Schwap CPU Design Documentation

Charlie Fenoglio, Alexander Hirschfeld, Andrew McKee, and Wesley Van Pelt

Winter 2015/2016

1 Registers

There are a total of 76 16-bit registers; 12 are fixed and 64 (spilt into 16 groups of 4) ”schwapable” registers. Some registers have alias names, see Section 6.3.1 for a list.

1.1 Register Names and Descriptions

Name	Number	Description	Saved Across Call?
$z0	0	The Value 0†	-
$a0	1	Assembler Temporary 0	No
$a1	2	Assembler Temporary 1	No
$pc	3	Program Counter‡	Yes
$sp	4	Stack Pointer	Yes
$ra	5	Return Address	Yes
$s0 - $s1	6 - 7	User Saved Temporaries	Yes
$t0 - $t3	8 - 11	User Temporaries	No
$h0 - $h3	12 - 15	Schwap	-

†See Section 6.1.1-1 for details
‡See Section 6.1.1-2 for details

1.2 Schwap Registers

The ”schwap” registers are registers that appear to be swapped using a command. There is no data movement when schwapping, it only changes which registers the $h0 - $h3 refer to. There are 8 groups the user can use for general purpose and 8 reserved groups.

1.2.1 Schwap Group Numbers, Descriptions, and Uses

Group Number	Uses	Saved Across Call?
0 - 3	User Temporaries	No
4 - 7	User Saved Temporaries	Yes
8	Arguments 0 - 3	No
9	Return Values 0 - 3	No
10 - 14	Reserved For Future Use†	-
15	Go to the last used group	-

†See Section 6.1.2-1 for details

2 Instructions

All instructions are 16-bits. The destination register is also used as a source unless otherwise noted. All offsets are bit shifted left by 1 since all instructions are 2 bytes long.

2.1 Instruction Types and Bit Layouts

Instructions can be manually translated by putting the bits for each of the components of the instructions in the places listed by the diagrams for each type. The OP codes, function codes, and types can by found on the ”Core Instructions Summary” (2.2.1) table. The destination and source are register numbers, which can be found under the ”Register Names and Descriptions” (1.1) table. Schwap group numbers can be found under the ”Schwap Group Numbers, Descriptions, and Uses” (1.2.1) table. The active schwap group is not preserved over a function call. See Section 6.2.1 for notes on the types and layouts.

2.1.1 A-Type

OP Code	Destination	Source	Func. Code
15 12	11 8	7 4	3 0

Immediate
15 0

Used for all ALU operations. It consists of a 4-bit OP code, 4-bit destination, 4-bit source, and a 4-bit function code. If the instruction has an immediate, it is inserted as the next instruction.

2.1.2 B-Type

OP Code	R0	R1	Offset
15 12	11 8	7 4	3 0

If it is being used for branching it consists of a 4-bit OP code, 4-bit 1st source (R0), 4-bit 2nd source (R1), and a 4-bit (unsigned) offset. If it is being used for reading from memory it consists of a 4-bit OP code, 4-bit destination (R0 not used as a source), 4-bit source (R1), and a 4-bit (unsigned) offset. If it is being used for writing to memory it consists of a 4-bit OP code, 4-bit source (R0), 4-bit destination (R1), and a 4-bit (unsigned) offset.

2.1.3 H-Type

OP Code		Group
15 12		3 0

Used for schwapping and sudo. It consists of a 4-bit OP code, 8 unused bits, and a 4-bit schwap group number or sudo use case.

2.1.4 J-Type

Used for jumping. It consists of a 4-bit OP code, 4-bit source, and an 8-bit (signed) offset.

OP Code	Source	Offset
15 12	11 8	7 0

2.2 Core Instructions

Some instructions have alias names, see Section 6.3.2 for a list. [dest], [src], [src0], [src1] all refer to a register in the register file, for example $t0. ”NAM” stands for the name of the instruction (for example, ”and”). ”OP” stands for whatever the op would be (for example, ”&”).

2.2.1 A-Type

Function Code	Name	Description
0x0	and	Bitwise ands 2 values
0x1	orr	Bitwise ors 2 values
0x2	xor	Bitwise xors 2 values
0x3	not	Bitwise nots the first value
0x4	tsc	Converts a number to 2's compliment
0x5	slt	Set less than
0x6	sgt	Set greater than
0x7	sll	Left logical bit shift
0x8	srl	Right logical bit shift
0x9	sra	Right arithmetic bit shift
0xA	add	Adds 2 values
0xB	sub	Subtracts 2 values
0xF	cpy	Copies the value in one register to another

All A-Type instructions, except for not, tsc, slt, and cpy, follow the syntax in the first section of the table below. Those three can be found in the next sections.

