Highridge Processor

Instruction Set Architecture Reference

Rev. 1.1

November, 2015

Alec E. Selfridge

Joshua H. Hightower

Table of Contents

This page intentionally left blank.

I. Introduction

The Highridge CPU is a non-pipelined, 32-bit MIPS-based processor with an extension allowing vector operations. The "baseline" architecture defines a set of elementary instructions common to many processors, including arithmetic and logical operations. These MIPS instructions are broken into three main categories: R-Type, I-Type, and J-Type (illustrated in figure A on the following page). R-Types include many arithmetic and logical instructions like ADD, SUB, shifts, AND, OR, etc. I-Type instructions utilize 16-bit immediate values (part of the instruction itself) to perform branches and some arithmetic instructions using the immediate. Finally, there are only two J-Type instructions, JUMP and JUMP-AND-LINK. These alter program flow similar to branches but utilize a 25-bit immediate value.

Enhancements to the baseline include a handful of extra instructions and vector operations. Vector-based instructions use an additional instruction format: V-Type. This is merely an extension of R-Types with some slight variation. The Vector Operations Unit allows for four, 8-bit values or two, 16-bit values to be operated on simultaneously. In later editions, byte and halfword loads and stores will be added to better accommodate these instructions. With this version, however, the Vector Operations Unit shares the same register space as it's traditional 32-bit counterpart.

The additional enhancements are aimed at speeding up, and simplifying, program execution. Thus, instructions like Bit Set/Clear and Branch-if-Divide-by-Zero are implemented. Since this processor version is not pipelined, these additional instructions may help increase CPI and throughput by eliminating one or more extra clock cycles compared to some other instructions.

II. Instruction Set Architecture

This page intentionally left blank.

A. Abbreviations & Formats

Below is a table of commonly used abbreviations throughout this document:

Abbreviation	Description
rs	Source Register/ Operand 1
rt	Source Register/Operand 2
rd	Destination Register
PC	Program Counter
IR	Instruction Register
VOU	Vector Operations Unit
V-Type	Vector-Type Instruction
QUAD8	Single 32-bit value divided into 4, 8-bit values
DUAL16	Single 32-bit value divided into 2, 16-bit values
GPR	General Purpose Register
ISR	Interrupt Service Routine

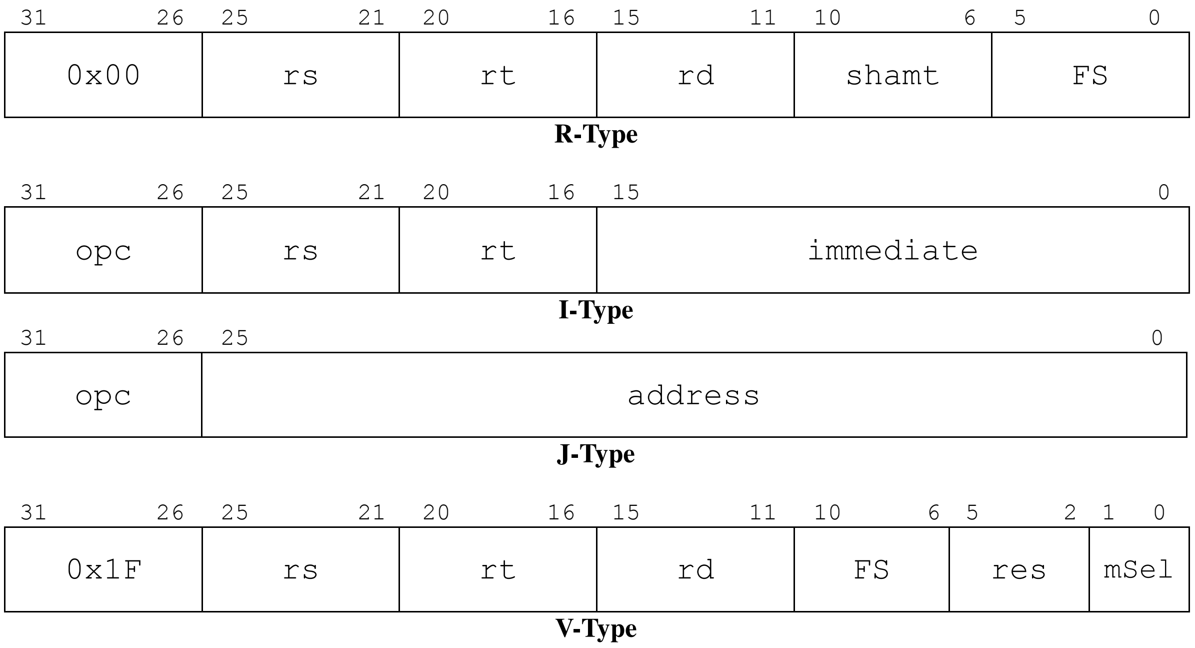
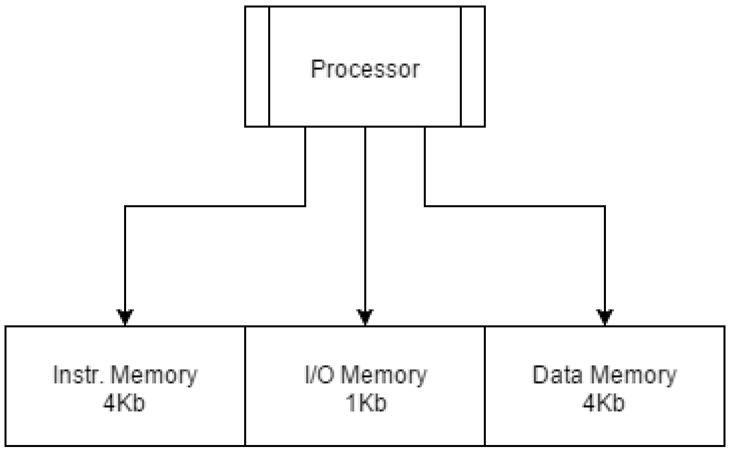


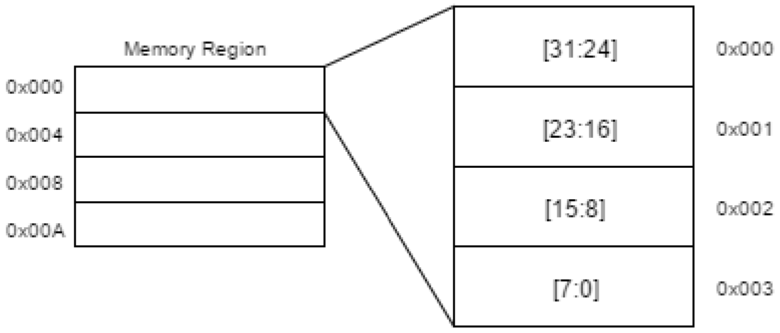
Figure A

B. Memory Organization

Utilizing a Harvard architecture, the Highridge processor has separate instruction and data memories. Each memory region has a 32-bit address space although only 4Kbyte of memory is available in each. With this scheme, the processor can execute instructions and read/write data in parallel, thus increasing performance. Additionally, there is 1Kbyte of I/O memory with one, maskable, interrupt line.

