Gisselquist
Technology, LLC

ZIP CPU SPECIFICATION

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

August 22, 2015

Revision History

Rev.	Date	Author	Description
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. I would like to be able to generate Verilog code 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 chip 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 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**, the **ALU/Memory** unit, and **Write-back**. See Fig. 1.1 for a diagram of this structure.
- Completely open source, licensed under the GPL.[1]

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:

- **Extenal 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:

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

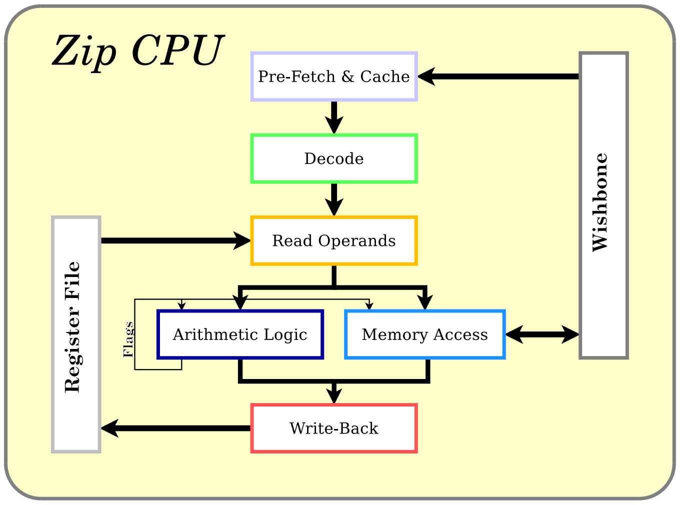

Figure 1.1: Zip CPU internal pipeline architecture

the 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 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. I therefore have a prefetch cache that issues pipelined wishbone accesses to memory and then pushes instructions at the CPU. Sadly, this accounts for about 20% of the logic in the entire CPU, or 15% of the logic in the entire system.

- **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`.

- **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:

  For example, in order to facilitate interrupt handling and debug stepping, the CPU needs to know what instructions have finished, and which have not. In other words, it needs to know where it can restart the pipeline from. Once restarted, it must act as though it had never stopped. This killed my idea of delayed branching, since what would be the appropriate program counter to restart at? The one the CPU was going to branch to, or the ones in the delay slots? This also makes the idea of compressed instruction codes difficult, since, again, where do you restart on interrupt?

  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 preceeding 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 preceeding are all wiped clean.

  The discrete execution model allows such things as sleeping: 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.

  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.

- **Verilog Modules:** When examining how other processors worked here on open cores, many of them had one separate module per pipeline stage. While this appeared to me to be a fascinating and commendable idea, my own implementation didn't work out quite so nicely.

  As an example, the decode module produces a *lot* of control wires and registers. Creating a module out of this, with only the simplest of logic within it, seemed to be more a lesson in passing wires around, rather than encapsulating logic.

  Another example was the register writeback section. I would love this section to be a module in its own right, and many have made them such. However, other modules depend upon writeback results other than just what's placed in the register (i.e., the control wires). For these reasons, I didn't manage to fit this section into it's own module.

The result is that the majority of the CPU code can be found in the `zipcpu.v` file.

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 bit 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. 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. The lower 10 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 eight condition codes, the bottom four are the current flags: Zero (Z), Carry (C), Negative (N), and Overflow (V).

The next bit is a clock enable (0 to enable) or sleep bit (1 to put the CPU to sleep). Setting this bit will cause the CPU to wait for an interrupt (if interrupts are enabled), or to completely halt (if interrupts are disabled). 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.