OP Code	Syntax	Meaning	Description
0x0	NAM [dest] [src]	dest = dest OP src	OPs the values in registers [dest] and [src]
0x1	NAM [dest] [src] [immediate]	src = immediate dest = dest OP src	Loads the immediate into the register [src] and then OPs the values in registers [dest] and [src]
	NAM [dest] [immediate]	dest = dest OP immediate	OPs the immediate and the value in the register [dest]
0x0	not [dest]	dest = ∼dest	Bitwise nots the value in the register [dest]
0x1	not [immediate]	dest = ∼immediate	Bitwise nots the immediate and puts the result in the register [dest]
0x0	tsc [dest]	dest = ∼dest + 1	Converts the value in the register [dest] to 2's compliment
0x1	tsc [dest] [src] [immediate]	src = immediate dest = ∼src + 1	Loads the immediate into the register [src] and then converts the value in register [src] to 2's compliment and stores into [dest]
	tsc [dest] [immediate]	dest = ∼immediate + 1	Converts the immediate to 2's compliment and stores into the register [dest]
0x0	slt [dest] [src]	dest = (dest < src) ? 1 : 0	If [dest] < [src], then [dest] gets set to 1 If [dest] ≥ [src], then [dest] gets set to 0
0x1	slt [dest] [src] [immediate]	src = immediate dest = (dest < src) ? 1 : 0	Loads the immediate into the register [src] then If [dest] < [src], then [dest] gets set to 1 If [dest] ≥ [src], then [dest] gets set to 0
	slt [dest] [immediate]	dest = (dest < immediate) ? 1 : 0	If [dest] < [immediate], then [dest] gets set to 1 If [dest] ≥ [immediate], then [dest] gets set to 0
0x0	cpy [dest] [src]	dest = src	Copies the value the in register [src] into [dest]
0x1	cpy [dest] [immediate]	dest = immediate	Loads the immediate into the register [dest]

2.2.2 B-Type

OP Code	Name	Description
0x2	beq	Branches if the 2 values are equal
0x3	bne	Branches if the 2 values are not equal
0x4	bgt	Branches if value0 > value1
0x5	blt	Branches if value0 < value1
0x7	r	Reads the value in memory into a register
0x8	w	Writes the value in a register into memory

The four different types of branches all follow the same syntax, ”bnh” represents any branch name and ”***” represents the condition. They will become pseudo instructions iff branching up, or down more than 16 instructions. Read and write each have their own syntaxes. [offset] is not a register, it is an immediate.

Syntax	Meaning	Description
bnh [src0] [src1] label	if(src0 *** src1) goto label	If [src0] *** [src1], branch to label
r [dest] [offset]([src])	dest = Mem[src + offset<<1]	Reads the data in the address of [src] + [offset] in memory into [dest]
w [offset]([dest]) [src]	Mem[dest + offset¡¡1] = src	Writes the data in the address of [dest] + [offset] in memory from [src]

2.2.3 H-Type

OP Code	Name	Syntax	Description
0xE	rsh	rsh [group]	Changes the schwap group number to [group], these numbers can be found in the table in 1.2.1
0xF	sudo	sudo [code]	Sames as syscall in MIPS

2.2.4 J-Type

OP Code	Name	Syntax	Meaning	Description
0x6	jr	jr [offset]([dest])	pc = dest + offset<<1	Jumps to the instruction at the address in [dest] + [offset]

2.3 Pseudo Instructions

There are two types of pseudo instructions. One are instructions which are always pseudo instructions, the other are sometimes pseudo depending on the conditions. Some instructions have alias names, see Section 6.3.2 for a list.