- 4Gb total address space
  - 4Kb used
- Byte-addressable
  - Word-aligned
- Big-endian





C. Data Types

The 32-bit MIPS processor has a total of two data types: signed and unsigned integers. No instruction can exceed a bigger number than 32-bits wide, but can use less than 32-bits if the instruction allows it to happen. With multiply and divide, they both yield a 64-bit result; in this case, the result is split into two separate 32-bit registers HI and LO and those are used to store the result into the register file.

A VOU allows for two additional data types: QUAD8/DUAL16. Their instructions follow conventional 32-bit MIPS format and yield either four, 8-bit results or two, 16-bit results. In the case of a multiply or divide a 64-bit product or quotient/remainder pair is split into two 32-bit halves, like traditional 32-bit operand results.

1) 32-bit signed integer
- Flags updated: N, V, C, and Z.
- Used in all instructions, excluding addu, sltu, subu, and sltiu.

2) 32-bit unsigned integer
- Flags updated: V, C, and Z.
- Used in addu, subu, sltu, and sltiu instructions.

3) QUAD8 unsigned integer
-No flags
-ADD8, SUB8, AND8, OR8, NOT8, XOR8, MUL8, DIV8, VMFHI, VMFLO

4) DUAL16 unsigned integer
-No flags
-ADD16, SUB16, AND16, OR16, NOT16, XOR16, MUL16, DIV16, VMFHI, VMFLO

D. Addressing Modes

1. Immediate (16-bit and 26 bit)
   - I-Type instructions have a 16-bit immediate signed value attached to the lower 16-bits of the instruction.
   - If they pass the test, branch instructions take the 16-bit value, sign extend the MSB to make it 32-bit, and shift the value to the left by 2. The final value is added to the PC(now PC+4) and executes instructions like normal.
   - Jump instructions have a 26 bit attached to the lower 26 bits of the instruction. In order to jump to a 32-bit address with 26 bits, the 26 bit value is shifted left by 2 and the current upper 4 bits of the PC(now PC+4) are added before the 28 bits, making a 32-bit value.

PC ← PC + sign_ext{IR[15:0], :00})

Example 1

2. Register
   - R-Type instructions uses register addressing mode to complete operations. The registers pointed at by rs and rt perform the instruction and store the result into the register pointed at by rd.

ADD $rd, $rs, $rt , (R[$rd] ←$rs + $rt)

Example 2

3. Register Indirect
   - Register indirect is used for the load and store instructions. Both instructions are written in the same format as follows:

     rt, offset(rs)

   - Load takes the address from rs, adds the16-bit offset, and loads rt with the memory at that location.
   - Store takes the value in rt and stores it at the memory location pointed at by rs plus the 16-bit offset.

LW $rt, 4($rs) , (R[$rt] ← M[4+$rs])

Example 3

E. Processor Register Set

There are a total of 32, 32-bit registers available to the user. Six of these are typically reserved for special outlined in the upcoming section. The remaining registers can be used at the programmer's discretion but recommended usage is outlined below:

Name & Number		Description
0	$zero	Constant 0
1	$at	Assembler Temporary
2-3	$v0-$v1	Function Values (results)
4-7	$a0-$a3	Arguments
8-14	$t0-$t7	Temporaries
15-23	$s0-$s7	Saved Temporaries
24-25	$t8-$t9:	Temporaries
26-27	$k0-$k1	Reserved for Kernel
28	$gp	Global Pointer
29	$sp	Stack Pointer
30	$fp	Frame Pointer
31	$ra	Return Address

Special Registers:

$zero: hardwired 0.

$at: used by the assembler to expand pseudoinstructions or calculations.

$gp: not commonly used and can serve as $s8. Otherwise points to middle of memory block.

$sp: points to the top of the stack.

$fp: points to the base of the stack frame. The parameters passed to a subroutine remain at a constant spot relative to the frame pointer.

$ra: holds the address of the next instruction after a CALL is finished to restore program flow.

Processor Register Set

Program Counter:

The 32-bit program counter (PC) is available to the user and contains the address of the next instruction to be fetched. It can be modified through **normal program flow**, **directly** by the programmer, or via **I/J-type** instructions. Although, it is usually best to only modify the PC via normal flow or I/J-type instructions.

Flags Register:

The status of the current program is contained in the flags register. Any instruction that uses flags for conditional execution will check one of the six bits here. The flags are set appropriately based on the instruction. For example, an ADD instruction would update the C, V, N, Z flags but an SLL instruction would only update the N and Z flags.

5	4	3	2	1	0
DBZ	IE	C	V	N	Z

Flags:

DBZ: divide-by-zero flag. Set if the divisor operand is zero.

IE: interrupt enable.

C: carry flag. Set if the result of an instruction caused a carry-out.