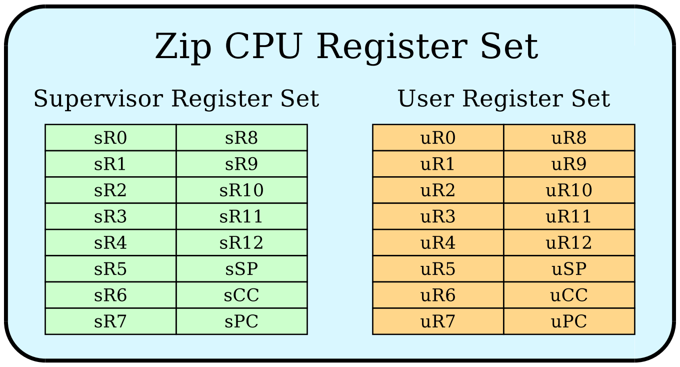

Figure 2.1: Zip CPU Register File

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 reserved for a floating point enable bit. When set, the arithmetic for the next instruction will be sent to a floating point unit. Such a unit may later be added as an extension to the Zip CPU. If the CPU does not support floating point instructions, this bit will never be set. The instruction set could also be simply extended to allow other data types in this fashion, such as two by 16–bit vector operations or four by 8–bit vector operations.

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.

These status register bits are summarized in Tbl. 2.1.

2.3 Conditional Instructions

Most, although not quite all, instructions are conditionally executed. From the four condition code flags, eight conditions are defined. These are shown in Tbl. 2.2. There is no condition code for less than or equal, not C or not V. Sorry, I ran out of space in 3–bits. Using these conditions will take an extra instruction and a pipeline stall. (Ex: *(Stall)*; `TST $4,CC; STO.NZ R0,(R1)`)

Bit	Meaning
9	Soft trap, set on a trap from user mode, cleared when returing to user mode
8	(Reserved for) Floating point enable
7	Halt on break, to support an external debugger
6	Step, single step the CPU in user mode
5	GIE, or Global Interrupt Enable
4	Sleep
3	V, or overflow bit.
2	N, or negative bit.
1	C, or carry bit.
0	Z, or zero bit.

Table 2.1: Condition Code / Status Register Bits

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

Table 2.2: Conditions for conditional operand execution

Bit 20	19 . . . 16	15 . . . 0
1'b0	20–bit Signed Immediate value	
1'b1	4-bit Register	16–bit Signed immediate offset

Table 2.3: Bit allocation for Operand B

2.4 Operand B

Many instruction forms have a 21-bit source "Operand B" associated with them. This Operand B is either equal to a register plus a signed immediate offset, or an immediate offset by itself. This value is encoded as shown in Tbl. 2.3.

Sixteen and twenty bit immediates don't make sense for all instructions. For example, what is the point of a 20–bit immediate when executing a 16–bit multiply? Likewise, why have a 16–bit immediate when adding to a logical or arithmetic shift? In these cases, the extra bits are reserved for future instruction possibilities.

2.5 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.

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,[1] 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.

2.6 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-Usr and B-Usr bits. 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

[1]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.

the user bit set will be treated as a user register. The CPU will quietly ignore the supervisor bits while in user mode, and anything marked as a user register will always be valid. (Did I just say that in the last paragraph?)

2.7 Multiply Operations

The Zip CPU supports two Multiply operations, a 16x16 bit signed multiply (MPYS) and the same but unsigned (MPYU). In both cases, the operand is a register plus a 16-bit immediate, subject to the rule that the register cannot be the PC or CC registers. The PC register field has been stolen to create a multiply by immediate instruction. The CC register field is reserved.

2.8 Floating Point

The ZIP CPU does not support floating point operations. However, the instruction set reserves two possibilities for future floating point operations.

The first floating point operation hole in the instruction set involves setting the floating point bit in the CC register. The next instruction will simply interpret its operands as floating point instructions. Not all instructions, however, have floating point equivalents. Further, the immediate fields do not apply in floating point mode, and must be set to zero. Not all instructions make sense as floating point operations. Therefore, only the CMP, SUB, ADD, and MPY instructions may be issued as floating point instructions. Other instructions allow the examining of the floating point bit in the CC register. In all cases, the floating point bit is cleared one instruction after it is set.

The other possibility for floating point operations involves exploiting the hole in the instruction set that the NOOP and BREAK instructions reside within. These two instructions use 24–bits of address space. A simple adjustment to this space could create instructions with 4–bit register addresses for each register, a 3–bit field for conditional execution, and a 2–bit field for which operation. In this fashion, such a floating point capability would only fill 13–bits of the 24–bit field, still leaving lots of room for expansion.

In both cases, the Zip CPU would support 32–bit single precision floats only.

The current architecture does not support a floating point not-implemented interrupt. Any soft floating point emulation must be done deliberately.

