Gisselquist
Technology, LLC

ZIP CPU
SPECIFICATION

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

December 21, 2015

Revision History

Rev.	Date	Author	Description
0.7	12/20/2015	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 Zip CPU. ARM processors are good and effective. Xilinx makes and markets Microblaze, Altera Nios, and both have better toolsets than the Zip CPU will ever have. OpenRISC is also available, RISC–V may be replacing it. 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 that I would like to do with a processor, and I want to be able to do it in a vendor independent fashion. First, I would like to be able to place this processor inside an FPGA. Without paying royalties, ARM is out of the question. I would then 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 and Altera chips, and that can be easily ported from one manufacturer's chipsets to another. Even more, before purchasing a chip or a board, I would like to know that my soft core works. I would like to build a test bench to test components with, and Verilator is my chosen test bench. This forces me to use all Verilog, and it prevents me from using any proprietary cores. For this reason, Microblaze and Nios are out of the question.

Why not OpenRISC? That's a hard question. The OpenRISC team has done some wonderful work on an amazing processor, and I'll have to admit that I am envious of what they've accomplished. I would like to port binutils to the Zip CPU, as I would like to port GCC and GDB. They are way ahead of me. The OpenRISC processor, however, is complex and hefty at about 4,500 LUTs. It has a lot of features of modern CPUs within it that ... well, let's just say it's not the little guy on the block. The Zip CPU is lighter weight, costing only about 2,300 LUTs with no peripherals, and 3,200 LUTs with some very basic peripherals.

My final reason is that I'm building the Zip CPU as a learning experience. The Zip CPU 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.

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

Dan Gisselquist, Ph.D.

1.

Introduction

The original goal of the Zip CPU was to be a very simple CPU. You might think of it as a poor man's alternative to the OpenRISC architecture. For this reason, all instructions have been designed to be as simple as possible, and the base instructions are all designed to be executed in one instruction cycle per instruction, barring pipeline stalls. Indeed, even the bus has been simplified to a constant 32-bit width, with no option for more or less. This has resulted in the choice to drop push and pop instructions, pre-increment and post-decrement addressing modes, and more.

For those who like buzz words, the Zip CPU is:

- A 32-bit CPU: 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. All instructions are designed to be completed in one clock cycle.
- A Load/Store architecture. (Only load and store instructions can access memory.)
- Wishbone compliant. All peripherals are accessed just like memory across this bus.
- A Von-Neumann architecture. (The instructions and data share a common bus.)
- A pipelined architecture, having stages for **Prefetch**, **Decode**, **Read-Operand**, a combined stage containing the **ALU**, **Memory**, **Divide**, and **Floating Point** units, and then the final **Write-back** stage. See Fig. 1.1 for a diagram of this structure.
- Completely open source, licensed under the GPL.[1]

The Zip CPU also has one very unique feature: the ability to do pipelined loads and stores. This allows the CPU to access on-chip memory at one access per clock, minus a stall for the initial access.

1.1 Characteristics of a SwiC

Here, we shall define a soft core internal to an FPGA as a "System within a Chip," or a SwiC. SwiCs have some very unique properties internal to them that have influenced the design of the Zip CPU. Among these are the bus, memory, and available peripherals.

Most other approaches to soft core CPU's employ a Harvard architecture. This allows these other CPU's to have two separate bus structures: one for the program fetch, and the other for the memory.

[1] Should you need a copy of the Zip CPU licensed under other terms, please contact me.

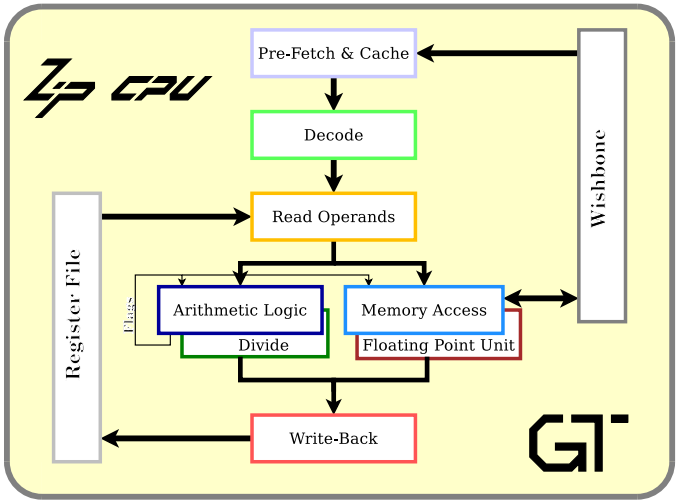

Figure 1.1: Zip CPU internal pipeline architecture

The Zip CPU is fairly unique in its approach because it uses a von Neumann architecture. This was done for simplicity. By using a von Neumann architecture, only one bus needs to be implemented within any FPGA. This helps to minimize real-estate, while maintaining a high clock speed. The disadvantage is that it can severely degrade the overall instructions per clock count.

Soft core's within an FPGA have an additional characteristic regarding memory access: it is slow. While memory on chip may be accessed at a single cycle per access, small FPGA's often have only a limited amount of memory on chip. Going off chip, however, is expensive. Two examples will prove this point. On the XuLA2 board, Flash can be accessed at 128 cycles per 32–bit word, or 64 cycles per subsequent word in a pipelined architecture. Likewise, the SDRAM chip on the XuLA2 board allows a 6 cycle access for a write, 10 cycles per read, and 2 cycles for any subsequent pipelined access read or write. Either way you look at it, this memory access will be slow and this doesn't account for any logic delays should the bus implementation logic get complicated.

As may be noticed from the above discussion about memory speed, a second characteristic of memory is that all memory accesses may be pipelined, and that pipelined memory access is faster than non–pipelined access. Therefore, a SwiC soft core should support pipelined operations, but it should also allow a higher priority subsystem to get access to the bus (no starvation).

As a further characteristic of SwiC memory options, on-chip cache's are expensive. If you want to have a minimum of logic, cache logic may not be the highest on the priority list.

In sum, memory is slow. While one processor on one FPGA may be able to fill its pipeline, the same processor on another FPGA may struggle to get more than one instruction at a time into the pipeline. Any SwiC must be able to deal with both cases: fast and slow memories.

A final characteristic of SwiC's within FPGA's is the peripherals. Specifically, FPGA's are highly reconfigurable. Soft peripherals can easily be created on chip to support the SwiC if necessary. As an example, a simple 30-bit peripheral could easily support reversing 30-bit numbers: a read from the peripheral returns it's bit–reversed address. This is cheap within an FPGA, but expensive in

instructions. Reading from another 16–bit peripheral might calculate a sine function, where the 16–bit address internal to the peripheral was the angle of the sine wave.

Indeed, anything that must be done fast within an FPGA is likely to already be done–elsewhere in the fabric. This leaves the CPU with the simple role of solely handling sequential tasks that need a lot of state.

This means that the SwiC needs to live within a very unique environment, separate and different from the traditional SoC. That isn't to say that a SwiC cannot be turned into a SoC, just that this SwiC has not been designed for that purpose.

1.2 Lessons Learned

Now, however, that I've worked on the Zip CPU for a while, it is not nearly as simple as I originally hoped. Worse, I've had to adjust to create capabilities that I was never expecting to need. These include:

- **External Debug:** Once placed upon an FPGA, some external means is still necessary to debug this CPU. That means that there needs to be an external register that can control the CPU: reset it, halt it, step it, and tell whether it is running or not. My chosen interface includes a second register similar to this control register. This second register allows the external controller or debugger to examine registers internal to the CPU.

- **Internal Debug:** Being able to run a debugger from within a user process requires an ability to step a user process from within a debugger. It also requires a break instruction that can be substituted for any other instruction, and substituted back. The break is actually difficult: the break instruction cannot be allowed to execute. That way, upon a break, the debugger should be able to jump back into the user process to step the instruction that would've been at the break point initially, and then to replace the break after passing it.

  Incidentally, this break messes with the prefetch cache and the pipeline: if you change an instruction partially through the pipeline, the whole pipeline needs to be cleansed. Likewise if you change an instruction in memory, you need to make sure the cache is reloaded with the new instruction.

- **Prefetch Cache:** My original implementation, `prefetch`, had a very simple prefetch stage. Any time the PC changed the prefetch would go and fetch the new instruction. While this was perhaps this simplest approach, it cost roughly five clocks for every instruction. This was deemed unacceptable, as I wanted a CPU that could execute instructions in one cycle.

  My second implementation, `pipefetch`, attempted to make the most use of pipelined memory. When a new CPU address was issued, it would start reading memory in a pipelined fashion, and issuing instructions as soon as they were ready. This cache was a sliding window in memory. This suffered from some difficult performance problems, though. If the CPU was alternating between two diverse sections of code, both could never be in the cache at the same time–causing lots of cache misses. Further, the extra logic to implement this window cost an extra clock cycle in the cache implementation, slowing down branches.

  The Zip CPU now has a third cache implementation, `pfcache`. This new implementation takes only a single cycle per access, but costs a full cache line miss on any miss. While configurable,

a full cache line miss might mean that the CPU needs to read 256 instructions from memory before it can execute the first one of them.

- **Operating System:** In order to support an operating system, interrupts and so forth, the CPU needs to support supervisor and user modes, as well as a means of switching between them. For example, the user needs a means of executing a system call. This is the purpose of the **'trap'** instruction. This instruction needs to place the CPU into supervisor mode (here equivalent to disabling interrupts), as well as handing it a parameter such as identifying which O/S function was called.

  My initial approach to building a trap instruction was to create an external peripheral which, when written to, would generate an interrupt and could return the last value written to it. In practice, this approach didn't work at all: the CPU executed two instructions while waiting for the trap interrupt to take place. Since then, I've decided to keep the rest of the CC register for that purpose so that a write to the CC register, with the GIE bit cleared, could be used to execute a trap. This has other problems, though, primarily in the limitation of the uses of the CC register. In particular, the CC register is the best place to put CPU state information and to "announce" special CPU features (floating point, etc). So the trap instruction still switches to interrupt mode, but the CC register is not nearly as useful for telling the supervisor mode processor what trap is being executed.

  Modern timesharing systems also depend upon a **Timer** interrupt to handle task swapping. For the Zip CPU, this interrupt is handled external to the CPU as part of the CPU System, found in `zipsystem.v`. The timer module itself is found in `ziptimer.v`.

- **Bus Errors:** My original implementation had no logic to handle what would happen if the CPU attempted to read or write a non-existent memory address. This changed after I needed to troubleshoot a failure caused by a subroutine return to a non-existent address.

  My next problem bus problem was caused by a misbehaving peripheral. Whenever the CPU attempted to read from or write to this peripheral, the peripheral would take control of the wishbone bus and not return it. For example, it might never return an `ACK` to signal the end of the bus transaction. This led to the implementation of a wishbone bus watchdog that would create a bus error if any particular bus action didn't complete in a timely fashion.

- **Pipeline Stalls:** My original plan was to not support pipeline stalls at all, but rather to require the compiler to properly schedule all instructions so that stalls would never be necessary. After trying to build such an architecture, I gave up, having learned some things:

  First, and ideal pipeline might look something like Fig. 1.2. Notice that, in this figure, all the pipeline stages are complete and full. Every instruction takes one clock and there are no delays. However, as the discussion above pointed out, the memory associated with a SwiC may not allow single clock access. It may be instead that you can only read every two clocks. In that case, what shall the pipeline look like? Should it look like Fig. 1.3, where instructions are held back until the pipeline is full, or should it look like Fig. 1.4, where each instruction is allowed to move through the pipeline independently? For better or worse, the Zip CPU allows instructions to move through the pipeline independently.