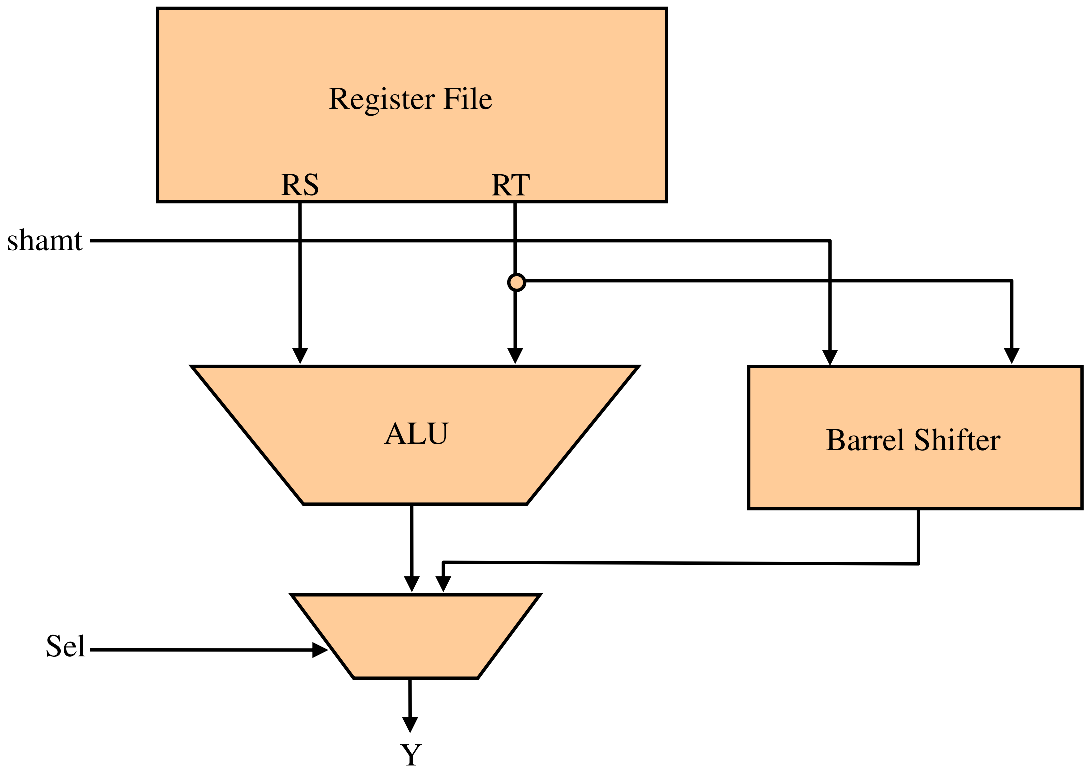
V: overflow flag. Set if the result of an instruction caused overflow; that is, the range of operand values was exceeded or an arithmetic error occurred.

N: negative flag. Set if the result of an instruction is negative.

Z: zero flag. Set if the result of an instruction was zero.

F. Shifts

The baseline MIPS architecture does not require the ability to shift multiple times in either direction, but such an operation is often necessary. Thus a 32-bit barrel shifter was implemented to allow shifts up to 32 places in one clock cycle. This provides an efficient mechanism to do multiplication and division by powers of two as well as variable logical shifts. Additionally, this barrel shifter can perform right-rotates, a feature not normally found on a MIPS processor. The shifter unit operates in parallel to the main ALU. This concept is shown in the following figure:



Some signals and components removed for clarity

G. Vector Operations Unit (VOU)

In order to more effectively deal with unsigned bytes and halfwords, a Vector Operations Unit is used. It can perform several common operations on four, 8-bit values (QUAD8) or two 16-bit values (DUAL16). These operations update no flags and are listed in the following table. Figures 1.0a and 1.0b show how logical and arithmetic instructions are executed. Results of multiplications and divisions are stored in a distinct HILO register so as to prevent losing the result of a 32-bit multiply or divide instruction. Figure 1.1a and 1.1b show how multiplication and division are performed.

8-bit Operations	**16-bit Operations**
add	add
sub	sub
and	and
or	or
not	not
xor	xor
multiply	multiply
divide	divide

**both modes use VMFHI and VMFLO*

The VOU uses the following instruction format (V-Type):

31 26	25 21	20 16	15 11	10 6	5 2	1 0
"E Key"	rs	rt	rd	FS	res	mSel

Instruction Segment	**Meaning**
E Key	Hardcoded to 0x1F - Designates V-Type
rs	Source Register/Operand 1
rt	Source Register/Operand 2
rd	Destination Register
FS	Function Select
res	Reserved
mSel	Mode Select - 8/16-bit

Vector Operations Unit (VOU)