2.9 Native Instructions

The instruction set for the Zip CPU is summarized in Tbl. 2.4.

As you can see, there's lots of room for instruction set expansion. The NOOP and BREAK instructions are the only instructions within one particular 24–bit hole. This spaces are reserved for future enhancements. For example, floating point operations, consisting of a 3-bit floating point operation, two 4-bit registers, no immediate offset, and a 3-bit condition would fit nicely into 14–bits of this address space–making it so that the floating point bit in the CC register need not be used.

Op Code	31…24		23…16			15…8	7…0	Sets CC?
CMP(Sub)	4'h0	D. Reg	Cond.	Operand B				Yes
TST(And)	4'h1	D. Reg	Cond.	Operand B				Yes
MOV	4'h2	D. Reg	Cond.	A-Usr	B-Reg	B-Usr	15'bit signed offset	
LODI	4'h3	R. Reg	24'bit Signed Immediate					
NOOP	4'h4	4'he	24'h00					
BREAK	4'h4	4'he	24'h01					
Rsrd	4'h4	4'he	24'bits, but not 0 or 1.					
LODIHI	4'h4	4'hf	Cond.	1'b1	R. Reg	16-bit Immediate		
LODILO	4'h4	4'hf	Cond.	1'b0	R. Reg	16-bit Immediate		
16-b MPYU	4'h4	R. Reg	Cond.	1'b0	Reg	16-bit Offset		Yes
16-b MPYU(I)	4'h4	R. Reg	Cond.	1'b0	4'hf	16-bit Offset		Yes
16-b MPYS	4'h4	R. Reg	Cond.	1'b1	Reg	16-bit Offset		Yes
16-b MPYS(I)	4'h4	R. Reg	Cond.	1'b1	4'hf	16-bit Offset		Yes
ROL	4'h5	R. Reg	Cond.	Operand B, truncated to low order 5 bits				
LOD	4'h6	R. Reg	Cond.	Operand B address				
STO	4'h7	D. Reg	Cond.	Operand B address				
SUB	4'h8	R. Reg	Cond.	Operand B				Yes
AND	4'h9	R. Reg	Cond.	Operand B				Yes
ADD	4'ha	R. Reg	Cond.	Operand B				Yes
OR	4'hb	R. Reg	Cond.	Operand B				Yes
XOR	4'hc	R. Reg	Cond.	Operand B				Yes
LSL/ASL	4'hd	R. Reg	Cond.	Operand B, imm. trucated to 6 bits				Yes
ASR	4'he	R. Reg	Cond.	Operand B, imm. trucated to 6 bits				Yes
LSR	4'hf	R. Reg	Cond.	Operand B, imm. trucated to 6 bits				Yes

Table 2.4: Zip CPU Instruction Set

2.10 Derived Instructions

The ZIP CPU supports many other common instructions, but not all of them are single cycle instructions. The derived instruction tables, Tbls. 2.5, 2.6, and 2.7, 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.

2.11 Pipeline Stages

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

1. **Prefetch**: Read instruction from memory (cache if possible). 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.
2. **Decode**: Decode instruction into op code, 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 two tracks: An **ALU** which will accomplish a simple instruction, and the **MemOps** stage which accomplishes memory read/write.
   - Loads stall instructions that access the register until it is written to the register set.
   - Condition codes are available upon completion
   - Issuing an instruction to the memory while the memory is busy will stall the entire pipeline. If the bus deadlocks, only a reset will release the CPU. (Watchdog timer, anyone?)
   - The Zip CPU currently has no means of reading and acting on any error conditions on the bus.
5. **Write-Back**: Conditionally write back the result to the register set, applying the condition. This routine is bi-re-entrant: either the memory or the simple instruction may request a register write.

The Zip CPU does not support out of order execution. Therefore, if the memory unit stalls, every other instruction stalls. Memory stores, however, can take place concurrently with ALU operations, although memory reads cannot.