  One approach to avoiding stalls is to use a branch delay slot, such as is shown in Fig. 1.5. In this figure, instructions `BR` (a branch), `BD` (a branch delay instruction), are followed by

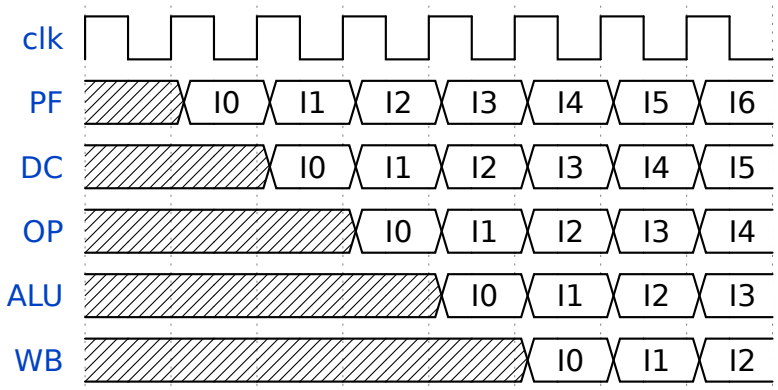

Figure 1.2: An Ideal Pipeline: One instruction per clock cycle

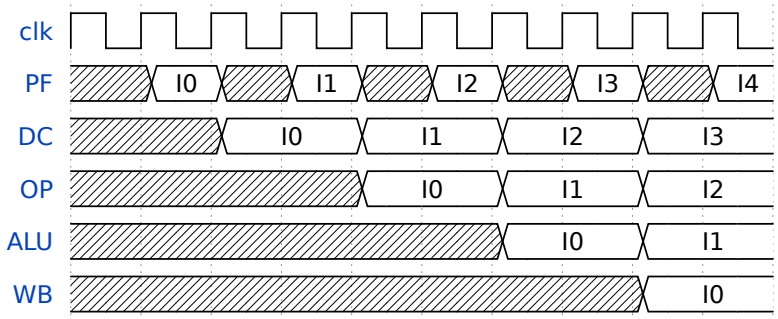

Figure 1.3: Instructions wait for each other

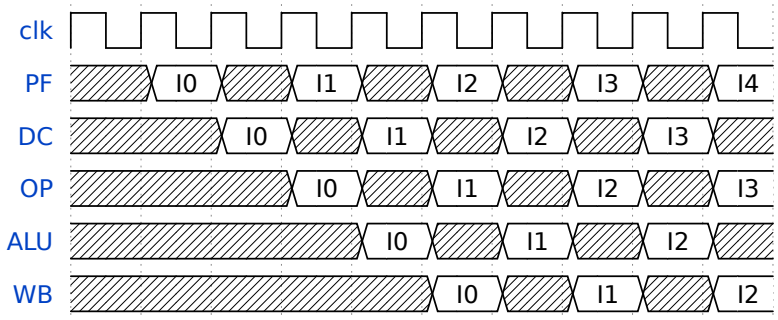

Figure 1.4: Instructions proceed independently

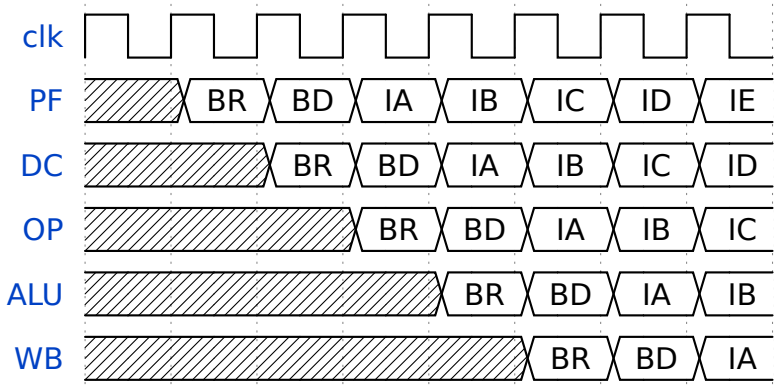

Figure 1.5: A typical branch delay slot approach

Figure 1.6: The branch delay slot breaks with a slow memory

instructions after the branch: `IA`, `IB`, etc. Since it takes a processor a clock cycle to execute a branch, the delay slot allows the processor to do something useful in that branch. The problem the Zip CPU has with this approach is, what happens when the pipeline looks like Fig. 1.6? In this case, the branch delay slot never gets filled in the first place, and so the pipeline squashes it before it gets executed. If not that, then what happens when handling interrupts or debug stepping: when has the CPU finished an instruction? When the `BR` instruction has finished, or must `BD` follow every `BR`? and, again, what if the pipeline isn't full? These thoughts killed any hopes of doing delayed branching.

So I switched to a model of discrete execution: Once an instruction enters into either the ALU or memory unit, the instruction is guaranteed to complete. If the logic recognizes a branch or a condition that would render the instruction entering into this stage possibly inappropriate (i.e. a conditional branch preceding a store instruction for example), then the pipeline stalls for one cycle until the conditional branch completes. Then, if it generates a new PC address, the stages preceding are all wiped clean.

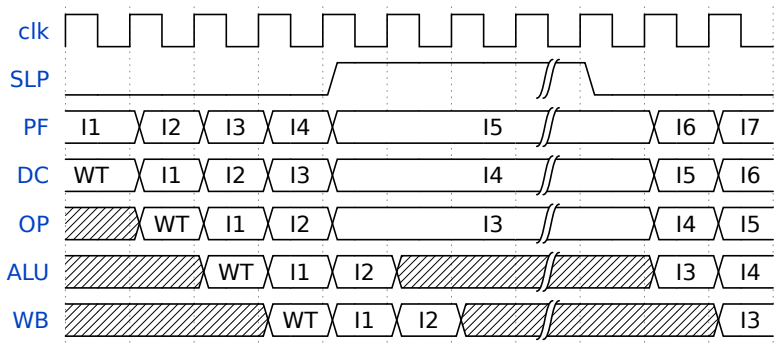

Figure 1.7: How the CPU halts when sleeping

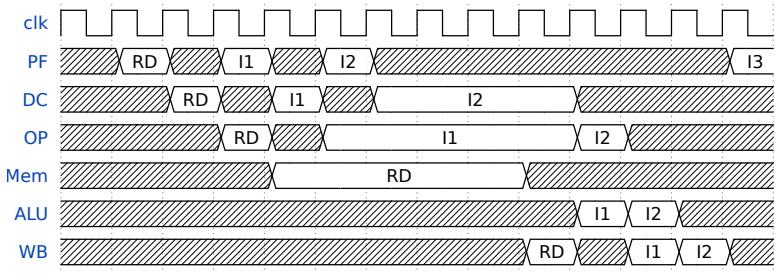

Figure 1.8: Instructions can stack up behind a stalled instruction

This model, however, generated too many pipeline stalls, so the discrete execution model was modified to allow instructions to go through the ALU unit and be canceled before writeback. This removed the stall associated with ALU instructions before untaken branches.

The discrete execution model allows such things as sleeping, as outlined in Fig. 1.7. If the CPU is put to "sleep," the ALU and memory stages stall and back up everything before them. Likewise, anything that has entered the ALU or memory stage when the CPU is placed to sleep continues to completion. To handle this logic, each pipeline stage has three control signals: a valid signal, a stall signal, and a clock enable signal. In general, a stage stalls if it's contents are valid and the next step is stalled. This allows the pipeline to fill any time a later stage stalls, as illustrated in Fig. 1.8. However, if a pipeline hazard is detected, a stage can stall in order to prevent the previous from moving forward.

This approach is also different from other pipeline approaches. Instead of keeping the entire pipeline filled, each stage is treated independently. Therefore, individual stages may move forward as long as the subsequent stage is available, regardless of whether the stage behind it is filled.

With that introduction out of the way, let's move on to the instruction set.

2.

CPU Architecture

The Zip CPU supports a set of two operand instructions, where the second operand (always a register) is the result. The only exception is the store instruction, where the first operand (always a register) is the source of the data to be stored.

2.1 Simplified Bus

The bus architecture of the Zip CPU is that of a simplified WISHBONE bus. It has been simplified in this fashion: all operations are 32–bit operations. The bus is neither little endian nor big endian. For this reason, all words are 32–bits. All instructions are also 32–bits wide. Everything has been built around the 32–bit word.

2.2 Register Set

The Zip CPU supports two sets of sixteen 32-bit registers, a supervisor and a user set as shown in Fig. 2.1. The supervisor set is used in interrupt mode when interrupts are disabled, whereas the user set is used otherwise. Of this register set, the Program Counter (PC) is register 15, whereas the status register (SR) or condition code register (CC) is register 14. By convention, the stack pointer will be register 13 and noted as (SP)–although there is nothing special about this register other than this convention. Also by convention register 12 will point to a global offset table, and may be abbreviated as the (GBL) register. The CPU can access both register sets via move instructions from the supervisor state, whereas the user state can only access the user registers.

The status register is special, and bears further mention. As shown in Fig. 2.1, the lower 11 bits of the status register form a set of CPU state and condition codes. Writes to other bits of this register are preserved.

Of the condition codes, the bottom four bits are the current flags: Zero (Z), Carry (C), Negative (N), and Overflow (V). On those instructions that set the flags, these flags will be set based upon the output of the instruction. If the result is zero, the Z flag will be set. If the high order bit is set, the N flag will be set. If the instruction caused a bit to fall off the end, the carry bit will be set. Finally, if the instruction causes a signed integer overflow, the V flag will be set afterwards.

The next bit is a sleep bit. Set this bit to one to disable instruction execution and place the CPU to sleep, or to zero to keep the pipeline running. Setting this bit will cause the CPU to wait for an interrupt (if interrupts are enabled), or to completely halt (if interrupts are disabled). In order to prevent users from halting the CPU, only the supervisor is allowed to both put the CPU to sleep and disable interrupts. Any user attempt to do so will simply result in a switch to supervisor mode.

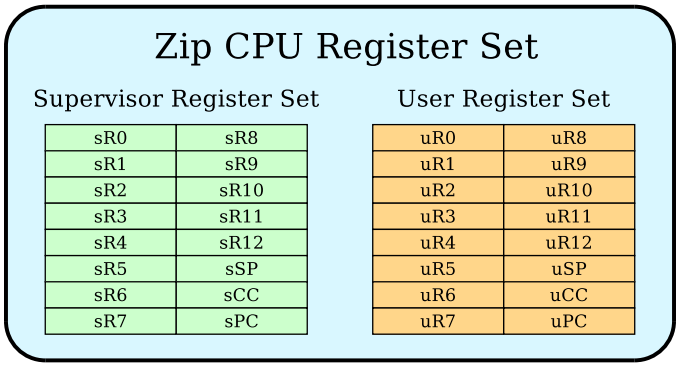

Figure 2.1: Zip CPU Register File

Bit #	Access	Description
31...13	R/W	Reserved for future uses
12	R	(Reserved for) Floating Point Exception
11	R	Division by Zero Exception
10	R	Bus-Error Flag
9	R	Trap, or user interrupt, Flag. Cleared on return to userspace.
8	R	Illegal Instruction Flag
7	R/W	Break–Enable
6	R/W	Step
5	R/W	Global Interrupt Enable (GIE)
4	R/W	Sleep. When GIE is also set, the CPU waits for an interrupt.
3	R/W	Overflow
2	R/W	Negative. The sign bit was set as a result of the last ALU instruction.
1	R/W	Carry
0	R/W	Zero. The last ALU operation produced a zero.

Table 2.1: Condition Code Register Bit Assignment

The sixth bit is a global interrupt enable bit (GIE). When this sixth bit is a '1' interrupts will be enabled, else disabled. When interrupts are disabled, the CPU will be in supervisor mode, otherwise it is in user mode. Thus, to execute a context switch, one only need enable or disable interrupts. (When an interrupt line goes high, interrupts will automatically be disabled, as the CPU goes and deals with its context switch.) Special logic has been added to keep the user mode from setting the sleep register and clearing the GIE register at the same time, with clearing the GIE register taking precedence.

The seventh bit is a step bit. This bit can be set from supervisor mode only. After setting this bit, should the supervisor mode process switch to user mode, it would then accomplish one instruction in user mode before returning to supervisor mode. Then, upon return to supervisor mode, this bit will be automatically cleared. This bit has no effect on the CPU while in supervisor mode.

This functionality was added to enable a userspace debugger functionality on a user process, working through supervisor mode of course.

The eighth bit is a break enable bit. This controls whether a break instruction in user mode will halt the processor for an external debugger (break enabled), or whether the break instruction will simply send send the CPU into interrupt mode. Encountering a break in supervisor mode will halt the CPU independent of the break enable bit. This bit can only be set within supervisor mode.

This functionality was added to enable an external debugger to set and manage breakpoints.

The ninth bit is an illegal instruction bit. When the CPU tries to execute either a non-existant instruction, or an instruction from an address that produces a bus error, the CPU will (if implemented) switch to supervisor mode while setting this bit. The bit will automatically be cleared upon any return to user mode.

The tenth bit is a trap bit. It is set whenever the user requests a soft interrupt, and cleared on any return to userspace command. This allows the supervisor, in supervisor mode, to determine whether it got to supervisor mode from a trap or from an external interrupt or both.

2.3 Instruction Format

All Zip CPU instructions fit in one of the formats shown in Fig. 2.2. 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, or DR. The Load 23–bit signed immediate instruction is different in that it requires no conditions, and uses only a 4-bit opcode.

This is actually a second version of instruction set definition, given certain lessons learned. For example, the original instruction set had the following problems:

1. No opcodes were available for divide or floating point extensions to be made available. Although there was space in the instruction set to add these types of instructions, this instruction space was going to require extra logic to use.