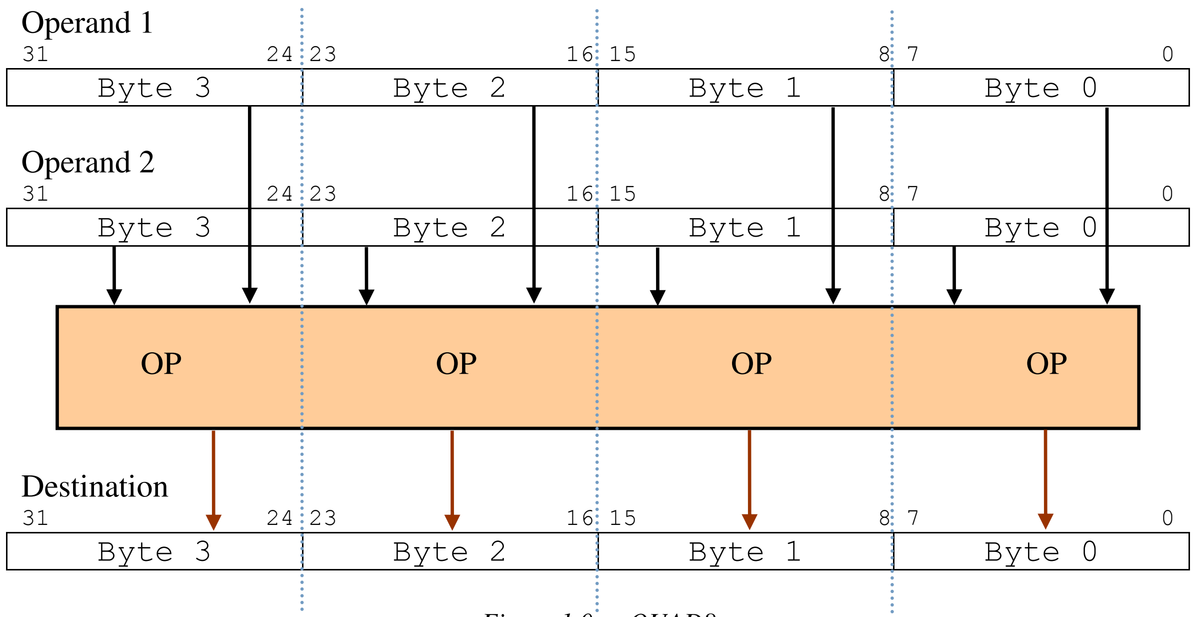


Figure 1.0a - QUAD8

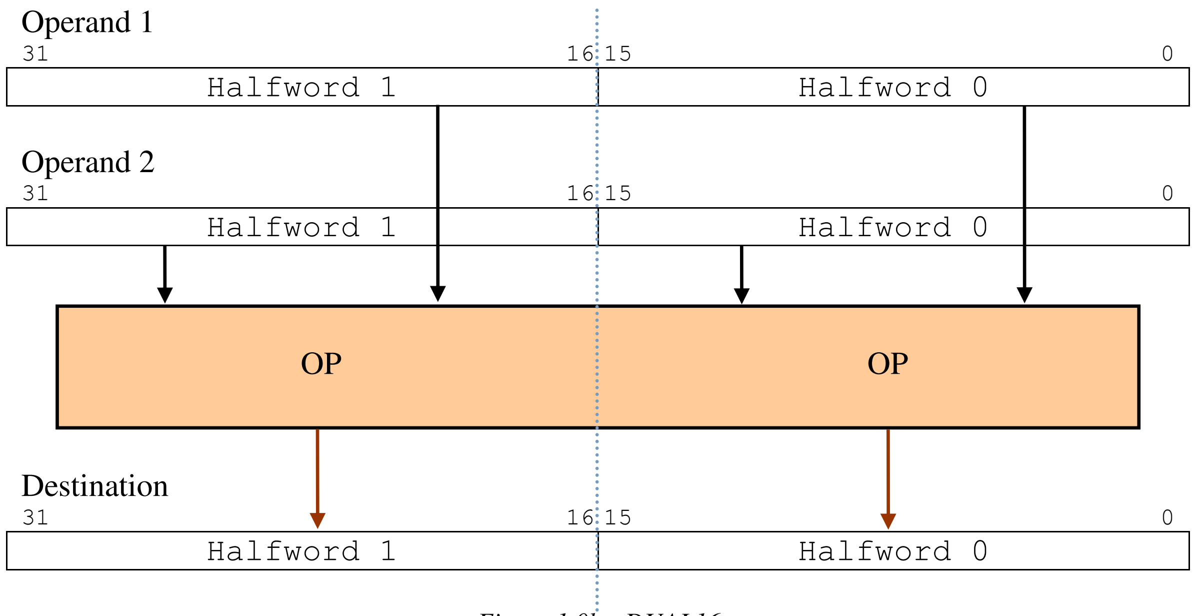


Figure 1.0b - DUAL16

Vector Operations Unit (VOU)