2.12 Pipeline Logic

How the CPU handles some instruction combinations can be telling when determining what happens in the pipeline. The following lists some examples:

Mapped	Actual	Notes
ADD Ra,Rx ADDC Rb,Ry	Add Ra,Rx ADD.C $1,Ry Add Rb,Ry	Add with carry
BRA.Cond +/-$Addr	Mov.cond $Addr+PC,PC	Branch or jump on condition. Works for 15–bit signed address offsets.
BRA.Cond +/-$Addr	LDI $Addr,Rx ADD.cond Rx,PC	Branch/jump on condition. Works for 23 bit address offsets, but costs a register, an extra instruction, and setsthe flags.
BNC PC+$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	Executed while in interrupt mode. In user mode this is simply a wait until interrupt instructioon.
INT	LDI $0,CC	Since we're using the CC register as a trap vector as well, this executes TRAP #0.
IRET	OR $GIE,CC	Also an RTU instruction (Return to Userspace)
JMP R6+$Addr	MOV $Addr(R6),PC	
JSR PC+$Addr	SUB $1,SP MOV $3+PC,R0 STO R0,1(SP) MOV $Addr+PC,PC ADD $1,SP	Jump to Subroutine. Note the required cleanup instruction after returning.
JSR PC+$Addr	MOV $3+PC,R12 MOV $addr+PC,PC	This is the high speed version of a subroutine call, necessitating a register to hold the last PC address. In its favor, this method doesn't suffer the mandatory memory access of the other approach.
LDI.l $val,Rx	LDIHI ($val>>16)&0x0ffff, Rx LDILO ($val & 0x0ffff)	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.

Table 2.5: Derived Instructions

Mapped	Actual	Notes
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 bytewise 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 othe 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	
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 $-1(SP),Rx ADD $1,SP	Note that for interrupt purposes, one can never depend upon the value at (SP). Hence you read from it, then increment it, lest having incremented it firost something then comes along and writes to that value before you can read the result.
PUSH Rx	SUB $1,SPa STO Rx,$1(SP)	
RESET	STO $1,$watchdog(R12) NOOP NOOP	This depends upon the peripheral base address being in R12. Another opportunity might be to jump to the reset address from within supervisor mode.
RET	LOD $-1(SP),PC	Note that this depends upon the calling context to clean up the stack, as outlined for the JSR instruction.

Table 2.6: Derived Instructions, continued

RET	MOV R12,PC	This is the high(er) speed version, that doesn't touch the stack. As such, it doesn't suffer a stall on memory read/write to the stack.
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.
TRAP #X	LDILO $x,CC	This approach uses the unused bits of the CC register as a TRAP address. The user will need to make certain that the SLEEP and GIE bits are not set in $x. LDI would also work, however using LDILO permits the use of conditional traps. (i.e., trap if the zero flag is set.) Should you wish to trap off of a register value, you could equivalently load $x into the register and then MOV it into the CC register.
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.
WAIT	Or $SLEEP,CC	Wait 'til interrupt. In an interrupts disabled context, this becomes a HALT instruction.

Table 2.7: Derived Instructions, continued

- **Delayed Branching**

  I had originally hoped to implement delayed branching. However, what happens in debug mode? That is, what happens when a debugger tries to single step an instruction? While I can easily single step the computer in either user or supervisor mode from externally, this processor does not appear able to step the CPU in user mode from within user mode–gosh, not even from within supervisor mode–such as if a process had a debugger attached. As the processor exists, I would have one result stepping the CPU from a debugger, and another stepping it externally.

  This is unacceptable, and so this CPU does not support delayed branching.

- **Register Result:** `MOV R0,R1; MOV R1,R2`

  What value does R2 get, the value of R1 before the first move or the value of R0? Placing the value of R0 into R1 requires a pipeline stall, and possibly two, as I have the pipeline designed.

  The ZIP CPU architecture requires that R2 must equal R0 at the end of this operation. Even better, such combinations do not (normally) stall the pipeline.

- **Condition Codes Result:** `CMP R0,R1;Mov.EQ $x,PC`

  At issue is the same item as above, save that the CMP instruction updates the flags that the MOV instruction depends upon.

  The Zip CPU architecture requires that condition codes must be updated and available immediately for the next instruction without stalling the pipeline.

- **Condition Codes Register Result:** `CMP R0,R1; MOV CC,R2`

  At issue is the fact that the logic supporting the CC register is more complicated than the logic supporting any other register.