2. The carveouts for instructions such as NOOP and LDIHI/LDILO required extra logic to process.

3. The instruction set wasn't very compact. One bus operation was required for every instruction.

This second version was designed with two criteria. The first was that the new instruction set needed to be compatible, at the assembly language level, with the previous instruction set. Thus, it must be able to support all of the previous menumonics and more. This was achieved with the

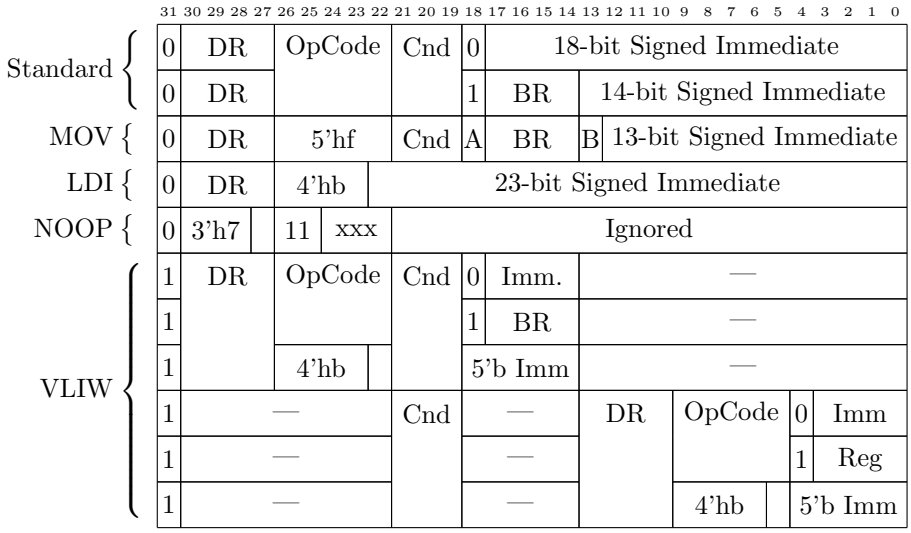

Figure 2.2: Zip Instruction Set Format

sole exception that instruction immediates are generally two bits shorter than before. (One bit was lost to the VLIW bit in front, another from changing from 4–bit to 5–bit opcodes.) Second, the new instruction set needed to be a drop–in replacement for the decoder, modifying nothing else. This was almost achieved, save for two issues: the ALU unit needed to be replaced since the OpCodes were reordered, and some condition code logic needed to be adjusted since the condition codes were renumbered as well. In the end, maximum reuse of the existing RTL (Verilog) code was achieved in this upgrade.

As of this second version of the Zip CPU instruction set, the Zip CPU also supports a very long instruction word (VLIW) set of instructions. These instruction formats pack two instructions into a single instuction word, trading immediate instruction space to do this, but in just about all other respects these are identical to two standard instructions. Other than instruction format, the only basic difference is that the CPU will not switch to interrupt mode in between the two instructions. Likewise a new job given to the assembler is that of automatically packing as many instructions as possible into the VLIW format. Where necessary to place both VLIW instructions on the same line, they will be separated by a vertical bar.

2.4 Instruction OpCodes

With a 5–bit opcode field, there are 32–possible instructions as shown in Tbl. 2.2. Of these opcodes, the `BREV` and `POPC` are experimental, and may be replaced later, and two floating point instruction opcodes are reserved for future use.

OpCode		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	LDIHI	Load Immediate High	N
5'h09	LDILO	Load Immediate Low	
5'h0a	MPYU	Unsigned 16–bit Multiply	Y
5'h0b	MPYS	Signed 16–bit Multiply	
5'h0c	BREV	Bit Reverse	
5'h0d	POPC	Population Count	
5'h0e	ROL	Rotate left	
5'h0f	MOV	Move register	N
5'h10	CMP	Compare	Y
5'h11	TST	Test (AND w/o setting result)	
5'h12	LOD	Load from memory	N
5'h13	STO	Store a register into memory	
5'h14	DIVU	Divide, unsigned	Y
5'h15	DIVS	Divide, signed	
5'h16/7	LDI	Load 23–bit signed immediate	N
5'h18	FPADD	Floating point add	Y
5'h19	FPSUB	Floating point subtract	
5'h1a	FPMPY	Floating point multiply	
5'h1b	FPDIV	Floating point divide	
5'h1c	FPCVT	Convert integer to floating point	
5'h1d	FPINT	Convert to integer	
5'h1e		*Reserved for future use*	
5'h1f		*Reserved for future use*	

Table 2.2: Zip CPU OpCodes

Code	Mneumonic	Condition
3'h0	None	Always execute the instruction
3'h1	`.LT`	Less than ('N' set)
3'h2	`.Z`	Only execute when 'Z' is set
3'h3	`.NZ`	Only execute when 'Z' is not set
3'h4	`.GT`	Greater than ('N' not set, 'Z' not set)
3'h5	`.GE`	Greater than or equal ('N' not set, 'Z' irrelevant)
3'h6	`.C`	Carry set
3'h7	`.V`	Overflow set