2.3.1 Always Pseudo Instructions

Name	Syntax	Actual Code	Description
j	j label	cpy $a0 [label pc] jr 0($a0)	Jumps to the instruction at label
jal	jal label	cpy $ra $pc j [label]	Stores the return address and then jumps to the label
bge	bge [src0] [src1] label	cpy $a0 [src0] slt $a0 [src1] beq $a0 $z0 label	If [src0] $\geq$ [src1], branch to label
ble	ble [src0] [src1] label	cpy $a0 [src1] slt $a0 [src0] beq $a0 $z0 label	If [src0] $\leq$ [src1], branch to label

2.3.2 Conditional Pseudo Instructions

Name	Syntax	Actual Code	Condition
beq	beq [src0] [src1] label	bnq [src0] [src1] Next j label Next:	Branching up or branching down more than 16 instructions
bne	bne [src0] [src1] label	beq [src0] [src1] Next j label Next:	Branching up or branching down more than 16 instructions
bgt	bgt [src0] [src1] label	blt [src0] [src1] Next j label Next:	Branching up or branching down more than 16 instructions
blt	blt [src0] [src1] label	bgt [src0] [src1] Next j label Next:	Branching up or branching down more than 16 instructions

3 RTL and Datapath

3.1 Components

3.1.1 Single 16-bit Register

I/O	Name	Size
In	in	16
Out	out	16
Control	write	1

Used as:	
IR	Stores the 16-bit instruction that comes from memory
NextInst	Stores the next 16-bit instruction that comes from memory
PC	Program Counter
R0	Stores the value that comes out of reg0
R1	Stores the value that comes out of reg1
ALUout	Stores the value that comes out of the ALU
PCtemp	Stores the value that comes out of ALUout for PC if needed
MemRead	Same as IR, but is used for values

3.1.2 Single 4-bit Register

I/O	Name	Size
In	in	4
Out	out	4
Control	writable	1

Used as:	
Hreg	Stores IR[3:0]
SchLatch	Stores schwap group number

3.1.3 SE<<1

Sign extends and then shifts it to the left 1

I/O	Name	Size
In	in	*
Out	out	*

3.1.4 16-bit Adder

I/O	Name	Size	Description
In	A	16	First input
In	B	16	Second input
Out	R	16	Result

3.1.5 ALU

I/O	Name	Size	Description
In	A	16	First input
In	B	16	Second input
Out	R	16	Result
Out	zero	1	If result is 0
Out	overflow	1	If there is overflow
Control	op	4	Operation code

3.1.6 Register File

I/O	Name	Size	Description
In	regID0	4	ID for first register
In	regID1	4	ID for second register
In	ImmIn0	16	Data to write to regID1
Out	regOut0	16	Data from regID0
Out	regOut1	16	Data from regID1
Control	SchLatch	4	The schwap group number to use
Control	RegWrite	1	If data can be written to regID1

3.1.7 Main Memory

I/O	Name	Size	Description
In	Addr0	16	Address for data
In	Addr1	16	Address for data
In	Store	16	Stores data at Addr0
Out	Out0	16	Data at Addr0
Out	Out1	16	Data at Addr1
Control	memRead	1	Set to 1 when data is to be read
Control	memWrite	1	Set to 1 when data is to be written