  The ZIP CPU will stall for a cycle cycle on this instruction.

- **Delayed Branching:** `ADD $x,PC; MOV R0,R1`

  At issues is whether or not the instruction following the jump will take place before the jump. In other words, is the MOV to the PC register handled differently from an ADD to the PC register?

  In the Zip architecture, MOV'es and ADD's use the same logic (simplifies the logic).

As I've studied this, I find several approaches to handling pipeline issues. These approaches (and their consequences) are listed below.

- **All All issued instructions complete, Stages stall individually**

  What about a slow pre-fetch?

  Nominally, this works well: any issued instruction just runs to completion. If there are four issued instructions in the pipeline, with the writeback instruction being a write-to-PC instruction, the other three instructions naturally finish.

  This approach fails when reading instructions from the flash, since such reads require N clocks to clocks to complete. Thus there may be only one instruction in the pipeline if reading from flash, or a full pipeline if reading from cache. Each of these approaches would produce a different response.

- **Issued instructions may be canceled**

  Stages stall individually

  First problem: Memory operations cannot be canceled, even reads may have side effects on peripherals that cannot be canceled later. Further, in the case of an interrupt, it's difficult to know what to cancel. What happens in a `MOV.C $x,PC` followed by a `MOV $y,PC` instruction? Which get canceled?

  Because it isn't clear what would need to be canceled, this instruction combination is not recommended.

- **All issued instructions complete.**

  All stages are filled, or the entire pipeline stalls.

  What about debug control? What about register writes taking an extra clock stage? MOV R0,R1; MOV R1,R2 should place the value of R0 into R2. How do you restart the pipeline after an interrupt? What address do you use? The last issued instruction? But the branch delay slots may make that invalid!

  Reading from the CPU debug port in this case yields inconsistent results: the CPU will halt or step with instructions stuck in the pipeline. Reading registers will give no indication of what is going on in the pipeline, just the results of completed operations, not of operations that have been started and not yet completed. Perhaps we should just report the state of the CPU based upon what instructions (PC values) have successfully completed? Thus the debug instruction is the one that will write registers on the next clock.

  Suggestion: Suppose we load extra information in the two CC register(s) for debugging intermediate pipeline stages?

  The next problem, though, is how to deal with the read operand pipeline stage needing the result from the register pipeline.

- **Memory instructions must complete**

  All instructions that enter into the memory module *must* complete. Issued instructions from the prefetch, decode, or operand read stages may or may not complete. Jumps into code must be valid, so that interrupt returns may be valid. All instructions entering the ALU complete.

  This looks to be the simplest approach. While the logic may be difficult, this appears to be the only re-entrant approach.

  A `new_pc` flag will be high anytime the PC changes in an unpredictable way (i.e., it doesn't increment). This includes jumps as well as interrupts and interrupt returns. Whenever this flag may go high, memory operations and ALU operations will stall until the result is known. When the flag does go high, anything in the prefetch, decode, and read-op stage will be invalidated.

2.13 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 should be obvious. If the prefetch cache doesn't have the instruction in memory, the entire pipeline must stall until enough of the prefetch cache is loaded to support the next instruction.

- While waiting for the pipeline to load following any taken branch, jump, return from interrupt or switch to interrupt context (6 clocks)

  If the PC suddenly changes, the pipeline is subsequently cleared and needs to be reloaded. Given that there are five stages to the pipeline, that accounts for five of the six delay clocks. The last clock is lost in the prefetch stage which needs at least one clock with a valid PC before it can produce a new output. Hence, six clocks will always be lost anytime the pipeline needs to be cleared.

- 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.

- When writing to the CC or PC Register

  1. `OPCODE RA,PC` *Ex: a branch opcode*
  2. *(stall, even if jump not taken)*
  3. `OPCODE RA,RB`

  Since branches take place in the writeback stage, the Zip CPU will stall the pipeline for one clock anytime there may be a possible jump. This prevents an instruction from executing a memory access after the jump but before the jump is recognized.

- When reading from the CC register after setting the flags

  1. `ALUOP RA,RB`
  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 after the writeback instruction is determined. Trying to then place these into the input for one of the operands 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 . . .).