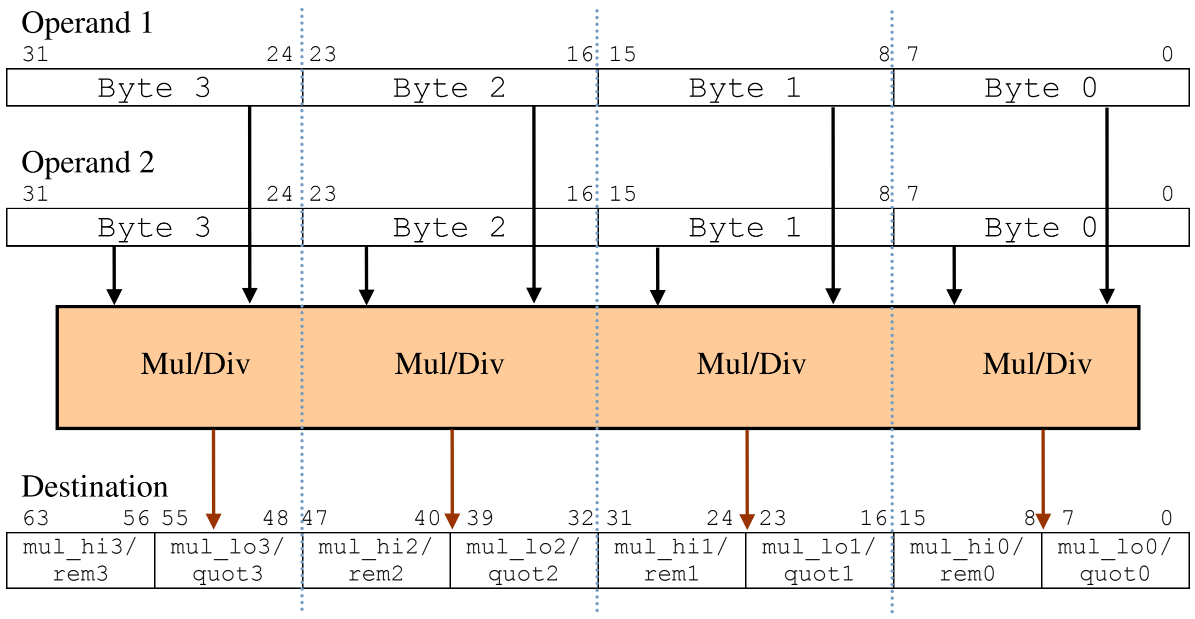


Figure 1.1a - QUAD8 MUL/DIV

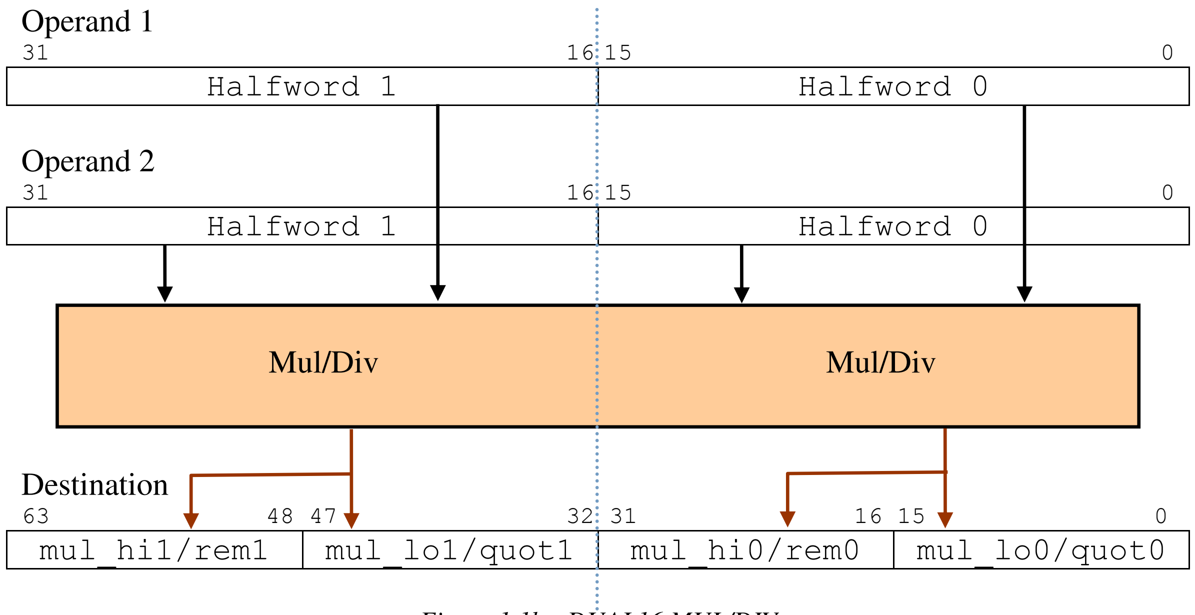


Figure 1.1b - DUAL16 MUL/DIV

H. Instruction Set Architecture

This page intentionally left blank.

1. Instruction Set - R-Type

R-Type

ADD Addition

R-type

31 26	25 21	20 16	15 11	10 6	5 0
0x00	rs	rt	rd	0	0x20

Format:

add rd, rs, rt

Purpose:

To add two 32-bit values and store the result into a 32-bit register.

Description:

rd ← rs + rt

The 32-bit value stored in register rs is added with the 32-bit register rt and the result is stored into the 32-bit register specified by rd. All four flags (N, C, Z, V) are checked and updated accordingly.

Limitations:

None.

Examples:

Assembly	**Machine Code**
`ADD R3, R1, R2`	`000000_00001_00010_00011_00000_100000`
`ADD R10, R13, R6`	`000000_01101_00110_01010_00000_100000`

Operand Values	**Result**

R1 = 0x12345678
R2 = 0x12345678
	0x2468ACF0
	C=0,N=0,V=0,Z=0

R13 = 0x5A5A5A5A
R6 = 0xA5A5A5A6
	0x00000001
	C=1,N=0,V=0,Z=0

ADDU Unsigned Addition

R-type

31 26	25 21	20 16	15 11	10 6	5 0
0x00	rs	rt	rd	0	0x21

Format:

addu rd, rs, rt

Purpose:

To add two 32-bit values and store the result into a 32-bit register.

Description:

rd ← rs + rt

The 32-bit value stored in register rs is added with the 32-bit register rt and the result is stored into the 32-bit register specified by rd. The difference between this operation and regular addition is the N flag does not matter; the other three flags are still set accordingly.

Limitations:

None.

Examples:

Assembly	**Machine Code**
`ADDU R3, R1, R2`	`000000_00001_00010_00011_00000_100001`
`ADDU R10, R13, R6`	`000000_01101_00110_01010_00000_100001`

Operand Values	**Result**

`R1 = 0x12345678`
`R2 = 0x12345678`
	`0x2468ACF0`
	`C=0,N=x,V=0,Z=0`

`R13 = 0x5A5A5A5A`
`R6 = 0xA5A5A5A6`
	`0x00000001`
	`C=1,N=x,V=0,Z=0`

AND Logical AND

R-type

31 26	25 21	20 16	15 11	10 6	5 0
0x00	rs	rt	rd	0	0x24

Format:

and rd, rs, rt

Purpose:

To logically and two 32-bit values are and store the result into a 32-bit register.

Description:

rd ← rs & rt

The 32-bit value stored in register rs is logically and'ed with the 32-bit register rt and the result is stored into the 32-bit register specified by rd. Two flags are updated with this operation, N and Z.

Limitations:

None.

Examples:

Assembly	**Machine Code**
`AND R3, R1, R2`	`000000_00001_00010_00011_00000_100100`
`AND R10, R13, R6`	`000000_01101_00110_01010_00000_100100`

Operand Values	**Result**

R1 = 0xABCDEF01
R2 = 0xA3CDCF09
	0xA3CDCF01
	C=x,N=1,V=x,Z=0

R13 = 0x00000000
R6 = 0xA5A5A5A5
	0x00000000
	C=x,N=0,V=x,Z=1

DIV

Divide

R-type

31 26	25 21	20 16	15 11	10 6	5 0
0x00	rs	rt	0	0	0x1A

Format:

div rs, rt

Purpose:

To divide two 32-bit registers and store the result in two separate 32-bit registers.

Description:

rs / rt
lo ← quotient
hi ← remainder

The 32-bit value stored in register rs is divided with the 32-bit register rt and the 64 bit result is stored into two separate 32-bit registers, *lo* and *hi*. The *lo* register receives the lower 32-bit of the result (the quotient) and the *hi* register receives the upper 32-bits of the result(the remainder). All four flags (N, C, Z, V) are checked and updated accordingly.

Limitations:

If the value stored in rt is 0, the result of the divide will be undefined.

Examples:

Assembly	**Machine Code**
DIV R3, R1	000000_00011_00001_00000_00000_011010
DIV R10, R13	000000_01010_01101_00000_00000_011010

Operand Values	**Result**

R3 = 0x0000001F
R1 = 0x00000002
	0x000000010000000F
	C=0,N=0,V=0,Z=0

R10 = 0x00000019
R13 = 0x000003e8
	LO=0x000061A8
	HI=0xFFFF9E58
	C=0,N=0,V=0,Z=0

JR — Jump Register

R-type

31 26	25 21	20 16	15 11	10 6	5 0
0x00	rs	0	0	0	0x08

Format:

jr rs

Purpose:

Unconditionally jump to the address stored inside register rs.

Description:

PC ← rs

The PC obtains the value that rs is holding and begins executing instructions at that point inside the memory. No flags are updated with this operation.