3.1.8 Other Control Signals Not Listed

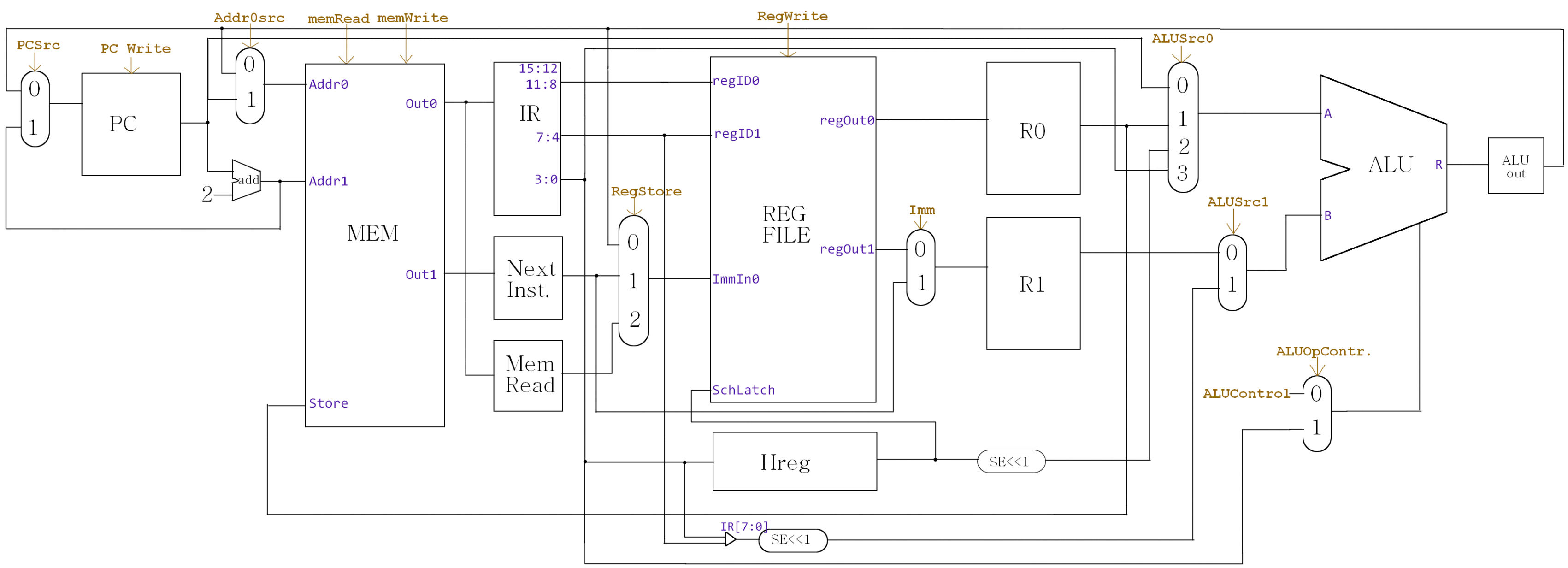
PCSrc: Where PC will get it's new value from
PCWrite: Set to 1 if the PC should write the data on its input
Addr0src: Where Addr0 will be getting its input from
RegStore: Where ImmIn0 will be getting its input from
Imm: If R1 should have an immediate loaded into it
ALUSrc0: Where the first input of the ALU will be coming from
ALUSrc1: Where the second input of the ALU will be coming from
ALUOpContr: If the function code or ALUContr will decide the ALU op
ALUContr: ALU op from the control unit

3.2 Summary Charts

3.2.1 RTL

Step	A-Type	B-Type			H-Type		J-Type
		Branches	Read	Write	Schwap	Sudo	
Get instruction	IR ::= MEM[PC] NextInst ::= MEM[PC+2] PC ::= PC + 2						
Decode instruction and get stuff from registers	R0 ::= reg#IR[11:8] if(IR[3:0]==0x1) {R1 ::= NextInst; PC ::= PC + 2} else {R1 ::= reg#IR[7:4]} ALUout ::= PC + SE(IR[3:0]) Hreg ::= IR[3:0]						
Do computation	ALUout ::= R0 aluop R1	if(R0 aluop R1 == 0) PC ::= ALUout	ALUout ::= R1 + SE(Hreg<<1)		SchLatch ::= Hreg	Change on Code#	PC ::= R0 + SE(IR[7:0]<<1)
Output	reg#IR[8:11] ::= ALUout		MemRead ::= MEM[ALUout]	MEM[ALUout] ::= R0			
Output 2			reg#IR[8:11] ::= MemRead				

3.2.2 Datapath



3.3 Unit Tests and Implementation

3.3.1 Single 16-bit Registers

Implementation: Use Micah's 16 bit register

Tests:
1. Pass in a 16-bit value with writing enabled ⇒ The whole value should be stored
2. Pass in a 16-bit value with writing disabled ⇒ The none of the value should be stored

3.3.2 Single 4-bit Registers

Implementation: Use Micah's 16 bit register and change it to only hold 4 bits

Tests:
1. Pass in a 4-bit value with writing enabled ⇒ The whole value should be stored
2. Pass in a 4-bit value with writing disabled ⇒ The none of the value should be stored

3.3.3 SE<<1

Implementation: Take the input wires and move them over by one to the outputs (and setting the rightmost bit to 0). Split the last wire to make as many more other output bits that are required.

Tests:
1. Pass in a value which has a sign bit of 0 ⇒ The value should be shifted to the left by one and all new bits should be 0
2. Pass in a value which has a sign bit of 1 ⇒ The value should be shifted to the left by one and all new bits should be 1

3.3.4 16-bit Adder

Implementation: Use 4 4-bit carry look ahead adders connected together like a ripple adder

Tests:
1. Pass in 0 and 0 ⇒ The result should be 0
2. Pass in 0xFFFF and 0x1 ⇒ The result should be 0
3. Pass in 0x1 and 0xFFFF ⇒ The result should be 0
4. Pass in 0xFFFF and 0xFFFF ⇒ The result should be 0xFFFE
5. Pass in any other 2 values ⇒ The result should be the 2 inputs added together

