, external reset, incoming external interrupt line(s), and the outgoing debug interrupt line. These are shown in Tbl. 8.4. The clock line was discussed briefly in Sec. 8.2. The reset line is a synchronous, active high, system reset line. It should be asserted on power up. Once released, assuming `START_HALTED` is clear, the CPU will start running from its `RESET_ADDRESS` in memory. The `i_ext_int` input contains a set of external interrupt lines to the ZipSystem. This line may actually be as wide as 16 external interrupts, depending upon the setting of the `EXTERNAL_INTERRUPTS` parameter. Finally, the ZipSystem produces one external interrupt. This will be set whenever the entire CPU comes to a halt to wait for the debugger.

Other I/O lines exist to support particular option. For example, the trace port contains a 32-bit output. These 32 bits will have one of the encodings shown in Fig. 8.1. These may be understood as follows: if a register is being written, then the register's address and lower 26–bits of its value are provided. Otherwise, on any jump, the lower 28–bits of the new program counter are provided. In

Port	Width	Direction	Description
`i_clk`	1	Input	The master CPU clock
`i_reset`	1	Input	Active high reset line
`i_ext_int`	1…16	Input	Incoming external interrupts, actual value set by implementation parameter. This is only ever one for the ZipBones implementation.
`o_ext_int`	1	Output	CPU Halted interrupt

Table 8.4: I/O Ports

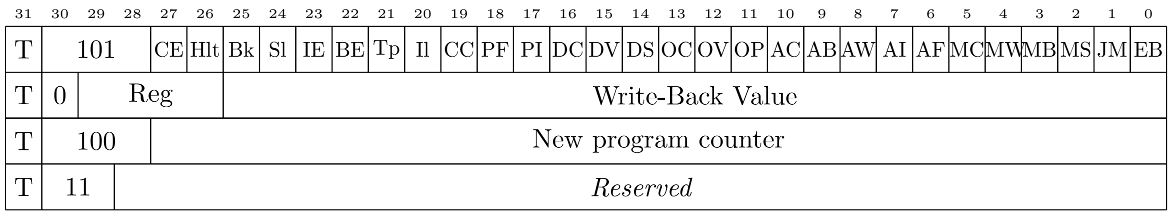

Figure 8.1: Trace Port encodings

all other cases, the internal CPU operational flags are dumped. Tbl. 8.5 shows the meaning of these bits.

Even though the trace port bits are rarely enough to reconstruct all of what takes place within the ZipCPU's core, they have historically been enough to diagnose any faults taking place within hardware.

The profiler consists of three outputs, as shown in Tbl. 8.6.

The I/O lines to the ZipBones package are identical to those of the ZipSystem, with the only exception that the ZipBones package has only a single interrupt line input. This means that the ZipBones implementation practically depends upon an external interrupt controller.

8.4 Wishbone Datasheets

Both the ZipSystem and ZipBones wrappers supports two wishbone ports, a slave debug port and a master port for the system itself. These are shown in Tbl. 8.7 and Tbl. 8.8 respectively. I do not recommend that you connect these together through the interconnect, since 1) it doesn't make sense that the CPU should be able to halt itself, and 2) it helps to be able to reboot the CPU in case something has gone terribly wrong and the CPU is stalling the entire interconnect. Rather, the debug port of the CPU should be accessible regardless of the state of the master bus.

You may wish to notice that neither the `LOCK` nor the `RTY` (retry) wires have been connected to the CPU's master interface. If necessary, a rudimentary `LOCK` may be created by tying this wire to the `wb_cyc` line. As for the `RTY`, all the CPU recognizes at this point are bus errors—it cannot tell