- When waiting for a memory read operation to complete

  1. `LOD address,RA`
  2. *(multiple stalls, bus dependent, 7 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 especially true of a load instruction, which will write its operand back to the register file. Store instructions are different, since they can be busy with no impact on later ALU write back operations. Hence, only loads stall the pipeline.

  This also assumes that the memory being accessed is a single cycle memory. Slower memories, such as the Quad SPI flash, will take longer–perhaps even as long as fourty clocks. During this time the CPU and the external bus will be busy, and unable to do anything else.

- Memory operation followed by a memory operation

  1. `STO address,RA`
  2. *(multiple stalls, bus dependent, 7 clocks best)*
  3. `LOD address,RB`
  4. *(multiple stalls, bus dependent, 7 clocks best)*

  In this case, the LOD instruction cannot start until the STALL is finished. With proper scheduling, it is possible to do something in the ALU while the STO is busy, but otherwise this pipeline will stall waiting for it to complete.

  Note that even though the Wishbone bus can support pipelined accesses at one access per clock, only the prefetch stage can take advantage of this. Load and Store instructions are stuck at one wishbone cycle per 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'b0) 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 interrupt line which may come from an external interrupt controller, 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.

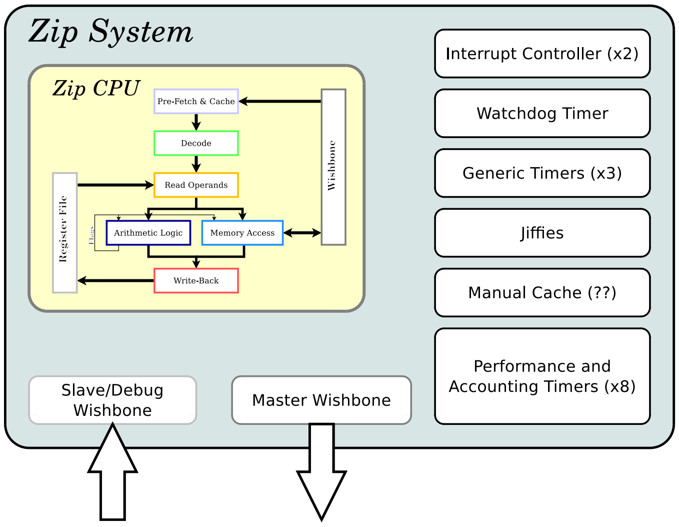

Figure 3.1: Zip System Peripherals

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 $2^{31}-1$, 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 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 sorted, and the next alarm in terms of Jiffies should be written to the register.

3.6 Manual Cache

The manual cache is an experimental setting that may not remain with the Zip CPU for very long. It is designed to facilitate running from FLASH or ROM memory, although the pipeline prefetch cache really makes this need obsolete. The manual cache works by copying data from a wishbone address (range) into the cache register, and then by making that memory available as memory to the Zip System. It is a *manual cache* because the processor must first specify what memory to copy, and then once copied the processor can only access the cache memory by the cache memory location. There is no transparency. It is perhaps best described as a combination DMA controller and local memory.

Worse, this cache is likely going to be removed from the ZipSystem. Having used the ZipSystem now for some time, I have yet to find a need or use for the manual cache. I will likely replace this peripheral with a proper DMA controller.

4. Operation

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 showin in Tbl. 5.2.

Name	Address	Width	Access	Description
PIC	`0xc0000000`	32	R/W	Primary Interrupt Controller
WDT	`0xc0000001`	32	R/W	Watchdog Timer
CCHE	`0xc0000002`	32	R/W	Manual Cache Controller
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

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

6. Wishbone Datasheet

The Zip System supports two wishbone accesses, 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	
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 `ERR` nor the `RETRY` wires have been implemented. What this means is that the CPU is currently unable to detect a bus error condition, and so may stall indefinitely (hang) should it choose to access a value not on the bus, or a peripheral that is not yet properly configured.

Description	Specification	
Revision level of wishbone	WB B4 spec	
Type of interface	Master, Read/Write, single cycle or pipelined	
Address Width	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	
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`

Table 6.2: Wishbone Datasheet for the CPU as Master

7. Clocks

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

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

Table 7.1: List of Clocks

8. I/O Ports