3.3.5 ALU

Implementation: Use the 16-bit adder above

Tests: Note: Any time the ALU result is 0, the "zero" output should be 1, all other times it should be 0

and:
1. Pass in 0 and 0 ⇒ The result should be 0
2. Pass in 0xFFFF and 0xFFFF ⇒ The result should be 0xFFFF
3. Pass in 0 and 0xFFFF ⇒ The result should be 0
4. Pass in any other 2 values ⇒ The result should be the 2 inputs anded together

orr:
1. Pass in 0 and 0 ⇒ The result should be 0
2. Pass in 0xFFFF and 0xFFFF ⇒ The result should be 0xFFFF
3. Pass in 0 and 0xFFFF ⇒ The result should be 0xFFFF
4. Pass in any other 2 values ⇒ The result should be the 2 inputs orred together

xor:
1. Pass in 0 and 0 ⇒ The result should be 0

2. Pass in 0xFFFF and 0xFFFF ⇒ The result should be 0
3. Pass in 0 and 0xFFFF ⇒ The result should be 0xFFFF
4. Pass in any other 2 values ⇒ The result should be the 2 inputs xorred together

not:
1. Pass in 0 ⇒ The result should be 0xFFFF
2. Pass in 0xFFFF ⇒ The result should be 0
3. Pass in any other value ⇒ The result should be the input notted

tsc:
1. Pass in 0 ⇒ The result should be 0
2. Pass in 0xFFFF ⇒ The result should be 0x1
3. Pass in any other value ⇒ The result should be the input converted to 2's compliment

slt:
1. Pass in 0 and 0 ⇒ The result should be 0
2. Pass in 0xFFFF and 0xFFFF ⇒ The result should be 0
3. Pass in 0xFFFF and 0 ⇒ The result should be 0x1
4. Pass in 0 and 0xFFFF ⇒ The result should be 0
5. Pass in 2 values such that the first is smaller than the second ⇒ The result should be 0
6. Pass in 2 values such that the second is smaller than the first ⇒ The result should be 0x1

sll:
1. Pass in 0 and 0 ⇒ The result should be 0
2. Pass in 0xFFFF and 0 ⇒ The result should be 0xFFFF
3. Pass in 0xFFFF and 0x1 ⇒ The result should be 0xFFFE
4. Pass in 0xFFFF and 0xF ⇒ The result should be 0x8000
5. Pass in 0 and 0x1 ⇒ The result should be 0
6. Pass in 0 and 0xF ⇒ The result should be 0
7. Pass in any other 16-bit value for the first and 4-bit value for the second ⇒ The result should be the first input shifted to the left by the amount of the second

srl:
1. Pass in 0 and 0 ⇒ The result should be 0
2. Pass in 0xFFFF and 0 ⇒ The result should be 0xFFFF
3. Pass in 0xFFFF and 0x1 ⇒ The result should be 0x7FFF
4. Pass in 0xFFFF and 0xF ⇒ The result should be 0x1
5. Pass in 0 and 0x1 ⇒ The result should be 0
6. Pass in 0 and 0xF ⇒ The result should be 0
7. Pass in any other 16-bit value for the first and 4-bit value for the second ⇒ The result should be the first input shifted to the right by the amount of the second and the new spots should be filled with 0's

sra:
1. Pass in 0 and 0 ⇒ The result should be 0
2. Pass in 0xFFFF and 0 ⇒ The result should be 0xFFFF
3. Pass in 0xFFFF and 0x1 ⇒ The result should be 0xFFFF
4. Pass in 0xFFFF and 0xF ⇒ The result should be 0xFFFF
5. Pass in 0 and 0x1 ⇒ The result should be 0
6. Pass in 0 and 0xF ⇒ The result should be 0
7. Pass in 0x7FFF and 0x1 ⇒ The result should be 0x3FFF
8. Pass in 0x7FFF and 0xE ⇒ The result should be 0x1
9. Pass in 0x7FFF and 0xF ⇒ The result should be 0
10. Pass in any other 16-bit value for the first and 4-bit value for the second ⇒ The result should be the first input shifted to the right by the amount of the second and the new spots should be filled with the leading digit

add:
1. Pass in 0 and 0 ⇒ The result should be 0, overflow should be 0
2. Pass in 0xFFFF and 0x1 ⇒ The result should be 0, overflow should be 0
3. Pass in 0x1 and 0xFFFF ⇒ The result should be 0, overflow should be 0