Table 2.3: 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.
;If R0 = 1, conditions are set based upon R1-2.
;Now do something based upon the conjunction of both conditions.
;While we use the example of a STO, it could be any instruction.
STO.Z R0,(R2)
```

Table 2.4: An example of a double conditional

2.5 Conditional Instructions

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

From the four condition code flags, eight conditions are defined for standard instructions. These are shown in Tbl. 2.3. There is no condition code for less than or equal, not C or not V—there just wasn't enough space in 3–bits. Conditioning on a non–supported condition is still possible, but it will take an extra instruction and a pipeline stall. (Ex: *(Stall)*; `TST $4,CC; STO.NZ R0,(R1)`) As an alternative, it is often possible to reverse the condition, and thus recovering those extra two clocks. Thus instead of `CMP Rx,Ry; BNV label` you can issue a `CMP Ry,Rx; BV label`.

Conditionally executed instructions will not further adjust the condition codes, with the exception of `CMP` and `TST` instructions. Conditional `CMP` or `TST` instructions will adjust conditions whenever they are executed. In this way, multiple conditions may be evaluated without branches. For example, to do something if `R0` is one and `R1` is two, one might try code such as Tbl. 2.4.

In the case of VLIW instructions, only four conditions are defined as shown in Tbl. 2.5. Further, the first bit is given a special meaning. If the first bit is set, the conditions apply to the second half of the instruction, otherwise the conditions will only apply to the first half of a conditional instruction.

Code	Mneumonic	Condition
2'h0	None	Always execute the instruction
2'h1	`.LT`	Less than ('N' set)
2'h2	`.Z`	Only execute when 'Z' is set
2'h3	`.NZ`	Only execute when 'Z' is not set

Table 2.5: VLIW 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 2.6: Bit allocation for Operand B

2.6 Operand B

Many instruction forms have a 19-bit source "Operand B" associated with them. This "Operand B" is shown in Fig. 2.2 as part of the standard instructions. This Operand B is either equal to a register plus a 14–bit signed immediate offset, or an 18–bit signed immediate offset by itself. This value is encoded as shown in Tbl. 2.6.

Fourteen and eighteen bit immediate values don't make sense for all instructions. For example, what is the point of an 18–bit immediate when executing a 16–bit multiply? Or a 16–bit load–immediate? In these cases, the extra bits are simply ignored.

VLIW instructions still use the same operand B, only there was no room for any instruction plus immediate addressing. Therefore, VLIW instructions have either a register or a 4–bit signed immediate as their operand B. The only exception is the load immediate instruction, which permits a 5–bit signed operand B.[1]

2.7 Address Modes

The Zip CPU supports two addressing modes: register plus immediate, and immediate address. Addresses are therefore encoded in the same fashion as Operand B's, shown above. Practically, the VLIW instruction set only offers register addressing, necessitating a non–VLIW instruction for most memory operations.

A lot of long hard thought was put into whether to allow pre/post increment and decrement addressing modes. Finding no way to use these operators without taking two or more clocks per instruction,[2] these addressing modes have been removed from the realm of possibilities. This means that the Zip CPU has no native way of executing push, pop, return, or jump to subroutine operations. Each of these instructions can be emulated with a set of instructions from the existing set.

[1] Although the space exists to extend this VLIW load immediate instruction to six bits, the 5–bit limit was chosen to simplify the disassembler. This may change in the future.

[2] The two clocks figure comes from the design of the register set, allowing only one write per clock. That write is either from the memory unit or the ALU, but never both.

2.8 Move Operands

The previous set of operands would be perfect and complete, save only that the CPU needs access to non–supervisory registers while in supervisory mode. Therefore, the MOV instruction is special and offers access to these registers . . . when in supervisory mode. To keep the compiler simple, the extra bits are ignored in non-supervisory mode (as though they didn't exist), rather than being mapped to new instructions or additional capabilities. The bits indicating which register set each register lies within are the A-User, marked 'A' in Fig. 2.2, and B-User bits, marked as 'B'. When set to a one, these refer to a user mode register. When set to a zero, these refer to a register in the current mode, whether user or supervisor. Further, because a load immediate instruction exists, there is no move capability between an immediate and a register: all moves come from either a register or a register plus an offset.

This actually leads to a bit of a problem: since the `MOV` instruction encodes which register set each register is coming from or moving to, how shall a compiler or assembler know how to compile a MOV instruction without knowing the mode of the CPU at the time? For this reason, the compiler will assume all MOV registers are supervisor registers, and display them as normal. Anything with the user bit set will be treated as a user register and displayed special. Since the CPU quietly ignores the supervisor bits while in user mode, anything marked as a user register will always be specific.

2.9 Multiply Operations

The Zip CPU supports two Multiply operations, a 16x16 bit signed multiply (`MPYS`) and a 16x16 bit unsigned multiply (`MPYU`). A 32–bit multiply, should it be desired, needs to be created via software from this 16x16 bit multiply.

2.10 Divide Unit

The Zip CPU also has a divide unit which can be built alongside the ALU. This divide unit provides the Zip CPU with its first 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.

The Divide is also a multi–clock instruction. While the divide is running, the ALU, memory unit, and floating point unit (if installed) will be idle. Once the divide completes, other units may continue.

Of course, divides can have errors: division by zero. In the case of division by zero, an exception will be caused that will send the CPU either from user mode to supervisor mode, or halt the CPU if it is already in supervisor mode.

2.11 NOOP, BREAK, and Bus Lock Instruction

Three instructions are not listed in the opcode list in Tbl. 2.2, yet fit in the NOOP type instruction format of Fig. 2.2. These are the `NOOP`, `Break`, and bus `LOCK` instructions. These are encoded according to Fig. 2.3, and have the following meanings:

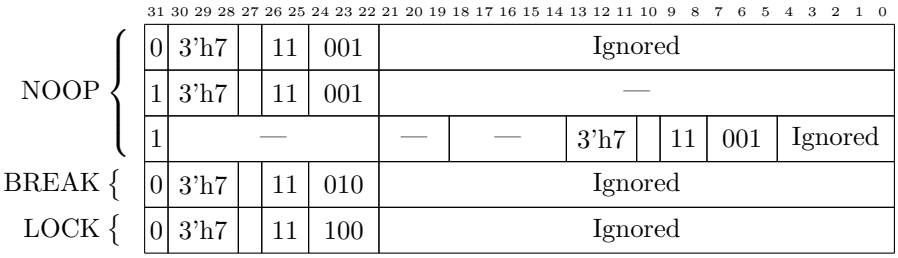

Figure 2.3: NOOP/Break/LOCK Instruction Format

The `NOOP` instruction is just that: an instruction that does not perform any operation. While many other instructions, such as a move from a register to itself, could also fit these roles, only the NOOP instruction guarantees that it will not stall waiting for a register to be available. For this reason, it gets its own place in the instruction set.

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.

Finally, the `LOCK` instruction was added in order to make a test and set multi–CPU operation possible. Following a LOCK instruction, the next two instructions, if they are memory LOD/STO instructions, will execute without dropping the wishbone `CYC` line between the instructions. Thus a `LOCK` followed by `LOD (Rx),Ry` and a `STO Rz,(Rx)`, where Rz is initially set, can be used to set an address while guaranteeing that Ry was the value before setting the address to Rz. This is a useful instruction while trying to achieve concurrency among multiple CPU's.

2.12 Floating Point

Although the Zip CPU does not (yet) have a floating point unit, the current instruction set offers eight opcodes for floating point operations, and treats floating point exceptions like divide by zero errors. Once this unit is built and integrated together with the rest of the CPU, the Zip CPU will support 32–bit floating point instructions natively. Any 64–bit floating point instructions will still need to be emulated in software.

2.13 Derived Instructions

The Zip CPU supports many other common instructions, but not all of them are single cycle instructions. The derived instruction tables, Tbls. 2.7, 2.8, 2.9 and 2.10, help to capture some of how these other instructions may be implemented on the Zip CPU. Many of these instructions will have assembly equivalents, such as the branch instructions, to facilitate working with the CPU.

Mapped	Actual	Notes
`ABS Rx`	`TST -1,Rx` `NEG.LT Rx`	Absolute value, depends upon derived NEG.

`ADD Ra,Rx`
`ADDC Rb,Ry`
	`Add Ra,Rx`
	`ADD.C $1,Ry`
	`Add Rb,Ry`
		Add with carry

`BRA.Cond +/-$Addr`	`MOV.cond $Addr+PC,PC`	Branch or jump on condition. Works for 13–bit signed address offsets.
`BRA.Cond +/-$Addr`	`LDI $Addr,Rx` `ADD.cond Rx,PC`	Branch/jump on condition. Works for 23 bit address offsets, but costs a register, an extra instruction, and sets the flags.
`BNC PC+$Addr`	`Test $Carry,CC` `MOV.Z PC+$Addr,PC`	Example of a branch on an unsupported condition, in this case a branch on not carry
`BUSY`	`MOV $-1(PC),PC`	Execute an infinite loop
`CLRF.NZ Rx`	`XOR.NZ Rx,Rx`	Clear Rx, and flags, if the Z-bit is not set
`CLR Rx`	`LDI $0,Rx`	Clears Rx, leaves flags untouched. This instruction cannot be conditional.
`EXCH.W Rx`	`ROL $16,Rx`	Exchanges the top and bottom 16'bit words of Rx
`HALT`	`Or $SLEEP,CC`	This only works when issued in interrupt/supervisor mode. In user mode this is simply a wait until interrupt instruction.
`INT`	`LDI $0,CC`	This is also known as a trap instruction
`IRET`	`OR $GIE,CC`	Also known as an RTU instruction (Return to Userspace)
`JMP R6+$Addr`	`MOV $Addr(R6),PC`	
`LJMP $Addr`	`LOD (PC),PC` *`Address`*	Although this only works for an unconditional jump, and it only works in a Von Neumann architecture, this instruction combination makes for a nice combination that can be adjusted by a linker at a later time.
`JSR PC+$Addr`	`MOV $1+PC,R0` `MOV $addr+PC,PC`	This is similar to the jump and link instructions from other architectures, save only that it requires a specific link instruction, also known as the `MOV` instruction on the left.

Table 2.7: Derived Instructions

Mapped	Actual	Notes
`LDI.l $val,Rx`	`LDIHI ($val>>16)&0x0ffff, Rx` `LDILO ($val&0x0ffff),Rx`	Sadly, there's not enough instruction space to load a complete immediate value into any register. Therefore, fully loading any register takes two cycles. The LDIHI (load immediate high) and LDILO (load immediate low) instructions have been created to facilitate this. This is also the appropriate means for setting a register value to an arbitrary 32–bit value in a post–assembly link operation.

`LOD.b $addr,Rx`	`LDI $addr,Ra`
	`LDI $addr,Rb`
	`LSR $2,Ra`
	`AND $3,Rb`
	`LOD (Ra),Rx`
	`LSL $3,Rb`
	`SUB $32,Rb`
	`ROL Rb,Rx`
	`AND $0ffh,Rx`
		This CPU is designed for 32'bit word length instructions. Byte addressing is not supported by the CPU or the bus, so it therefore takes more work to do.
		Note also that in this example, $Addr is a byte-wise address, where all other addresses in this document are 32-bit wordlength addresses. For this reason, we needed to drop the bottom two bits. This also limits the address space of character accesses using this method from 16 MB down to 4MB.

`LSL $1,Rx`
`LSLC $1,Ry`
	`LSL $1,Ry`
	`LSL $1,Rx`
	`OR.C $1,Ry`
		Logical shift left with carry. Note that the instruction order is now backwards, to keep the conditions valid. That is, LSL sets the carry flag, so if we did this the other way with Rx before Ry, then the condition flag wouldn't have been right for an OR correction at the end.

`LSR $1,Rx`
`LSRC $1,Ry`
	`CLR Rz`
	`LSR $1,Ry`
	`LDIHI.C $8000h,Rz`
	`LSR $1,Rx`
	`OR Rz,Rx`
		Logical shift right with carry

`NEG Rx`	`XOR $-1,Rx` `ADD $1,Rx`	
`NEG.C Rx`	`MOV.C $-1+Rx,Rx` `XOR.C $-1,Rx`	
`NOOP`	`NOOP`	While there are many operations that do nothing, such as MOV Rx,Rx, or OR $0,Rx, these operations have consequences in that they might stall the bus if Rx isn't ready yet. For this reason, we have a dedicated NOOP instruction.
`NOT Rx`	`XOR $-1,Rx`	
`POP Rx`	`LOD $(SP),Rx` `ADD $1,SP`	

Table 2.8: Derived Instructions, continued

PUSH Rx	SUB $1,SP STO Rx,$(SP)	Note that for pipelined operation, it helps to coalesce all the SUB's into one command, and place the STO's right after each other. Further, to avoid a pipeline stall, the immediate value for the store must be zero.
PUSH Rx-Ry	SUB $$n$,SP STO Rx,$(SP) ... STO Ry,$$(n-1)$(SP)	Multiple pushes at once only need the single subtract from the stack pointer. This derived instruction is analogous to a similar one on the Motoroloa 68k architecture, although the Zip Assembler does not support this instruction (yet). This instruction also supports pipelined memory access.
RESET	STO $1,$watchdog(R12) NOOP NOOP	This depends upon the peripheral base address being preloaded into R12. Another opportunity might be to jump to the reset address from within supervisor mode.
RET	MOV R0,PC	This depends upon the form of the JSR given on the previous page that stores the return address into R0.
STEP Rr,Rt	LSR $1,Rr XOR.C Rt,Rr	Step a Galois implementation of a Linear Feedback Shift Register, Rr, using taps Rt

STO.b Rx,$addr	LDI $addr,Ra
	LDI $addr,Rb
	LSR $2,Ra
	AND $3,Rb
	SUB $32,Rb
	LOD (Ra),Ry
	AND $0ffh,Rx
	AND ~$0ffh,Ry
	ROL Rb,Rx
	OR Rx,Ry
	STO Ry,(Ra)
		This CPU and it's bus are *not* optimized for byte-wise operations.
		Note that in this example, $addr is a byte-wise address, whereas in all of our other examples it is a 32-bit word address. This also limits the address space of character accesses from 16 MB down to 4MB.F Further, this instruction implies a byte ordering, such as big or little endian.

SWAP Rx,Ry	XOR Ry,Rx XOR Rx,Ry XOR Ry,Rx	While no extra registers are needed, this example does take 3-clocks.

Table 2.9: Derived Instructions, continued

`TRAP #X`	`LDI $x,R0` `AND ~$GIE,CC`	This works because whenever a user lowers the $GIE flag, it sets a TRAP bit within the CC 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 an LDILO and LDIHI pair, but the effect would be the same.

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

`TST Rx`	`TST $-1,Rx`	Set the condition codes based upon Rx. Could also do a CMP $0,Rx, ADD $0,Rx, SUB $0,Rx, etc, AND $-1,Rx, etc. The TST and CMP approaches won't stall future pipeline stages looking for the value of Rx. (Future versions of the assembler may shorten this to a `TST Rx` instruction.)
`WAIT`	`Or $GIE | $SLEEP,CC`	Wait until the next interrupt, then jump to supervisor/interrupt mode.

Table 2.10: Derived Instructions, continued

2.14 Interrupt Handling

The Zip CPU does not maintain any interrupt vector tables. If an interrupt takes place, the CPU simply switches to interrupt mode. The supervisor code continues in this interrupt mode from where it left off before, after executing a return to userspace `RTU` instruction.

At this point, the supervisor code needs to determine first whether an interrupt has occurred, and then whether it is in interrupt mode due to an exception and handle each case appropriately.

2.15 Pipeline Stages

As mentioned in the introduction, and highlighted in Fig. 1.1, the Zip CPU supports a five stage pipeline.

1. **Prefetch**: Reads instruction from memory and into a cache, if so configured. This stage is actually pipelined itself, and so it will stall if the PC ever changes. Stalls are also created here if the instruction isn't in the prefetch cache.

   The Zip CPU supports one of three prefetch methods, depending upon a flag set at build time within the `cpudefs.v` file. The simplest is a non–cached implementation of a prefetch. This implementation is fairly small, and ideal for users of the Zip CPU who need the extra space on the FPGA fabric. However, because this non–cached version has no cache, the maximum number of instructions per clock is limited to about one per five.

   The second prefetch module is a pipelined prefetch with a cache. This module tries to keep the instruction address within a window of valid instruction addresses. While effective, it is not a traditional cache implementation. One unique feature of this cache implementation, however, is that it can be cleared in a single clock. A disappointing feature, though, was that it needs an extra internal pipeline stage to be implemented.

   The third prefetch and cache module implements a more traditional cache. While the resulting code tends to be twice as fast as the pipelined cache architecture, this implementation uses a large amount of distributed FPGA RAM to be successful. This then inflates the Zip CPU's FPGA usage statistics.

2. **Decode**: Decodes an instruction into OpCode, register(s) to read, and immediate offset. This stage also determines whether the flags will be set or whether the result will be written back.

3. **Read Operands**: Read registers and apply any immediate values to them. There is no means of detecting or flagging arithmetic overflow or carry when adding the immediate to the operand. This stage will stall if any source operand is pending.

4. Split into one of four tracks: An **ALU** track which will accomplish a simple instruction, the **MemOps** stage which handles `LOD` (load) and `STO` (store) instructions, the **divide** unit, and the **floating point** unit.

   - Loads will stall instructions in the decode stage until the entire pipeline until complete, lest a register be read in the read operands stage only to be updated unseen by the Load.
   - Condition codes are available upon completion of the ALU, divide, or FPU stage.

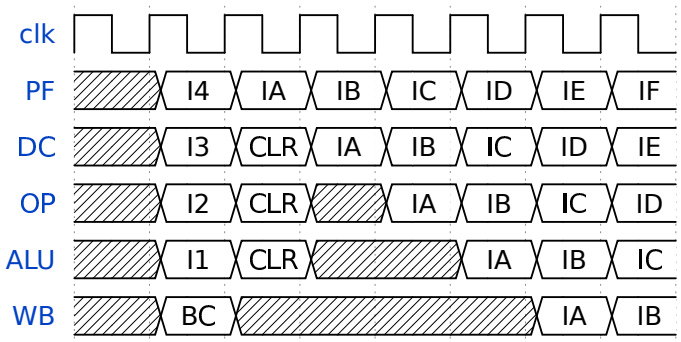

Figure 2.4: A conditional branch generates 4 stall cycles

- Issuing a non–pipelined memory instruction to the memory unit while the memory unit is busy will stall the entire pipeline.

5. **Write-Back**: Conditionally write back the result to the register set, applying the condition. This routine is quad-entrant: either the ALU, the memory, the divide, or the FPU may write back a register. The only design rule is that no more than a single register may be written back in any given clock.

The Zip CPU does not support out of order execution. Therefore, if the memory unit stalls, every other instruction stalls. The same is true for divide or floating point instructions–all other instructions will stall while waiting for these to complete. Memory stores, however, can take place concurrently with non–memory operations, although memory reads (loads) cannot.

2.16 Pipeline Stalls

The processing pipeline can and will stall for a variety of reasons. Some of these are obvious, some less so. These reasons are listed below:

- When the prefetch cache is exhausted

  This reason should be obvious. If the prefetch cache doesn't have the instruction in memory, the entire pipeline must stall until an instruction can be made ready. In the case of the `pipefetch` windowed approach to the prefetch cache, this means the pipeline will stall until enough of the prefetch cache is loaded to support the next instruction. In the case of the more traditional `pfcache` approach, the entire cache line must fill before instruction execution can continue.

- While waiting for the pipeline to load following any taken branch, jump, return from interrupt or switch to interrupt context (4 stall cycles)

  Fig. 2.4 illustrates the situation for a conditional branch. In this case, the branch instruction, `BC`, is nominally followed by instructions `I1` and so forth. However, since the branch is taken, the next instruction must be `IA`. Therefore, the pipeline needs to be cleared and reloaded.

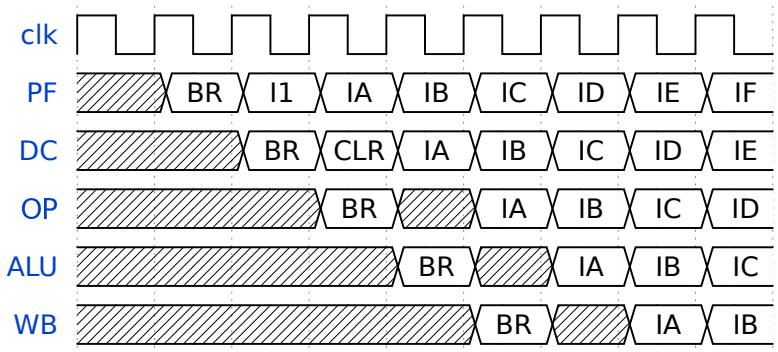

Figure 2.5: An expedited branch costs a single stall cycle

Given that there are five stages to the pipeline, that accounts for the four stalls. (Were the `pipefetch` cache chosen, there would be another stall internal to the `pipefetch` cache.)

The Zip CPU handles `MOV $X(PC),PC`, `ADD $X,PC`, and `LDI $X,PC` instructions specially, however. These instructions, when not conditioned on the flags, can execute with only a single stall cycle, such as is shown in Fig. 2.5.[3] In this example, `BR` is a branch always taken, `I1` is the instruction following the branch in memory, while `IA` is the first instruction at the branch address. (`CLR` denotes a clear–pipeline operation, and does not represent any instruction.)

- When reading from a prior register while also adding an immediate offset

  1. `OPCODE ?,RA`
  2. *(stall)*
  3. `OPCODE I+RA,RB`

  Since the addition of the immediate register within OpB decoding gets applied during the read operand stage so that it can be nicely settled before the ALU, any instruction that will write back an operand must be separated from the opcode that will read and apply an immediate offset by one instruction. The good news is that this stall can easily be mitigated by proper scheduling. That is, any instruction that does not add an immediate to `RA` may be scheduled into the stall slot.

  This is also the reason why, when setting up a stack frame, the top of the stack frame is used first: it eliminates this stall cycle. Hence, to save registers at the top of a procedure, one would write:

  1. `SUB 2,SP`
  2. `STO R1,(SP)`
  3. `STO R2,1(SP)`

[3] Note that when using the `pipefetch` cache, this requires an additional stall cycle due to that cache's implementation.

Had `R1` instead been stored at `1(SP)` as the top of the stack, there would've been an extra stall in setting up the stack frame.

- When reading from the CC register after setting the flags

  1. `ALUOP RA,RB` *; Ex: a compare opcode*
  2. *(stall)*
  3. `TST sys.ccv,CC`
  4. `BZ somewhere`

  The reason for this stall is simply performance: many of the flags are determined via combinatorial logic *during* the writeback cycle. Trying to then place these into the input for one of the operands for an ALU instruction during the same cycle created a time delay loop that would no longer execute in a single 100 MHz clock cycle. (The time delay of the multiply within the ALU wasn't helping either ...).

  This stall may be eliminated via proper scheduling, by placing an instruction that does not set flags in between the ALU operation and the instruction that references the CC register. For example, `MOV $addr+PC,uPC` followed by an `RTU` (`OR $GIE,CC`) instruction will not incur this stall, whereas an `OR $BREAKEN,CC` followed by an `OR $STEP,CC` will incur the stall, while a `LDI $BREAKEN|$STEP,CC` will not since it doesn't read the condition codes before executing.

- When waiting for a memory read operation to complete

  1. `LOD address,RA`
  2. *(multiple stalls, bus dependent, 4 clocks best)*
  3. `OPCODE I+RA,RB`

  Remember, the Zip CPU does not support out of order execution. Therefore, anytime the memory unit becomes busy both the memory unit and the ALU must stall until the memory unit is cleared. This is illustrated in Fig. 2.6, since it is especially true of a load instruction, which must still write its operand back to the register file. Further, note that on a pipelined memory operation, the instruction must stall in the decode operand stage, lest it try to read a result from the register file before the load result has been written to it. Finally, note that there is an extra stall at the end of the memory cycle, so that the memory unit will be idle for two clocks before an instruction will be accepted into the ALU. Store instructions are different, as shown in Fig. 2.7, since they can be busy with the bus without impacting later write back pipeline stages. Hence, only loads stall the pipeline.

  This, of course, also assumes that the memory being accessed is a single cycle memory and that there are no stalls to get to the memory. Slower memories, such as the Quad SPI flash, will take longer–perhaps even as long as forty clocks. During this time the CPU and the external bus will be busy, and unable to do anything else. Likewise, if it takes a couple of clock cycles for the bus to be free, as shown in both Figs. 2.6 and 2.7, there will be stalls.

- Memory operation followed by a memory operation

  1. `STO address,RA`

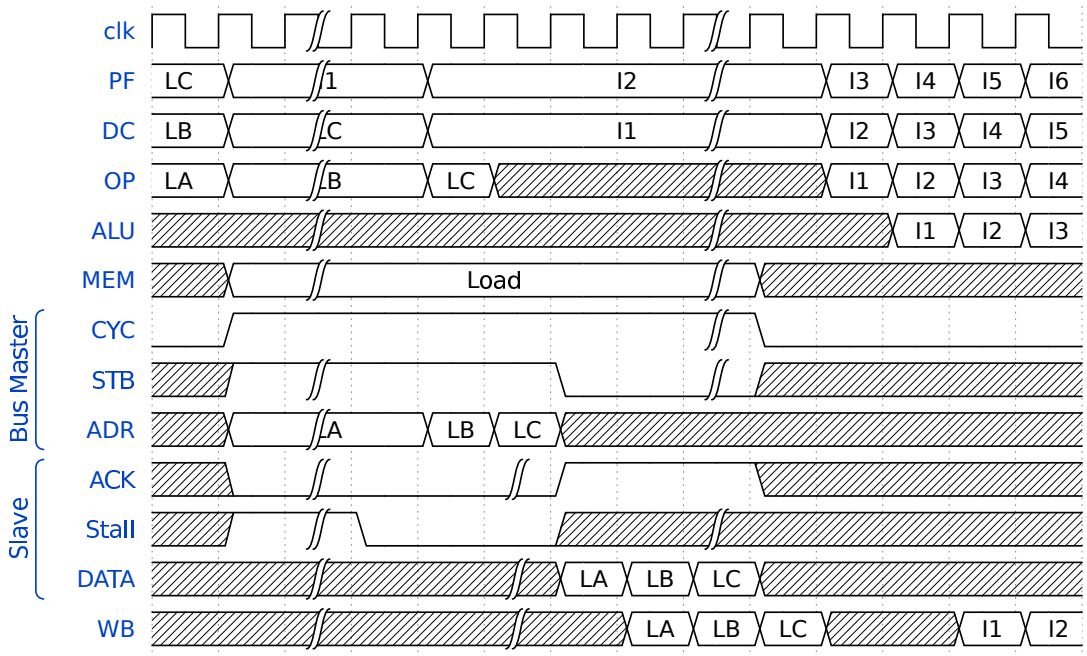

Figure 2.6: Pipeline handling of a load instruction

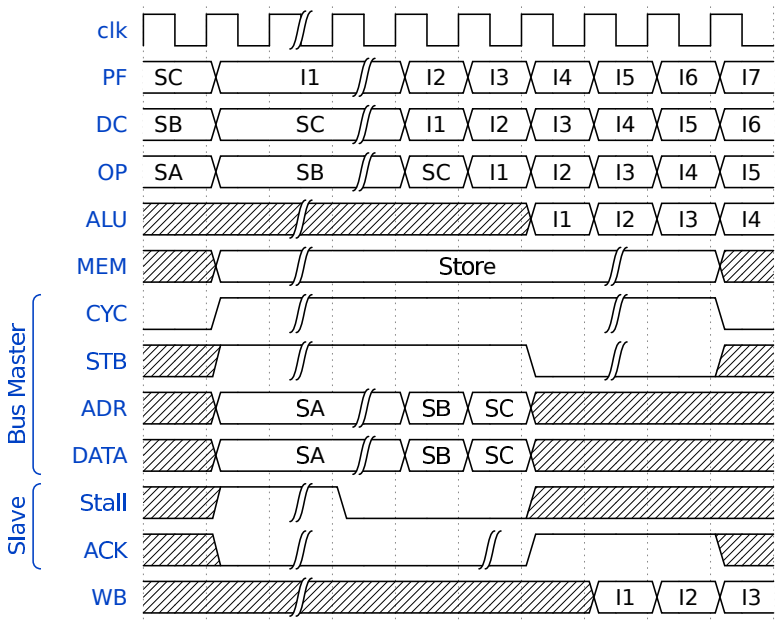

Figure 2.7: Pipeline handling of a store instruction

2. *(multiple stalls, bus dependent, 4 clocks best)*
3. `LOD address,RB`
4. *(multiple stalls, bus dependent, 4 clocks best)*

In this case, the LOD instruction cannot start until the STO is finished, as illustrated by Fig. 2.8. With proper scheduling, it is possible to do something in the ALU while the memory unit is busy with the STO instruction, but otherwise this pipeline will stall while waiting for it to complete before the load instruction can start.

The Zip CPU does have the capability of supporting pipelined memory access, but only under the following conditions: all accesses within the pipeline must all be reads or all be writes, all must use the same register for their address, and there can be no stalls or other instructions between pipelined memory access instructions. Further, the offset to memory must be increasing by one address each instruction. These conditions work well for saving or storing registers to the stack. Indeed, if you noticed, both Fig. 2.6 and Fig. 2.7 illustrated pipelined memory accesses.

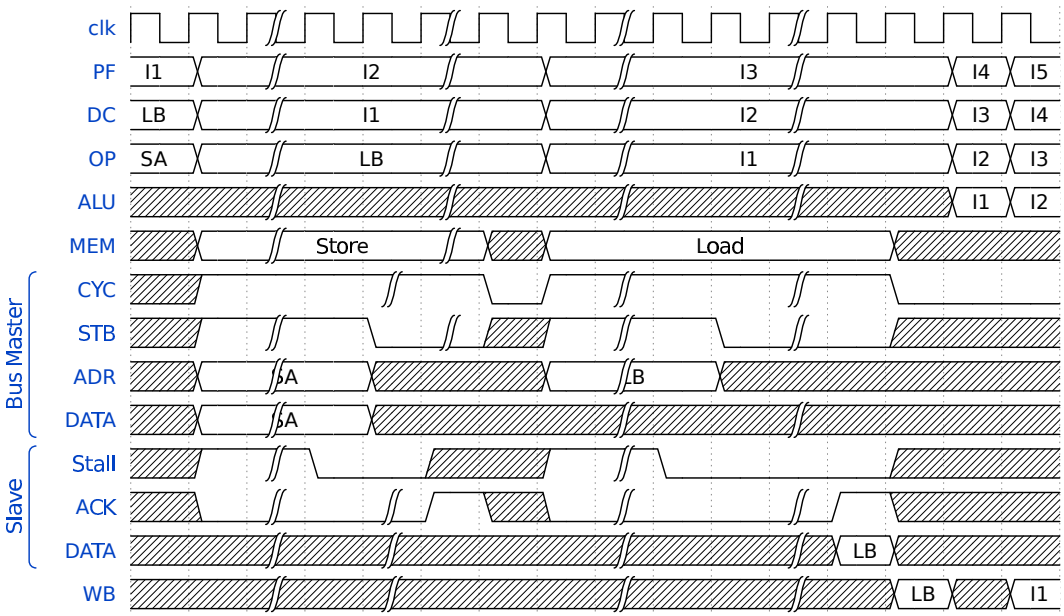

Figure 2.8: Pipeline handling of a store followed by a load instruction

3.

Peripherals

While the previous chapter describes a CPU in isolation, the Zip System includes a minimum set of peripherals as well. These peripherals are shown in Fig. 3.1 and described here. They are designed to make the Zip CPU more useful in an Embedded Operating System environment.

3.1 Interrupt Controller

Perhaps the most important peripheral within the Zip System is the interrupt controller. While the Zip CPU 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 Zip System interrupt controller module supports up to 15 interrupts, all controlled from one register. Bit 31 of the interrupt controller controls overall whether interrupts are enabled (1'b1) or disabled (1'b0). Bits 16–30 control whether individual interrupts are enabled (1'b1) or disabled (1'b0). Bit 15 is an indicator showing whether or not any interrupt is active, and bits 0–15 indicate whether or not an individual interrupt is active.

The interrupt controller has been designed so that bits can be controlled individually without having any knowledge of the rest of the controller setting. To enable an interrupt, write to the register with the high order global enable bit set and the respective interrupt enable bit set. No other bits will be affected. To disable an interrupt, write to the register with the high order global enable bit cleared and the respective interrupt enable bit set. To clear an interrupt, write a '1' to that interrupts status pin. Zero's written to the register have no affect, save that a zero written to the master enable will disable all interrupts.

As an example, suppose you wished to enable interrupt #4. You would then write to the register a `0x80100010` to enable interrupt #4 and to clear any past active state. When you later wish to disable this interrupt, you would write a `0x00100010` to the register. As before, this both disables the interrupt and clears the active indicator. This also has the side effect of disabling all interrupts, so a second write of `0x80000000` may be necessary to re-enable any other interrupts.

The Zip System currently hosts two interrupt controllers, a primary and a secondary. The primary interrupt controller has one (or more) interrupt line(s) which may come from an external interrupt source, and one interrupt line from the secondary controller. Other primary interrupts include the system timers, the jiffies interrupt, and the manual cache interrupt. The secondary interrupt controller maintains an interrupt state for all of the processor accounting counters.

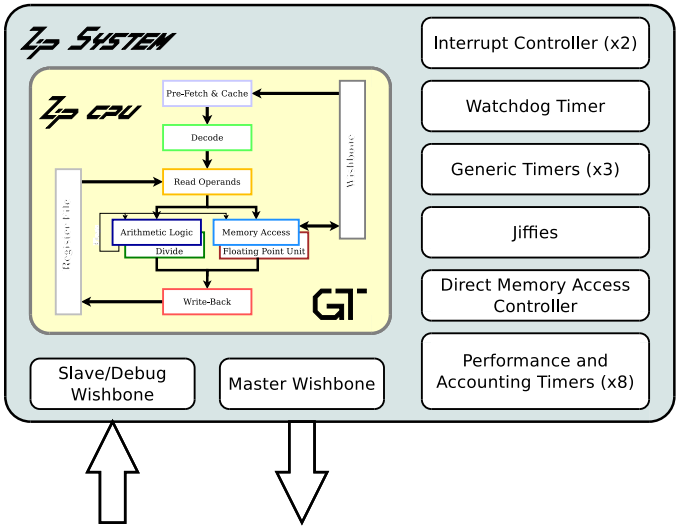

Figure 3.1: Zip System Peripherals

3.2 Counter

The Zip Counter is a very simple counter: it just counts. It 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.

Eight counters are implemented in the Zip System for process accounting. This may change in the future, as nothing as yet uses these counters.

3.3 Timer

The Zip Timer is also very simple: it simply counts down to zero. When it transitions from a one to a zero it creates an interrupt.

Writing any non-zero value to the timer starts the timer. If the high order bit is set when writing to the timer, the timer becomes an interval timer and reloads its last start time on any interrupt. Hence, to mark seconds, one might set the timer to 100 million (the number of clocks per second), and set the high bit. Ever after, the timer will interrupt the CPU once per second (assuming a 100 MHz clock). This reload capability also limits the maximum timer value to $2^{31}-1$ (about 21 seconds using a 100 MHz clock), rather than $2^{32}-1$.

3.4 Watchdog Timer

The watchdog timer is no different from any of the other timers, save for one critical difference: 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.

While the watchdog timer supports interval mode, it doesn't make as much sense as it did with the other timers.

3.5 Bus Watchdog

There is an additional watchdog timer on the Wishbone bus. 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 its port to find out which memory location created the problem.

Aside from its unusual configuration, the bus watchdog is just another implementation of the fundamental timer described above–stripped down for simplicity.

3.6 Jiffies

This 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, it will be enabled. (There is currently no way to disable a jiffie interrupt once set, other than to disable the interrupt line in the interrupt controller.) The processor may then place this sleep request into a list among 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 Jiffies register cannot be 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.

The purpose of this register is to support alarm times within a CPU. To set an alarm for a particular process N clocks in advance, read the current Jiffies value, and 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.

3.7 Direct Memory Access Controller

The Direct Memory Access (DMA) controller can be used to either move memory from one location to another, to read from a peripheral into memory, or to write from a peripheral into memory all without CPU intervention. Further, since the DMA controller can issue (and does issue) pipeline

wishbone accesses, any DMA memory move will by nature be faster than a corresponding program accomplishing the same move. To put this to numbers, it may take a program 18 clocks per word transferred, whereas this DMA controller can move one word in two clocks–provided it has bus access. (The CPU gets priority over the bus.)

When copying memory from one location to another, the DMA controller will copy in units of a given transfer length–up to 1024 words at a time. It will read that transfer length into its internal buffer, and then write to the destination address from that buffer.

When coupled with a peripheral, the DMA controller can be configured to start a memory copy when any interrupt line going high. Further, the controller can be configured to issue reads from (or to) the same address instead of incrementing the address at each clock. The DMA completes once the total number of items specified (not the transfer length) have been transferred.

In each case, once the transfer is complete and the DMA unit returns to idle, the DMA will issue an interrupt.

4.

Operation

The Zip CPU, and even the Zip System, is not a System on a Chip (SoC). It needs to be connected to its operational environment in order to be used. Specifically, some per system adjustments need to be made:

1. The Zip System depends upon an external 32-bit Wishbone bus. This must exist, and must be connected to the Zip CPU for it to work.
2. The Zip System needs to be told of its `RESET_ADDRESS`. This is the program counter of the first instruction following a reset.
3. To conserve logic, you'll want to set the `ADDRESS_WIDTH` parameter to the number of address bits on your wishbone bus.
4. Likewise, the `LGICACHE` parameter sets the number of bits in the instruction cache address. This means that the instruction cache will have $2^{\texttt{LGICACHE}}$ locations within it.
5. If you want the Zip System to start up on its own, you will need to set the `START_HALTED` parameter to zero. Otherwise, if you wish to manually start the CPU, that is if upon reset you want the CPU start start in its halted, reset state, then set this parameter to one. This latter configuration is useful for a CPU that should be idle (i.e. halted) until given an explicit instruction from somewhere else to start.
6. The third parameter to set is the number of interrupts you will be providing from external to the CPU. This can be anything from one to sixteen, but it cannot be zero. (Set this to 1 and wire the single interrupt line to a 1'b0 if you do not wish to support any external interrupts.)
7. Finally, you need to place into some wishbone accessible address, whether RAM or (more likely) ROM, the initial instructions for the CPU.

If you have enabled your CPU to start automatically, then upon power up the CPU will immediately start executing your instructions, starting at the given `RESET_ADDRESS`.

This is, however, not how I have used the Zip CPU. I have instead used the Zip CPU in a more controlled environment. For me, the CPU starts in a halted state, and waits to be told to start. Further, the RESET address is a location in RAM. After bringing up the board I am using, and further the bus that is on it, the RAM memory is then loaded externally with the program I wish the Zip System to run. Once the RAM is loaded, I release the CPU. The CPU then runs until either its halt condition or an exception occurrs in supervisor mode, at which point its task is complete.

Eventually, I intend to place an operating system onto the ZipSystem, I'm just not there yet.

The rest of this chapter examines some common programming models, and how they might be applied to the Zip System, and then finish with a couple of examples.