Limitations:

The lower byte must be a 0, 4, 8, or C in order for the jump to work properly.

Examples:

Assembly	**Machine Code**
JR R31	000000_11111_00000_00000_00000_001000
JR R16	000000_10000_00000_00000_00000_001000

Operand Values	**Result**
R31 = 0x00800020	PC=0x00800020
R16 = 0x04040404	PC=0x04040404

MFHI — Move from Hi

R-type

31 26	25 21	20 16	15 11	10 6	5 0
0x00	0	0	rd	0	0x10

Format:

mfhi rd

Purpose:

To move the value stored in *hi* after a multiply or divide into the destination register rd.

Description:

rd ← *hi*

The rd register receives and stores the value that is currently stored in register *hi*. No flags are updated with this operation.

Limitations:

None.

Examples:

Assembly	**Machine Code**
`MFHI R6`	`000000_00000_00000_00000_00110_010000`
`MFHI R8`	`000000_00000_00000_00000_01000_010000`

Operand Values	**Result**
`HI = 0xF3FB0891`	`0xF3FB0891`
`HI = 0x00000002`	`x00000002`

MFLO Move from Lo

R-type

31 26	25 21	20 16	15 11	10 6	5 0
0x00	0	0	rd	0	0x12

Format:

mflo rd

Purpose:

To move the value stored in *lo* after a multiply or divide into the destination register rd.

Description:

rd ← *lo*

The rd register receives and stores the value that is currently stored in register *lo*. No flags are updated with this operation.

Limitations:

None.

Examples:

Assembly	Machine Code
MFLO R6	000000_00000_00000_00000_00110_010010
MFLO R8	000000_00000_00000_00000_01000_010010

Operand Values	Result
LO = 0xF3FB0891	0xF3FB0891
LO = 0x00000002	0x00000002

MULT Multiply

R-type

31 26	25 21	20 16	15 11	10 6	5 0
0x00	rs	rt	0	0	0x18

Format:

mult rs, rt

Purpose:

To multiply two 32-bit registers and store the result in two separate 32-bit registers.

Description:

rs * rt
lo ← lower 32-bits
hi ← upper 32-bits

The 32-bit value stored in register rs is multiplied with the 32-bit register rt and the 64 bit result is stored into two separate 32-bit registers, *lo* and *hi*. The *lo* register receives the lower 32-bit of the result and the *hi* register receives the upper 32-bits of the result. All four flags (N, C, Z, V) are checked and updated accordingly.

Limitations:

None.

Examples:

Assembly	Machine Code
MULT R3, R1	000000_00011_00001_00000_00000_011000
MULT R10, R13	000000_01010_01101_00000_00000_011000

Operand Values	Result

R3 = 0x0000001F
R1 = 0x00000002
	0x000000000000003E
	C=0,N=0,V=0,Z=0

R10 = 0xFFFFEDBA
R13 = 0x77777777
	0xFFFFFFFFEEEEF776
	C=0,N=1,V=0,Z=0

NOR Logical NOT OR

R-type

31 26	25 21	20 16	15 11	10 6	5 0
0x00	rs	rt	rd	0	0x27

Format:

nor rd, rs, rt

Purpose:

To NOR two 32-bit registers and store the result into a 32-bit register.

Description:

rd ← ~(rs | rt)

The 32-bit value stored in register rs is NOR'ed with the 32-bit register rt and the result is written to register rd. The N and Z flags are the only flags to be updated.

Limitations:

None.

Examples:

Assembly	**Machine Code**
`NOR R3, R1, R2`	`000000_00001_00010_00011_00000_100111`
`NOR R10, R13, R6`	`000000_01101_00110_01010_00000_100111`

Operand Values	**Result**

R1 = 0x5A5A3C3C
R2 = 0x5A5A3C3C
	0xA5A5C3C3
	C=x,N=1,V=x,Z=0

R13 = 0x00000000
R6 = 0xA5A5A5A5
	0x5A5A5A5A
	C=x,N=0,V=x,Z=0

OR — Logical OR

R-type

31 … 26	25 … 21	20 … 16	15 … 11	10 … 6	5 … 0
0x00	rs	rt	rd	0	0x25

Format:

or rd, rs, rt

Purpose:

To OR two 32-bit registers and store the result into a 32-bit register.

Description:

rd ← rs | rt

The 32-bit value stored in register rs is OR'ed with the 32-bit register rt and the result is written to register rd. The N and Z flags are the only flags to be updated.

Limitations:

None.

Examples:

Assembly	Machine Code
OR R3, R1, R2	000000_00001_00010_00011_00000_100101
OR R10, R13, R6	000000_01101_00110_01010_00000_100101

Operand Values	Result

R1 = 0x5A5A3C3C
R2 = 0x50503030
	0x5A5A3C3C
	C=x,N=0,V=x,Z=0

R13 = 0x00000000
R6 = 0xA5A5A5A5
	0xA5A5A5A5
	C=x,N=1,V=x,Z=0

RB — Reverse Bit Order

R-type

31 – 26	25 – 21	20 – 16	15 – 11	10 – 6	5 – 0
0x00	rs	0	rd	0	0x2C

Format:

rb rd, rs

Purpose:

To reverse the bit order of a 32-bit register and store the result into a 32-bit register.

Description:

rd ← rs(reversed bits)

The 32 bits stored in register rs are reversed, meaning the MSB becomes the LSB, and so on. The sign bit is not preserved. No flags are affected.

Limitations:

None.

Examples:

Assembly	Machine Code
`RB R1, R9`	`000000_01001_00000_00001_00000_101100`
`RB R11, R5`	`000000_00101_00000_01011_00000_101100`

Operand Values	Result
R9 = 0x13DBAE87	0xE175DBC8
R5 = 0x08DA3297	0xE94C5B10

RBO Reverse Byte Order

R-type

31 26	25 21	20 16	15 11	10 6	5 0
0x00	rs	0	rd	0	0x2D

Format:

rbo rd, rs

Purpose:

To reverse the endianness of a 32-bit register and store the result into a 32-bit register.

Description:

rd ← {rs[7:0], rs[15:8], rs[23:16], rs[31:24]}

The 32-bit value stored in register rs is partitioned into four bytes, MSB aligned. The order of these bytes is then swapped, putting the lowest byte in the highest byte position and so forth.