4. Pass in 0xFFFF and 0xFFFF ⇒ The result should be 0xFFFE, overflow should be 0
5. Pass in 0x7FFF and 0x1 ⇒ The result should be 0x8000, overflow should be 1
6. Pass in 0x7FFF and 0x7FFF ⇒ The result should be 0xFFFE, overflow should be 1
7. Pass in any other 2 values ⇒ The result should be the 2 inputs added together, overflow should be 1 if there is 2's compliment overflow

sub:
1. Pass in 0 and 0 ⇒ The result should be 0, overflow should be 0
2. Pass in 0xFFFF and 0x1 ⇒ The result should be 0xFFFE, overflow should be 0
3. Pass in 0x1 and 0xFFFF ⇒ The result should be 0x10, overflow should be 0
4. Pass in 0xFFFF and 0xFFFF ⇒ The result should be 0, overflow should be 0
5. Pass in 0x8000 and 0x1 ⇒ The result should be 0x7FFF, overflow should be 1
6. Pass in 0x8000 and 0x7FFF ⇒ The result should be 0x1, overflow should be 1
7. Pass in any other 2 values ⇒ The result should be the 2 inputs subtracted, overflow should be 1 if there is 2's compliment overflow

cpy:
1. Pass in 0 ⇒ The result should be 0
2. Pass in 0xFFFF ⇒ The result should be 0xFFFF
3. Pass in any other value ⇒ The result should the value

3.3.6 Register File

Implementation: Create one 16-bit register hard-wired to 0, 11 normal 16-bit registers, and 4 registers which go to a schwapbox.

Tests:
1. Pass in 2 register IDs between 0 and 11 into the to regID inputs ⇒ The result should be the values in the given registers (regOut0 has the value from regID0 and regOut1 has the value regID1)
2. Pass in 2 register IDs between 12 and 15 into the to regID inputs ⇒ The result should be the values in the given registers (regOut0 has the value from regID0 and regOut1 has the value regID1) and use the correct schwap group
3. Pass in a number between 0 and 14 to SchLatch ⇒ The data coming out of registers 12 - 15 should be from the schwap group number given to SchLatch
4. Pass in 15 to SchLatch ⇒ The data coming out of registers 12 - 15 should be from the previous schwap group number given to SchLatch (if it was 2, and then changed to 5, after giving SchLatch 15 the group number should be 2 again)
5. Pass in a register ID using regID1 between 0 and 11 with a value to ImmIn0 and RegWrite set to 1 ⇒ The result should be stored in the register given on regID1, and regID0 should be ignored
6. Pass in a register ID using regID1 between 12 and 15 with a value to ImmIn0 and RegWrite set to 1 ⇒ The result should be stored in the register given on regID1 in the correct schwap group, and regID0 should be ignored

3.3.7 Main Memory

Implementation: Use a Xilnix created memory with 2 address inputs, 2 outputs, and at least 1 write data input

Tests:
1. Pass in 2 valid memory addresses and set the memRead to 1 ⇒ The data in at those memory addresses should be output
2. Pass in data to Store and a valid memory address and set memWrite to 1 ⇒ The the data from Store should have been written to memory
3. Pass in data to Store and a valid memory address and set memWrite to 0 ⇒ The the data from Store should not have been written to memory

3.4 Integration Tests

3.4.1 Get instruction

1. Give PC a memory address with an instruction
2. Toggle memRead and the instruction at the given address, as well as the next instruction, should now be in IR and NextInst respectively
3. PC should have been incremented by 2 so the process can be repeated

3.4.2 Decode instruction and get stuff from registers

1. IR and NextInst should be populated with a current instruction and its next instruction respectively
2. R0 should get the value from the register pointed to by IR[11:8]
3. If IR[3:0] is 0x1 then R1 should get the value in NextInst and PC should be incremented by 2, if IR[3:0] is not 0x1 it should get the value in the register pointed to by IR[7:4]
4. ALUout should get PC added to IR[15:12] which has been sign extended and shifted left by 1
5. Hreg should get IR[3:0]

3.4.3 Do computation

A-Type:
1. R0 and R1 should get some values, ALUop should be set to a valid ALU op code
2. ALUout should now get R0 and R1 combined in the way the function/ALU op code intended
3. This should be repeated for every valid ALU op code