```
supervisor_idle:
    ; While not strictly required, the following move helps to
    ; ensure that the prefetch doesn't try to fetch an instruction
    ; outside of the CPU's address space when it switches to user
    ; mode.
    MOV supervisor_idle_continue,uPC
    ; Put the processor into user mode and to sleep in the same
    ; instruction.
    OR $SLEEP|$GIE,CC
supervisor_idle_continue:
    ; Now, if we haven't done this inline, we need to return
    ; to whatever function called us.
    RETN
```

Table 4.1: Executing an idle from supervisor mode

4.1 System High

The easiest and simplest way to run the Zip CPU is in the system high mode. In this mode, the CPU runs your program in supervisor mode from reboot 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 useful, and can handle many microcontroller 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.

The one thing that cannot be done in supervisor mode is a wait for interrupt instruction. This, however, is easily rectified by jumping to a user task within the supervisors memory space, such as Tbl. 4.1.

4.2 Traditional Interrupt Handling

Although the Zip CPU does not have a traditional interrupt architecture, it is possible to create the more traditional interrupt 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. 4.2.

We can work through the interrupt handling process by examining Tbl. 4.2. First, remember, the CPU is always running either the user or the supervisor context. Once the supervisor switches to user mode, control does not return until either an interrupt or a trap has taken place. (Okay, there's also the possibility of a bus error, or an illegal instruction such as an unimplemented floating point instruction—but for now we'll just focus on the trap instruction.) Therefore, if the trap bit isn't set, then we know an interrupt has taken place.

To process an interrupt, we steal the user's stack: the PC and CC registers are saved on the stack, as outlined in Tbl. 4.3. This is much cheaper than the full context swap of a preemptive multitasking kernel, but it also depends upon the ISR saving any state it uses. Further, if multiple ISR's get called at once, this looses its optimality property very quickly.

As Sec. 3.1 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 the top bit (master interrupt enable). This has the consequence of disabling any and all further interrupts, not just the ones that have tripped. Hence, upon completion, we re–enable the master interrupt bit again. Finally, we keep track of which interrupts have tripped.

Using the bit mask of interrupts that have tripped, we walk through all fifteen possible interrupts. If there is an ISR installed, we acknowledge and reset the interrupt within the PIC, and then call the ISR. The ISR, however, cannot re–enable its interrupt without re-enabling the master interrupt bit. Thus, to keep things simple, when the ISR is finished it places its interrupt mask back into R0, or clears R0. This tells the supervisor mode process which interrupts to re–enable. Any other registers that the ISR uses must be saved and restored. (This is only truly optimal if only a single ISR is called.) As a final instruction, the ISR clears the GIE bit executing a user trap. (Remember, the Zip CPU has no `RETI` instruction to restore the stack and return to userland. It needs to go through the supervisor mode to get there.)

Then, once all interrupts are handled, the user context is restored in a fashion similar to Tbl. 4.4. Again, this is short and sweet simply because any other registers that needed saving were saved within the ISR.

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

4.3 Example: Idle Task

One task every operating system needs is the idle task, the task that takes place when nothing else can run. On the Zip CPU, this task is quite simple, and it is shown in assemble 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. Sure, there may be some more settling, the pipe cache continue to read until full, other instructions may issue until the pipeline fills, but then everything will stall. 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.