Limitations:

None.

Examples:

Assembly	**Machine Code**
RBO R15, R12	000000_01100_00000_01111_00000_101101
RBO R3, R10	000000_01010_00000_00011_00000_101101

Operand Values	**Result**
R12 = 0xABCD0123	0x2301CDAB
R10 = 0xFE1263DC	0xDC6312FE

RR Rotate Right

R-type

31 26	25 21	20 16	15 11	10 6	5 0
0x00	0	rt	rd	shamt	0x34

Format:

rb rd, rt, shamt

Purpose:

To rotate a 32-bit register by the amount of times designated in the shamt field and store the result into a 32-bit register.

Description:

rd ← rt RR shamt

The 32-bit value stored in register rt is rotated to the right the amount specified by shamt and stored into the register specified by rd. The N and Z flags are updated accordingly.

Limitations:

None.

Examples:

Assembly	**Machine Code**
RR R3, R2, 2	000000_00010_00000_00011_00010_110100
RR R9, R6, 8	000000_00110_00000_01001_01000_110100

Operand Values	**Result**
R2 = 0x1873D96E	0x861CF65B
R6 = 0xFFE7834D	0x4DFFE783

SLL Shift Left Logical

R-type

31 26	25 21	20 16	15 11	10 6	5 0
0x00	0	rt	rd	shamt	0x00

Format:

sll rd, rt, shamt

Purpose:

To logically shift left register rt by the amount specified by shamt and store the result into register rd.

Description:

rd ← rt << shamt

The 32-bit value stored in register rt is shifted left by the amount specified by shamt and stored into the register specified by rd. The only flag not updated is V; all other flags(C, N, Z) are checked and updated accordingly.

Limitations:

None.

Examples:

Assembly	**Machine Code**
`SLL R3, R1, 29`	`000000_00001_00000_00011_11101_000000`
`SLL R10, R13, 4`	`000000_01101_00000_01010_00100_000000`

Operand Values	**Result**
`R1 = 0xDC653ADF`	`0xE0000000` `C=0,N=1,V=x,Z=0`
`R13 = 0x9630ABD9`	`0x630ABD90` `C=0,N=0,V=x,Z=0`

SLT Set Less Than

R-type

31 26	25 21	20 16	15 11	10 6	5 0
0x00	rs	rt	rd	0	0x2A

Format:

slt rd, rs, rt

Purpose:

To check is rs is less than rt and set rd accordingly.

Description:

```
if( rs < rt)
    rd ←1
Else
    rd ← 0
```

The 32-bit value stored in register rs is checked with the 32-bit value stored in register rt. If rs is less than rt, rd is set to 1. If this is not the case, rd is set to 0. The flags updated with this operation is N and Z.

Limitations:

If rs and rt and equal to each other, rd is set to a 0.

Examples:

Assembly	**Machine Code**
SLT R3, R1, R2	000000_00001_00010_00011_00000_101010
SLT R10, R13, R6	000000_01101_00110_01010_00000_101010

Operand Values	**Result**

R1 = 0xF0000000
R2 = 0x70000000
	R3=1
	C=x,N=x,V=x,Z=0

R13 = 0x00000000
R6 = 0xA5A5A5A5
	R10=0
	C=x,N=x,V=x,Z=1

SLTU Set Less Than (Unsigned)

R-type

31 26	25 21	20 16	15 11	10 6	5 0
0x00	rs	rt	rd	0	0x2B

Format:

slt rd, rs, rt

Purpose:

To check is rs is less than rt and set rd accordingly.

Description:

```
If (rs < rt)
    rd ← 1
Else
    rd ← 0
```

The 32-bit value stored in register rs is checked with the 32-bit value stored in register rt. If rs is less than rt, rd is set to 1. If this is not the case, rd is set to 0. The difference in this instruction is the compare looks at the 32-bit values as unsigned. The flag updated with this operation is Z since N is a don't care.

Limitations:

If rs and rt and equal to each other, rd is set to a 0.

Examples:

Assembly	Machine Code
SLT R3, R1, R2	000000_00001_00010_00011_00000_101011
SLT R10, R13, R6	000000_01101_00110_01010_00000_101011

Operand Values	Result

R1 = 0xF0000000
R2 = 0x70000000
	R3=0
	C=x,N=x,V=x,Z=1

R13 = 0x00000000
R6 = 0xA5A5A5A5
	R10=0
	C=x,N=x,V=x,Z=1

SRA

Shift Right Arithmetic

R-type

31 26	25 21	20 16	15 11	10 6	5 0
0x00	0	rt	rd	shamt	0x03

Format:

sra rd, rt, shamt

Purpose:

To arithmetically shift right register rt by the amount specified by shamt and store the result into register rd.

Description:

rd ← rt >> shamt

The 32-bit value stored in register rt is shifted right by the amount specified by shamt and stored into the register specified by rd. With arithmetic shifts, the MSB is kept the same throughout the shifts and the V flag is checked to make sure it stays. All other flags(N, Z, C) are set as well.

Limitations:

None.

Examples:

Assembly	Machine Code
`SRA R3, R1, 6`	`000000_00001_00000_00011_00110_00011`
`SRA R10, R13, 20`	`000000_01101_00000_01010_10100_00011`

Operand Values	Result
R1 = 0xA78CD62D	0xFE9E3358 C=0,N=1,V=x,Z=0
R13 = 0xBC54987A	0xFFFFFBC5 C=0,N=1,V=x,Z=0

SRL

Shift Right Logical

R-type

31 26	25 21	20 16	15 11	10 6	5 0
0x00	0	rt	rd	shamt	0x02

Format:

srl rd, rt, shamt

Purpose:

To logically shift right register rt by the amount specified by shamt and store the result into register rd.

Description:

rd ← rt >> shamt

The 32-bit value stored in register rt is shifted right by the amount specified by shamt and stored into the register specified by rd. The only flag not updated is V; all other flags(C, N, Z) are checked and updated accordingly.

Limitations:

None.