Branches:
1. R0 and R1 should get some values, ALUop should be set to the subtract ALU op code
2. If ALUout is 0 then PC should now get ALUout

Read/Write:
1. R1 and Hreg should be given values
2. ALUout should now contain R1 added to Hreg which has been sign extended and shifted left by 1

Schwap:
1. Hreg should be set to a valid schwap group number
2. SchLatch should now be the value in Hreg

Sudo:
1. Will vary based on code number

J-Type:
1. R0 and IR[7:0] should be set to values
2. PC should now be changed to R0 + IR[7:0] which has been sign extended and shifted to the left by 1

3.4.4 Output

A-Type:
1. R0 should get a value, IR[11:8] should be set to a register ID
2. The value in ALUout should now be in the register which IR[11:8] was pointing to

Read:
1. ALUout should be set to a valid memory address
2. MemRead should now be set to the value in memory at the address from ALUout

Write:
1. MemRead should get a value, IR[11:8] should be set to a register ID
2. The value which was in MemRead should now be in the register which IR[11:8] was pointing to

3.4.5 Output 2

Read:
1. ALUout should be set to a valid memory address
2. MemRead should now be set to the value in memory at the address from ALUout

3.5 Plan Tests

3.5.1 A-Type

No Immediate:
1. Get this instruction
2. Increase PC by 2 for the next instruction
3. R0 and R1 should be loaded with the values from the first and second register in the instruction
4. The ALUout should have the value of the two values combined using the function code that was passed in
5. That value should be in the first specified register for the instruction

With Immediate:
1. Get this and the next instruction
2. Increase PC by 2 for the next instruction
3. R0 should be loaded with the value from the first register in the instruction and R1 should have the immediate which was in NextInst, PC should have been increased by 2 again
4. The ALUout should have the value of the two values combined using the function code that was passed in
5. That value should be in the first specified register for the instruction, the immediate should also have been stored in the second register from the instruction if it was given

3.5.2 B-Type

Branches:
1. Get this instruction
2. Increase PC by 2 for the next instruction
3. R0 and R1 should be loaded with the values from the first and second register in the instruction
4. The ALUout should have the new PC value for use if the branch is supposed to branch
5. ALUout should be transferred to PCtemp, the combination of R0 and R1 should be in ALUout
6. If ALUout is 0 PC should have been changed to what is in PCtemp

Read:
1. Get this instruction
2. Increase PC by 2 for the next instruction
3. R0 and R1 should be loaded with the values from the first and second register in the instruction, Hreg should have the offset from the instruction

4. The ALUout should have the value of what's in R1 added to the sign extended, shifted to the left by one value in Hreg
5. ALUout should be passed into main memory and the value at that address should be in MemRead
6. The first register specified in the instruction should have the value from memory

Write:
1. Get this instruction
2. Increase PC by 2 for the next instruction
3. R0 and R1 should be loaded with the values from the first and second register in the instruction, Hreg should have the offset from the instruction
4. The ALUout should have the value of what's in R1 added to the sign extended, shifted to the left by one value in Hreg
5. ALUout should be passed into memory as well as the value in R0, the value in R0 should be in memory at that address

3.5.3 H-Type

Schwap:
1. Get this instruction
2. Increase PC by 2 for the next instruction
3. Hreg should get the schwap group number
4. SchLatch should now be set to the new schwap group number

Sudo:
1. Get this instruction
2. Increase PC by 2 for the next instruction
3. Hreg should get the sudo code number
4. The correct action should now be performed based on the code number

3.5.4 J-Type

jr:
1. Get this instruction
2. Increase PC by 2 for the next instruction
3. PC should be changed to the value in R0 added to the sign extended, shifted to the left by one value in IR[7:0]

4 Assembler and Coding Practices

4.1 Assembler

1. The order of the parameters for "r" must be flipped. The hardware expects the memory location in the 2nd register, not the first as in the syntax.

4.2 Code

1. Avoid branching up and more than 16 instructions down. The hardware implementation of branching limits branching to only going down a maximum of 16 instructions, but the assembler will convert these to a combination of a branch and jump.