```
while(true) {
    rtu();
    if (trap) { // Here, we allow users to install ISRs, or
        // whatever else they may wish to do in supervisor mode.
    } else {
        volatile int *pic = PIC_ADDRESS;

        // Save the user context before running any ISRs. This could easily be
        // implemented as an inline assembly routine or macro
        SAVE_PARTIAL_CONTEXT;
        // 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 = *pic;
        // Turn off all active interrupts
        // Globally disable interrupt generation in the process
        int active = (picv >> 16) & picv & 0x07fff;
        *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])) {
                mov(isr_table[i],uPC);
                // Acknowledge this particular interrupt. While we could acknowledge all
                // interrupts at once, by acknowledging only those with ISR's we allow
                // the user process to use peripherals manually, and to manually check
                // whether or no those other interrupts had occurred.
                *pic = msk;
                rtu();
                // The ISR will only exit on a trap in the Zip archtecture. There is
                // no RETI instruction. Since the PIC holds all interrupts disabled,
                // there is no need to check for further interrupts.
                //
                // The tricky part is that, because of how the PIC is built, the ISR cannot
                // re-enable its own interrupt without re-enabling all interrupts. Hence, we
                // look at R0 upon ISR completion to know if an interrupt needs to be
                // re-enabled.
                mov(uR0,tmp);
                picv |= (tmp & 0x7fff) << 16;
            }
        }
        RESTORE_PARTIAL_CONTEXT;
        // Re-activate all (requested) interrupts
        *pic = picv | 0x80000000;
    }
}
```

Table 4.2: Traditional Interrupt handling

```
SAVE_PARTIAL_CONTEXT:
    ; We save R0, CC, and PC only
    MOV -3(uSP),R3
    MOV uR0,R0
    MOV uCC,R1
    MOV uPC,R2
    STO R0,(R3) ; Exploit memory pipelining:
    STO R1,1(R3) ; All instructions write to stack
    STO R2,2(R3) ; All offsets increment by one
    MOV R3,uSP ; Return the updated stack pointer
```

Table 4.3: Example Saving Minimal User Context

```
RESTORE_PARTIAL_CONTEXT:
    ; We retore R0, CC, and PC only
    MOV uSP,R3 ; Return the updated stack pointer
    LOD R0,(R3),R0 ; Exploit memory pipelining:
    LOD R1,1(R3),R1 ; All instructions write to stack
    LOD R2,2(R3),R2 ; All offsets increment by one
    MOV R0,uR0
    MOV R1,uCC
    MOV R2,uPC
    MOV 3(R3),uSP
```

Table 4.4: Example Restoring Minimal User Context

```
idle_task:
            ; Wait for the next interrupt, then switch to supervisor task
            WAIT
            ; When we come back, it's because the supervisor wishes to
            ; wait for an interrupt again, so go back to the top.
            BRA idle_task
```

Table 4.5: Example Idle Loop

```
void memcp(void *dest, void *src, int len) {
     for(int i=0; i<len; i++)
        *dest++ = *src++;
}
```

Table 4.6: Example Memory Copy code in C

This should be the lowest priority task, the task that runs when nothing else can. It will help lower the FPGA power usage overall—at least its dynamic power usage.

4.4 Example: Memory Copy

One common operation is that of a memory move or copy. Consider the C code shown in Tbl. 4.6. This same code can be translated in Zip Assembly as shown in Tbl. 4.7. This example points out several things associated with the Zip CPU. First, a straightforward implementation of a for loop is not the fastest loop structure. For this reason, we have placed the test to continue at the end. Second, all pointers are `void` pointers to arbitrary 32–bit data types. The Zip CPU does not have explicit support for smaller or larger data types, and so this memory copy cannot be applied at a byte level. Third, we've optimized the conditional jump to a return instruction into a conditional return instruction.

4.5 Example: 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 ways. The first step is that an interrupt happens. Anytime an interrupt happens, the CPU needs to execute the following tasks in supervisor mode:

1. Check for a trap instruction, or other user exception such as a break, bus error, division by zero error, or floating point exception. That is, if the user process needs attending then we may not wish to adjust the context, check interrupts, or call the scheduler. Tbl. 4.8 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.[1]

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

[1]The one exception to this rule is upon reset where supervisor mode is entered at a pre–programmed wishbone memory address.

```
memcp:
                ; R0 = *dest, R1 = *src, R2 = LEN, R3 = return addr
                ; The following will operate in 12N + 19 clocks.
                CMP 0,R2
                MOV.Z R3,PC ; A conditional return
                SUB 1,SP ; Create a stack frame
                STO R4,(SP) ; and a local variable
                ; (4 stalls, cannot be further scheduled away)
loop:
                LOD (R1),R4
                ; (4 stalls, cannot be scheduled away)
                STO R4,(R0) ; (4 schedulable stalls, has no impact now)
                SUB 1,R2
                BZ memcpend
                ADD 1,R0
                ADD 1,R1
                BRA loop
                ; (1 stall on a BRA instruction)
memcpend:       LOD (SP),R4
                ; (4 stalls, cannot be further scheduled away)
                ADD 1,SP
                JMP R3
                ; (4 stalls)
```

Table 4.7: Example Memory Copy code in Zip Assembly

```
return_to_user:
                ; 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.
                RTU
trap_check:
                MOV uCC,R0
                TST $TRAP |$BUSERR |$DIVE |$FPE,R0
                BNZ swap_out
                ; Do something here to execute the trap
                ; Don't need to call the scheduler, so we can just return
                BRA return_to_user
```

Table 4.8: Checking for whether the user task needs our attention

```
swap_out:
            MOV -15(uSP),R5
            STO R5,stack(R12)
            MOV uR0,R0
            MOV uR1,R1
            MOV uR2,R2
            MOV uR3,R3
            MOV uR4,R4
            STO R0,(R5) ; Exploit memory pipelining:
            STO R1,1(R5) ; All instructions write to stack
            STO R2,2(R5) ; All offsets increment by one
            STO R3,3(R5) ; Longest pipeline is 5 cycles.
            STO R4,4(R5)
            ...; Need to repeat for all user registers
            MOV uR10,R0
            MOV uR11,R1
            MOV uR12,R2
            MOV uCC,R3
            MOV uPC,R4
            STO R0,10(R5)
            STO R1,11(R5)
            STO R2,12(R5)
            STO R3,13(R5)
            STO R4,14(R5)
            ; We can skip storing the stack, uSP, since it'll be stored
            ; elsewhere (in the task structure)
```

Table 4.9: Example Storing User Task Context

3. Preserve the old context. This involves pushing all the user registers onto the user stack and then copying the resulting stack address into the tasks task structure, as shown in Tbl. 4.9. For the sake of discussion, we assume the supervisor maintains a pointer to the current task's structure in supervisor register `R12`, and that `stack` is an offset to the beginning of this structure indicating where the stack pointer is to be kept within it.

   For those who are still interested, the full code for this context save can be found as an assembler macro within the assembler include file, `sys.i`.

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. Example code for this is shown in Tbl. 4.10.

5. Interrupt handling. An interrupt handler within the Zip System is nothing more than a task. At context swap time, the supervisor needs to disable all of the interrupts that have tripped, and then enable all of the tasks that would deal with each of these interrupts. These can be user tasks, run at higher priority than any other user tasks. Either way, they will need to re–enable their own interrupt themselves, if the interrupt is still relevant.