CE	Master chip enable is set, CPU is running
Hlt	A halt has been requested
Bk	External Break
Sl	CPU sleep request
IE	Interrupts are enabled, CPU is in user mode
BE	Supervisor bus error flag
Tp	Trap active
Il	Supervisor illegal instruction flag
CC	Clear cache request
PF	Prefetch valid instruction
PI	Prefetch instruction is illegal (bus err response)
DC	The decode stage is enabled
DV	A decoded instruction is available
DS	The decode stage is stalled
OP	The operand stage is enabled
OV	The operand stage has a valid instruction
OP	
AC	The ALU is enabled for this instruction
AB	The ALU is busy
AW	The ALU will write its result when ready
AF	The ALU will write flags once ready
MC	A memory request is issued
MW	The request is to write memory
MB	The memory unit is busy
MS	The memory unit is stalled, and will not accept a subsequent memory request
JM	A new program counter is available
EB	The decoder is executing an early branch

Table 8.5: Trace port flag bits

Port	Width	Direction	Description
`o_prof_stb`	1	Output	Set when the CPU moves on to the next instruction
`o_prof_addr`	30	Output	The address of the next instruction
`o_prof_ticks`	31...0	Output	The number of clock ticks since startup, for which the CPU has not been halted. This can be used to calculate the number of ticks per instruction.

Table 8.6: Profiler outputs

Description	Specification	
Revision level of wishbone	WB B4 spec	
Type of interface	Slave, Read/Write, single words only	
Address Width	7b for ZipSystem, 6b for the ZipBones wrapper	
Port size	32–bit	
Port granularity	32–bit	
Maximum Operand Size	32–bit	
Data transfer ordering	(Irrelevant)	
Clock constraints	See Sec. 8.1	
Signal Names	Signal Name	Wishbone Equivalent
	`i_clk`	`CLK_I`
	`i_dbg_cyc`	`CYC_I`
	`i_dbg_stb`	`(CYC_I)&(STB_I)`
	`i_dbg_we`	`WE_I`
	`i_dbg_addr`	`ADR_I`
	`i_dbg_data`	`DAT_I`
	`o_dbg_ack`	`ACK_O`
	`o_dbg_stall`	`STALL_O`
	`o_dbg_data`	`DAT_O`

Table 8.7: Wishbone Datasheet for the Debug Interface

Description	Specification	
Revision level of wishbone	WB B4 spec	
Type of interface	Master, Read/Write, pipelined	
Address Width	Configurable, maximum width references 2^{32} bytes	
Port size	Configurable, minimum width is 32–bits	
Port granularity	8–bit	
Maximum Operand Size	32–bit from the CPU, full bus size from the ZipDMA	
Data transfer ordering	Big–Endian	
Clock constraints	See Sec. 8.1	
Signal Names	Signal Name	Wishbone Equivalent
	`i_clk`	`CLK_O`
	`o_wb_cyc`	`CYC_O`
	`o_wb_stb`	`(CYC_O)&(STB_O)`
	`o_wb_we`	`WE_O`
	`o_wb_addr`	`ADR_O`
	`o_wb_data`	`DAT_O`
	`o_wb_sel`	`SEL_O`
	`i_wb_ack`	`ACK_I`
	`i_wb_stall`	`STALL_I`
	`i_wb_data`	`DAT_I`
	`i_wb_err`	`ERR_I`

Table 8.8: Wishbone Datasheet for the CPU as Master

the difference between a temporary and a permanent bus error. Therefore, one might logically OR the bus error and bus retry flags on input to the CPU's `i_wb_err` flag for compatibility if necessary.

The final simplification made of the standard wishbone bus B4 specification, is that the strobe lines are assumed to be zero in any slave if `CYC_I` is zero, and the master is responsible for ensuring that `STB_O` is never true when `CYC_O` is true in order to make this work. All of the ZipCPU Wishbone components have have been designed with this assumption. Converting peripherals that have made this assumption to work with masters that don't guarantee this property is as simple as anding the slave's `CYC_I` and `STB_I` lines together. No change needs to be made to any ZipCPU master, however, in order to access any peripheral that hasn't been so simplified.

8.5 AXI/AXI-Lite Datasheets

AXI4 integration data sheets are not required by the AXI4 specification.