2. Schwap groups are not preserved across calls.

5 Examples

5.1 Basic Use Examples

5.1.1 Loading an immediate into a register

```
cpy $t0 32        # Loads 32 into t0
```

5.1.2 Making a Procedure Call

```
rsh 8             # Switch to arguments schwap
cpy $h0 $t0       # Put argument0 in
cpy $h1 $s1       # Put argument1 in
# Store any wanted temporaries somewhere
jal Call
rsh 9             # Switch to return values schwap
cpy $s0 $h0       # Copy the return values out
```

5.1.3 Iteration and Conditionals

This is an example of which will iterate over 4 array elements in memory and add 32 to each of them. It will stop repeating after the 4 elements using beq.

```
        # There is a base memory address for an array in memory at s0
        cpy $t0 8
        cpy $t1 $z0
loop:
        r   $t2 0($s0)
        add $t2 32
        w   0($s0) $t2
        add $t1 2
        beq $t0 $t1 loop
```

5.2 relPrime and gcd Implementation

5.2.1 Assembly

```
RelPrime:
        rsh     8                   #set schwap
        cpy     $s2 $ra             #save $ra
        cpy     $s0 $h0             #copy n out of schwap
        cpy     $s1 0x2             #load 2 to m
        rsh     8                   #set schwap to args
While:
        cpy     $h0 $s0             #set a0 to n
        cpy     $h1 $s1             #set a1 to m
        jal     GCD                 #call GCD
        rsh     9                   #set schwap
        cpy     $t0 0x1             #load immediate 0x1 to t0
        bne     $h0 $t0 Done        #branch to done if r0 != 1
        add     $s1 0x1             #add 1 to m
        j       While               #jump to the start of the loop
Done:
        rsh     9                   #load return registers
        cpy     $h0 $s1             #set r0 to m
        j       $s2 0               #return to the previous function
```

```
GCD:
          rsh     8                  #schwap to argument register
Base:
          bne     $h0 $z0 GMain      #a!=0 go to GMain
          cpy     $t0 $h1            #copy h1 to t0 for RSH
          rsh     9                  #schwap to return registers
          cpy     $h0 $t0            #load t0 to r1
          j       $ra 0              #return
GMain:
          beq     $h1 $z0 Exit       #jump to exit if b is zero
          bgt     $h0 $h1 If         #jump to If if a>b
Else:
          sub     $h1 $h0            #else: b=b-a
          j       GMain              #loop
If:
          sub     $h0 $h1            #if: a=a-b
          j       GMain              #loop
Exit:
          cpy     $t0 $h0            #copy h0 to t0 for rsh schwap
          rsh     9                  #make sure we're in the right spot
          cpy     $h0 $t0            #copy t0 to h0
          j       $ra                #return
```

5.3 Machine Code

Machine code for all of the examples will be included once the assembler is complete.

RelPrime		GCD	
PC	Hex	PC	Hex
00	3009	42	3009
02	063F	44	5C05
04	04CF	46	085F
06	150F	48	300A
08	0002	4A	0A8F
0A	3009	4C	2300
0C	0C4F	4E	4B05
0E	0D5F	50	1C0F
10	031F	52	0072
12	300F	54	2C00
14	1C0F	56	301F
16	0042	58	6CD7
18	2C00	5A	0DC2
1A	301F	5C	300F
1C	300A	5E	1C0F
1E	180F	60	004E
20	0001	62	2C00
22	4C85	64	301F
24	1C0F	66	0CD2
26	000A	68	300F
28	2C00	6A	1C0F
2A	301F	6C	004E
2C	1500	6E	2C00
2E	0001	70	301F
30	300F	72	08CF
34	1C0F	74	300A
36	000C	76	0C8F
38	2C00	78	2300
3A	301F		
3C	300A		
3E	0C5F		
40	2600		

6 Notes

6.1 Registers

6.1.1 Non-Schwappable

1. $z0 is reset on the rising edge of each CPU cycle, so it can be used for cycle-temporary storage.
2. The value in $pc should always be what the current instruction address is +2.

6.1.2 Schwappable

1. Possible uses for the reserved for future use groups:

Group Number	ID	Use
10	0	The constant 1
	1	The constant -1
	2	User set constant0
	3	User set constant1
11	0 - 3	I/O for devices 0 - 3
12	0 - 3	Syscall values 0 - 3
13	0 - 3	Kernel
14	0	Exception Cause
	1	Exception Status
	2	EPC
	3	Exception Temporary

6.2 Instructions

6.2.1 Types and Layouts

1. More of the types could be combined, but they will run faster if they are not.

6.3 Alias names

6.3.1 Registers

$z0: $0, $00, $zz, $zero

6.3.2 Instructions

orr: or

bne: bnq