```
‘define WATCHDOG_ADDRESS 32’hc000_0002
‘define WATCHDOG_TICKS 32’d1_000_000 ; = 10 ms
  LDI WATCHDOG_ADDRESS,R0
  LDI WATCHDOG_TICKS,R1
  STO R1,(R0)
```

Table 4.10: Example Watchdog Reset

An example of this master interrut handling is shown in Tbl. 4.11.

6. Calling the scheduler. This needs to be done to pick the next task to switch to. It 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

   For our purposes here, we'll just assume that a pointer to the current task is maintained in `R12`, that a `JSR scheduler` is called, and that the next current task is likewise placed into `R12`.

7. Restore the new tasks context. Given that the scheduler has returned a task that can be run at this time, the stack pointer needs to be pulled out of the tasks task structure, placed into the user register, and then the rest of the user registers need to be popped back off of the stack to run this task. An example of this is shown in Tbl. 4.12, assuming as before that the task pointer is found in supervisor register `R12`. As with storing the user context, the full code associated with restoring the user context can be found in the assembler include file, `sys.i`.

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. Jump back to the instruction just before saving the last tasks context, because that location in memory contains the return from interrupt command that we are going to need to execute, in order to guarantee that we return back here again.

```
pre_handler:
        LDI PIC_ADDRESS,R0
        ; Start by grabbing the interrupt state from the interrupt
        ; controller. We'll store this into the register R7 so that
        ; we can keep and preserve this information for the scheduler
        ; to use later.
        LOD (R0),R1
        MOV R1,R7
        ; As a next step, we need to acknowledge and disable all active
        ; interrupts. We'll start by calculating all of our active
        ; interrupts.
        AND 0x07fff,R1
        ; Put the active interrupts into the upper half of R1
        ROL 16,R1
        LDILO 0x0ffff,R1
        AND R7,R1
        ; Acknowledge and disable active interrupts
        ; This also disables all interrupts from the controller, so
        ; we'll need to re-enable interrupts in general shortly
        STO R1,(R0)
        ; We leave our active interrupt mask in R7 so the scheduler can
        ; release any tasks that depended upon them.
```

Table 4.11: Example checking for active interrupts

```
swap_in:
        LOD stack(R12),R5
        MOV 15(R1),uSP
        ; Be sure to exploit the memory pipelining capability
        LOD (R5),R0
        LOD 1(R5),R1
        LOD 2(R5),R2
        LOD 3(R5),R3
        LOD 4(R5),R4
        MOV R0,uR0
        MOV R1,uR1
        MOV R2,uR2
        MOV R3,uR3
        MOV R4,uR4
        ...; Need to repeat for all user registers
        LOD 10(R5),R0
        LOD 11(R5),R1
        LOD 12(R5),R2
        LOD 13(R5),R3
        LOD 14(R5),R4
        MOV R0,uR10
        MOV R1,uR11
        MOV R2,uR12
        MOV R3,uCC
        MOV R4,uPC
        BRA return_to_user
```

Table 4.12: Example Restoring User Task Context

5.

Registers

The ZipSystem registers fall into two categories, ZipSystem internal registers accessed via the ZipCPU shown in Tbl. 5.1, and the two debug registers shown in Tbl. 5.2.

Name	Address	Width	Access	Description
PIC	0xc0000000	32	R/W	Primary Interrupt Controller
WDT	0xc0000001	32	R/W	Watchdog Timer
	0xc0000002	32	R	Address of last bus error
CTRIC	0xc0000003	32	R/W	Secondary Interrupt Controller
TMRA	0xc0000004	32	R/W	Timer A
TMRB	0xc0000005	32	R/W	Timer B
TMRC	0xc0000006	32	R/W	Timer C
JIFF	0xc0000007	32	R/W	Jiffies
MTASK	0xc0000008	32	R/W	Master Task Clock Counter
MMSTL	0xc0000009	32	R/W	Master Stall Counter
MPSTL	0xc000000a	32	R/W	Master Pre–Fetch Stall Counter
MICNT	0xc000000b	32	R/W	Master Instruction Counter
UTASK	0xc000000c	32	R/W	User Task Clock Counter
UMSTL	0xc000000d	32	R/W	User Stall Counter
UPSTL	0xc000000e	32	R/W	User Pre–Fetch Stall Counter
UICNT	0xc000000f	32	R/W	User Instruction Counter
DMACTRL	0xc0000010	32	R/W	DMA Control Register
DMALEN	0xc0000011	32	R/W	DMA total transfer length
DMASRC	0xc0000012	32	R/W	DMA source address
DMADST	0xc0000013	32	R/W	DMA destination address

Table 5.1: Zip System Internal/Peripheral Registers

Name	Address	Width	Access	Description
ZIPCTRL	0	32	R/W	Debug Control Register
ZIPDATA	1	32	R/W	Debug Data Register

Table 5.2: Zip System Debug Registers

Bit #	Access	Description
31	R/W	Master Interrupt Enable
30...16	R/W	Interrupt Enables, write '1' to change
15	R	Current Master Interrupt State
15...0	R/W	Input Interrupt states, write '1' to clear

Table 5.3: Interrupt Controller Register Bits

5.1 Peripheral Registers

The peripheral registers, listed in Tbl. 5.1, are shown in the CPU's address space. These may be accessed by the CPU at these addresses, and when so accessed will respond as described in Chapt. 3. These registers will be discussed briefly again here.

5.1.1 Interrupt Controller(s)

The Zip CPU Interrupt controller has four different types of bits, as shown in Tbl. 5.3. 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.

Bits 30 . . . 16 are interrupt enable bits. Should the interrupt line go hi while enabled, an interrupt will be generated. (All interrupts are positive edge triggered.) To set an interrupt enable bit, one needs to write the master interrupt enable while writing a '1' to this the bit. To clear, one need only write a '0' to the master interrupt enable, while leaving this line high.

Bits 15...0 are the current state of the interrupt vector. Interrupt lines trip when they go high, and remain tripped until they are acknowledged. If the interrupt goes high for longer than one pulse, it may be high when a clear is requested. If so, the interrupt will not clear. The line must go low again before the status bit can be cleared.

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

1. The Supervisor will first, while in the interrupts disabled mode, write a `32'h800f000f` to the controller. The supervisor may then switch to the user state with interrupts enabled.

2. When an interrupt occurs, the supervisor will switch to the interrupt state. It will 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 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. Once all interrupts have been handled, the supervisor will write a `32'h8000_0000` to the interrupt register to re-enable interrupt generation.

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

Table 5.4: Timer Register Bits

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

Table 5.5: Jiffies Register Bits

6. The supervisor should also check the user trap bit, and possible soft interrupt bits here, but this action has nothing to do with the interrupt control register.
7. The supervisor will then leave interrupt mode, possibly adjusting whichever task is running, by executing a return from interrupt command.

5.1.2 Timer Register

Leaving the interrupt controller, we show the timer registers bit definitions in Tbl. 5.4. As you may recall, the timer just counts down to zero and then trips an interrupt. Writing to the current timer value sets that value, 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.

5.1.3 Jiffies

The Jiffies register is somewhat similar in that the register always changes. In this case, the register counts up, whereas the timer always counted down. Reads from this register, as shown in Tbl. 5.5, always return the time value contained in the register. Writes greater than the current Jiffy value, that is where the new value minus the old value is greater than zero while ignoring truncation, will set a new Jiffy interrupt time. At that time, the Jiffy vector will clear, and another interrupt time may either be written to it, or it will just continue counting without activating any more interrupts.

5.1.4 Performance Counters

The Zip CPU also supports several counter peripherals, mostly in the way of process accounting. This peripherals have a single register associated with them, shown in Tbl. 5.6. Writes to this register set the new counter value. Reads read the current counter value.

The current design operation of these counters is that of performance counting. Two sets of four registers are available for keeping track of performance. The first is a task counter. This just counts

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

Table 5.6: Counter Register Bits

clock ticks. The second counter is a prefetch stall counter, then an master stall counter. These allow the CPU to be evaluated as to how efficient it is. The fourth and final counter is an instruction counter, which counts how many instructions the CPU has issued.

It is envisioned that these counters will be used 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.

5.1.5 DMA Controller

The final peripheral to discuss is the DMA controller. This controller has four registers. Of these four, the 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. The registers can be written to when the DMA is idle, and read at any time. The control register, however, will need some more explanation.

The bit allocation of the control register is shown in Tbl. 5.7. 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'h0fed03ff to the control register will start any memory transfer. On the other hand, if you wished to read from a serial port (constant address) and put the result into a buffer every time a word was available, you might wish to write 32'h2fed8000–this assumes, of course, that you have a serial port wired to the zero bit of this interrupt control. (The DMA controller does not use the interrupt controller, and cannot clear interrupts.) As a third example, if you wished to write to an external FIFO anytime it was less than half full (had fewer than 512 items), and interrupt line 2 indicated this condition, you might wish to issue a 32'h1fed8dff to this port.

5.2 Debug Port Registers

Accessing the Zip System via the debug port isn't as straight forward as accessing the system via the wishbone bus. The debug port itself has been reduced to two addresses, as outlined earlier in Tbl. 5.2. Access to the Zip System begins with the Debug Control register, shown in Tbl. 5.8.

The first step in debugging access is to determine whether or not the CPU is halted, and to halt it if not. To do this, first write a '1' to the Command HALT bit. This will halt the CPU and place it into debug mode. Once the CPU is halted, the stall status bit will drop to zero. Thus, if bit 10 is high and bit 9 low, the debug port is open to examine the internal state of the CPU.

At this point, the external debugger may examine internal state information from within the CPU. To do this, first write again to the command register a value (with command halt still high) containing the address of an internal register of interest in the bottom 6 bits. Internal registers that

Bit #	Access	Description
31	R	DMA Active
30	R	Wishbone error, transaction aborted. This bit is cleared the next time this register is written to.
29	R/W	Set to '1' to prevent the controller from incrementing the source address, '0' for normal memory copy.
28	R/W	Set to '1' to prevent the controller from incrementing the destination address, '0' for normal memory copy.
27 . . . 16	W	The DMA Key. Write a 12'hfed to these bits to start the activate any DMA transfer.
27	R	Always reads '0', to force the deliberate writing of the key.
26 . . . 16	R	Indicates the number of items in the transfer buffer that have yet to be written.
15	R/W	Set to '1' to trigger on an interrupt, or '0' to start immediately upon receiving a valid key.
14. . . 10	R/W	Select among one of 32 possible interrupt lines.
9. . . 0	R/W	Intermediate transfer length minus one. Thus, to transfer one item at a time set this value to 0. To transfer 1024 at a time, set it to 1024.

Table 5.7: DMA Control Register Bits

Bit #	Access	Description
31. . . 14	R	External interrupt state. Bit 14 is valid for one interrupt only, bit 15 for two, etc.
13	R	CPU GIE setting
12	R	CPU is sleeping
11	W	Command clear PF cache
10	R/W	Command HALT, Set to '1' to halt the CPU
9	R	Stall Status, '1' if CPU is busy (i.e., not halted yet)
8	R/W	Step Command, set to '1' to step the CPU, also sets the halt bit
7	R	Interrupt Request Pending
6	R/W	Command RESET
5. . . 0	R/W	Debug Register Address

Table 5.8: Debug Control Register Bits

may be accessed this way are listed in Tbl. 5.9. Primarily, these "registers" include access to the entire CPU register set, as well as the internal peripherals. To read one of these registers once the address is set, simply issue a read from the data port. To write one of these registers or peripheral ports, simply write to the data port after setting the proper address.

In this manner, all of the CPU's internal state may be read and adjusted.

As an example of how to use this, consider what would happen in the case of an external break point. If and when the CPU hits a break point that causes it to halt, the Command HALT bit will activate on its own, the CPU will then raise an external interrupt line and wait for a debugger to examine its state. After examining the state, the debugger will need to remove the breakpoint by writing a different instruction into memory and by writing to the command register while holding the clear cache, command halt, and step CPU bits high, (32'hd00). The debugger may then replace the breakpoint now that the CPU has gone beyond it, and clear the cache again (32'h500).

To leave this debug mode, simply write a '32'h0' value to the command register.

Name	Address	Width	Access	Description
sR0	0	32	R/W	Supervisor Register R0
sR1	0	32	R/W	Supervisor Register R1
sSP	13	32	R/W	Supervisor Stack Pointer
sCC	14	32	R/W	Supervisor Condition Code Register
sPC	15	32	R/W	Supervisor Program Counter
uR0	16	32	R/W	User Register R0
uR1	17	32	R/W	User Register R1
uSP	29	32	R/W	User Stack Pointer
uCC	30	32	R/W	User Condition Code Register
uPC	31	32	R/W	User Program Counter
PIC	32	32	R/W	Primary Interrupt Controller
WDT	33	32	R/W	Watchdog Timer
BUS	34	32	R	Last Bus Error
CTRIC	35	32	R/W	Secondary Interrupt Controller
TMRA	36	32	R/W	Timer A
TMRB	37	32	R/W	Timer B
TMRC	38	32	R/W	Timer C
JIFF	39	32	R/W	Jiffies peripheral
MTASK	40	32	R/W	Master task clock counter
MMSTL	41	32	R/W	Master memory stall counter
MPSTL	42	32	R/W	Master Pre-Fetch Stall counter
MICNT	43	32	R/W	Master instruction counter
UTASK	44	32	R/W	User task clock counter
UMSTL	45	32	R/W	User memory stall counter
UPSTL	46	32	R/W	User Pre-Fetch Stall counter
UICNT	47	32	R/W	User instruction counter
DMACMD	48	32	R/W	DMA command and status register
DMALEN	49	32	R/W	DMA transfer length
DMARD	50	32	R/W	DMA read address
DMAWR	51	32	R/W	DMA write address

Table 5.9: Debug Register Addresses

6.

Wishbone Datasheets

The Zip System supports two wishbone ports, a slave debug port and a master port for the system itself. These are shown in Tbl. 6.1 and Tbl. 6.2 respectively. I do not recommend that you connect

Description	Specification
Revision level of wishbone	WB B4 spec
Type of interface	Slave, Read/Write, single words only
Address Width	1–bit
Port size	32–bit
Port granularity	32–bit
Maximum Operand Size	32–bit
Data transfer ordering	(Irrelevant)
Clock constraints	Works at 100 MHz on a Basys–3 board, and 80 MHz on a XuLA2–LX25
Signal Names	Signal Name / Wishbone Equivalent i_clk / CLK_I i_dbg_cyc / CYC_I i_dbg_stb / 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 6.1: Wishbone Datasheet for the Debug Interface

these together through the 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 the wire to the `wb_cyc` line. As for the `RTY`, all the CPU recognizes at this point are bus errors—it cannot tell the difference between a temporary and a permanent bus error.

Description	Specification
Revision level of wishbone	WB B4 spec
Type of interface	Master, Read/Write, single cycle or pipelined
Address Width	(Zip System parameter, can be up to 32–bit bits)
Port size	32–bit
Port granularity	32–bit
Maximum Operand Size	32–bit
Data transfer ordering	(Irrelevant)
Clock constraints	Works at 100 MHz on a Basys–3 board, and 80 MHz on a XuLA2–LX25
Signal Names	Signal Name / Wishbone Equivalent i_clk / CLK_O o_wb_cyc / CYC_O o_wb_stb / STB_O o_wb_we / WE_O o_wb_addr / ADR_O o_wb_data / DAT_O i_wb_ack / ACK_I i_wb_stall / STALL_I i_wb_data / DAT_I i_wb_err / ERR_I

Table 6.2: Wishbone Datasheet for the CPU as Master

7.

Clocks

This core is based upon the Basys–3 development board sold by Digilent. The Basys–3 development board contains one external 100 MHz clock, which is sufficient to run the Zip CPU core. I hesitate

Name	Source	Rates (MHz)		Description
		Max	Min	
i_clk	External	100 MHz	100 MHz	System clock.

Table 7.1: List of Clocks

to suggest that the core can run faster than 100 MHz, since I have had struggled with various timing violations to keep it at 100 MHz. So, for now, I will only state that it can run at 100 MHz.

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

8.

I/O Ports

The I/O ports to the Zip CPU 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. 8.1 and 8.2 respectively.

There are only four other 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. 8.3. The clock line was discussed briefly in Chapt. 7. We typically run it at 100 MHz. The reset line is an active high reset. When asserted, the CPU will start running again from its reset address in memory. Further, depending upon how the CPU is configured and specifically on the `START_HALTED` parameter, it may or may not start running automatically. The `i_ext_int` line is for an external interrupt. This line may be as wide as 6 external interrupts, depending upon the setting of the `EXTERNAL_INTERRUPTS` line. As currently configured, the ZipSystem only supports one such interrupt line by default. For us, this line is the output of another interrupt controller, but that's a board specific setup detail. Finally, the Zip System produces one external interrupt whenever the CPU halts to wait for the debugger.

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`	32	Output	Bus address
`o_wb_data`	32	Output	Data on WB write
`i_wb_ack`	1	Input	Slave has completed a R/W cycle
`i_wb_stall`	1	Input	WB bus slave not ready
`i_wb_data`	32	Input	Incoming bus data
`i_wb_err`	1	Input	Bus Error indication

Table 8.1: CPU Master Wishbone I/O Ports

Port	Width	Direction	Description
i_wb_cyc	1	Input	Indicates an active Wishbone cycle
i_wb_stb	1	Input	WB Strobe signal
i_wb_we	1	Input	Write enable
i_wb_addr	1	Input	Bus address, command or data port
i_wb_data	32	Input	Data on WB write
o_wb_ack	1	Output	Slave has completed a R/W cycle
o_wb_stall	1	Output	WB bus slave not ready
o_wb_data	32	Output	Incoming bus data

Table 8.2: CPU Debug Wishbone I/O Ports

Port	Width	Direction	Description
i_clk	1	Input	The master CPU clock
i_rst	1	Input	Active high reset line
i_ext_int	1...16	Input	Incoming external interrupts, actual value set by implementation parameter
o_ext_int	1	Output	CPU Halted interrupt

Table 8.3: I/O Ports

9.

Initial Assessment

Having now worked with the Zip CPU for a while, it is worth offering an honest assessment of how well it works and how well it was designed. At the end of this assessment, I will propose some changes that may take place in a later version of this Zip CPU to make it better.

9.1 The Good

- The Zip CPU is light weight and fully featured as it exists today. For anyone who wishes to build a general purpose CPU and then to experiment with building and adding particular features, the Zip CPU makes a good starting point–it is fairly simple. Modifications should be simple enough.
- The Zip CPU was designed to be an implementable soft core that could be placed within an FPGA, controlling actions internal to the FPGA. It fits this role rather nicely. It does not fit the role of a system on a chip very well, but then it was never intended to be a system on a chip but rather a system within a chip.
- The extremely simplified instruction set of the Zip CPU was a good choice. Although it does not have many of the commonly used instructions, PUSH, POP, JSR, and RET among them, the simplified instruction set has demonstrated an amazing versatility. I will contend therefore and for anyone who will listen, that this instruction set offers a full and complete capability for whatever a user might wish to do with two exceptions: bytewise character access and accelerated floating-point support.
- This simplified instruction set is easy to decode.
- The simplified bus transactions (32-bit words only) were also very easy to implement.
- The pipelined load/store approach is novel, and can be used to greatly increase the speed of the processor.
- The novel approach of having a single interrupt vector, which just brings the CPU back to the instruction it left off at within the last interrupt context doesn't appear to have been that much of a problem. If most modern systems handle interrupt vectoring in software anyway, why maintain hardware support for it?
- My goal of a high rate of instructions per clock may not be the proper measure. For example, if instructions are being read from a SPI flash device, such as is common among FPGA

implementations, these same instructions may suffer stalls of between 64 and 128 cycles per instruction just to read the instruction from the flash. Executing the instruction in a single clock cycle is no longer the appropriate measure. At the same time, it should be possible to use the DMA peripheral to copy instructions from the FLASH to a temporary memory location, after which they may be executed at a single instruction cycle per access again.

9.2 The Not so Good

- While one of the stated goals was to use a small amount of logic, 3k LUTs isn't that impressively small. Indeed, it's really much too expensive when compared against other 8 and 16-bit CPUs that have less than 1k LUTs.

  Still, ... it's not bad, it's just not astonishingly good.

- The fact that the instruction width equals the bus width means that the instruction fetch cycle will always be interfering with any load or store memory operation, with the only exception being if the instruction is already in the cache.

  This could be fixed in one of three ways: the instruction set architecture could be modified to handle Very Long Instruction Words (VLIW) so that each 32–bit word would encode two or more instructions, the instruction fetch bus width could be increased from 32–bits to 64–bits or more, or the instruction bus could be separated from the data bus. Any and all of these approaches would increase the overall LUT count.

- The (non-existant) floating point unit was an after-thought, isn't even built as a potential option, and most likely won't support the full IEEE standard set of FPU instructions–even for single point precision. This (non-existant) capability would benefit the most from an out-of-order execution capability, which the Zip CPU does not have.

  Still, sharing FPU registers with the main register set was a good idea and worth preserving, as it simplifies context swapping.

  Perhaps this really isn't a problem, but rather a feature. By not implementing FPU instructions, the Zip CPU maintains a lower LUT count than it would have if it did implement these instructions.

- The CPU has no character support. This is both good and bad. Realistically, the CPU works just fine without it. Characters can be supported as subsets of 32-bit words without any problem. Practically, though, it will make compiling non-Zip CPU code difficult–especially anything that assumes sizeof(int)=4*sizeof(char), or that tries to create unions with characters and integers and then attempts to reference the address of the characters within that union.

- The Zip CPU does not support a data cache. One can still be built externally, but this is a limitation of the CPU proper as built. Further, under the theory of the Zip CPU design (that of an embedded soft-core processor within an FPGA, where any "address" may reference either memory or a peripheral that may have side-effects), any data cache would need to be based upon an initial knowledge of whether or not it is supporting memory (cachable) or peripherals. This knowledge must exist somewhere, and that somewhere is currently (and by design) external to the CPU.

This may also be written off as a "feature" of the Zip CPU, since the addition of a data cache can greatly increase the LUT count of a soft core.

The Zip CPU compensates for this via its pipelined load and store instructions.

- Many other instruction sets offer three operand instructions, whereas the Zip CPU only offers two operand instructions. This means that it takes the Zip CPU more instructions to do many of the same operations. The good part of this is that it gives the Zip CPU a greater amount of flexibility in its immediate operand mode, although that increased flexibility isn't necessarily as valuable as one might like.
- The Zip CPU does not currently detect and trap on either illegal instructions or bus errors. Attempts to access non–existent memory quietly return erroneous results, rather than halting the process (user mode) or halting or resetting the CPU (supervisor mode).
- The Zip CPU doesn't support out of order execution. I suppose it could be modified to do so, but then it would no longer be the "simple" and low LUT count CPU it was designed to be. The two primary results are that 1) loads may unnecessarily stall the CPU, even if other things could be done while waiting for the load to complete, 2) bus errors on stores will never be caught at the point of the error, and 3) branch prediction becomes more difficult.
- Although switching to an interrupt context in the Zip CPU design doesn't require a tremendous swapping of registers, in reality it still does–since any task swap still requires saving and restoring all 16 user registers. That's a lot of memory movement just to service an interrupt.
- The Zip CPU is by no means generic: it will never handle addresses larger than 32-bits (16GB) without a complete and total redesign. This may limit its utility as a generic CPU in the future, although as an embedded CPU within an FPGA this isn't really much of a limit or restriction.
- While the Zip CPU has its own assembler, it has no linker and does not (yet) support a compiler. The standard C library is an even longer shot. My dream of having binutils and gcc support has not been realized and at this rate may not be realized. (I've been intimidated by the challenge everytime I've looked through those codes.)

9.3 The Next Generation

This section could also be labeled as my "To do" list.

Given the feedback listed above, perhaps its time to consider what changes could be made to improve the Zip CPU in the future. I offer the following as proposals:

- **Remove the low LUT goal.** It wasn't really achieved, and the proposals below will only increase the amount of logic the Zip CPU requires. While I expect that the Zip CPU will always be somewhat of a light weight, it will never be the smallest kid on the block.

  I'm actually struggling with this idea. The whole goal of the Zip CPU was to be light weight. Wouldn't it make more sense to create and maintain options whereby it would remain lightweight? For example, if the process accounting registers are anything but light weight, why keep them? Why not instead make some compile flags that just turn them off, keeping the CPU lightweight? The same holds for the prefetch cache.

- The '`.V`' condition was never used in any code other than my test code. Suggest changing it to a '`.LE`' condition, which seems to be more useful.
- **Consider a more traditional Instruction Cache.** The current pipelined instruction cache just reads a window of memory into its cache. If the CPU leaves that window, the entire cache is invalidated. A more traditional cache, however, might allow common subroutines to stay within the cache without invalidating the entire cache structure.
- **Very Long Instruction Word (VLIW).** The goal here would be to create a new instruction set whereby two instructions would be encoded in each 32–bit word. While this may speed up CPU operation, it would necessitate an instruction redesign.