Examples:

Assembly	Machine Code
SRL R3, R1, 5	000000_00001_00000_00011_00101_000010
SRL R10, R13, 12	000000_01101_00000_01010_01100_000010

Operand Values	Result
R1 = 0xA79EDC32	0x053CF6E1 C=0,N=0,V=x,Z=0
R13 = 0xB4D3329F	0x000B4D33 C=0,N=0,V=x,Z=0

SUB Subtraction

R-type

31 26	25 21	20 16	15 11	10 6	5 0
0x00	rs	rt	rd	0	0x22

Format:

sub rd, rs, rt

Purpose:

To subtract two 32-bit values and store the result into a 32-bit register.

Description:

rd ← rs - rt

The 32-bit value stored in register rs is subtracted with the 32-bit register rt and the result is stored into the 32-bit register specified by rd. All four flags (N, C, Z, V) are checked and updated accordingly.

Limitations:

None.

Examples:

Assembly	**Machine Code**
`SUB R3, R1, R2`	`000000_00001_00010_00011_00000_100010`
`SUB R10, R13, R6`	`000000_01101_00110_01010_00000_100010`

Operand Values	**Result**

R1 = 0x12345678
R2 = 0x12345678
	0x00000000
	C=0,N=0,V=0,Z=1

R13 = 0x5A5A5A5A
R6 = 0xA5A5A5A6
	0xB4B4B4B4
	C=1,N=1,V=1,Z=0

SUBU Unsigned Subtraction

R-type

31 26	25 21	20 16	15 11	10 6	5 0
0x00	rs	rt	rd	0	0x23

Format:

subu rd, rs, rt

Purpose:

To subtract two 32-bit values and store the result into a 32-bit register.

Description:

rd ← rs - rt

The 32-bit value stored in register rs is subtracted with the 32-bit register rt and the result is stored into the 32-bit register specified by rd. The difference between this operation and regular subtraction is the N flag does not matter; the other three flags are still set accordingly.

Limitations:

None

Examples:

Assembly	**Machine Code**
SUBU R3, R1, R2	000000_00001_00010_00011_00000_100011
SUBU R10, R13, R6	000000_01101_00110_01010_00000_100011

Operand Values	**Result**

R1 = 0x12345678
R2 = 0x12345678
	0x00000000
	C=0,N=x,V=0,Z=1

R13 = 0x5A5A5A5A
R6 = 0xA5A5A5A6
	0xB4B4B4B4
	C=1,N=x,V=1,Z=0

XOR Exclusive OR

R-type

31 26	25 21	20 16	15 11	10 6	5 0
0x00	rs	rt	rd	0	0x26

Format:

xor rd, rs, rt

Purpose:

To XOR two 32-bit registers and store the result into a 32-bit register.

Description:

rd ← rs ^ rt;

The 32-bit value stored in register rs is XOR'ed with the 32-bit register rt and the result is written to register rd. The N and Z flags are the only flags to be updated.

Limitations:

None.

Examples:

Assembly	**Machine Code**
XOR R3, R1, R2	000000_00001_00010_00011_00000_100110
XOR R10, R13, R6	000000_01101_00110_01010_00000_100110

Operand Values	**Result**

R1 = 0xABCDEF01
R2 = 0xA3CDCF09
	0x08002008
	C=x,N=1,V=x,Z=0

R13 = 0xA0A0A0A0
R6 = 0xA5A5A5A5
	0x05050505
	C=x,N=0,V=x,Z=0

2. Instruction Set - I-Type

I-Type

ADDI Add Immediate

I-type

31 26	25 21	20 16	15 0
0x08	rs	rt	immediate

Format:

addi rt, rs, immediate

Purpose:

To add a signed constant value to a register.

Description:

rt ← rs + immediate

The 16-bit immediate value is added to register *rs* to make a 32-bit result. If the 32-bit 2's complement addition creates an overflow, the V flag is set. The destination register is overwritten either way and not cleared upon overflow.

Limitations:

The immediate value must be within the range -32,768 – 32,767.

Examples:

Assembly	**Machine Code**
`ADDI R2, R1, 5`	`001000_00001_00010_0000000000001010`
`ADDI R3, R0, -1`	`001000_00000_00011_1111111111111111`

Operand Values	**Result**
R1 = 0x12345678	0x1234567D C=x,N=0,V=x,Z=0
R0 = 0x00000000	0xFFFFFFFF C=x,N=1,V=x,Z=0

ANDI AND Immediate

I-type

31 26	25 21	20 16	15 0
0x0C	rs	rt	immediate

Format:

andi rt, rs, immediate

Purpose:

To logically AND an operand with a constant.

Description:

rt ← rs AND immediate

The source operand is combined with a 16-bit zero-extended constant using a bitwise logical AND. The result is placed in the destination register, *rt*.

Limitations:

The immediate value must be within the range -32,768 – 32,767.

Examples:

Assembly	**Machine Code**
`ANDI R2, R1, 8`	`001100_00001_00010_0000000000001000`
`ANDI R3, R0, -1`	`001100_00000_00011_1111111111111111`

Operand Values	**Result**

`R1 = 0x12345675`	`0x12345678`
	`C=0,N=0,V=0,Z=0`

`R0 = 0x00000000`	`0x00000000`
	`C=0,N=0,V=0,Z=0`

BCI

Bit Clear Immediate

I-type

31 26	25 21	20 16	15 0
0x24	rs	rt	immediate

Format:

bci rt, rs, immediate

Purpose:

To clear one bit of a register designated by a constant value.

Description:

rs[imm] = 0

rt ← rs

The 16-bit immediate value is used as an index for register rs and the corresponding bit is set to zero. The result is stored in rd.

Limitations:

The immediate value must be within the range 0 – 31.

Examples:

Assembly	**Machine Code**
`BCI R3, R9, 31`	`100101_01001_00011_0000000000011111`
`BCI R14, R3, 9`	`100101_00011_01110_0000000000001001`

Operand Values	**Result**
`R9 = 0xFDA671DD`	`0x7DA671DD`
`R3 = 0xA4A46320`	`0xA4A46120`

BDF Branch if Divide by Zero False

I-type

31 26	25 21	20 16	15 0
0x11	rs	rt	immediate

Format:

bdf offset

Purpose:

To branch within program execution if the Divide by Zero flag is not set.

Description:

```
If (DBZ == 0)
    PC ← PC + offset
Else
    PC ← PC
```

Checks if a division by zero occurred and loads the PC with PC+offset if false, otherwise the PC is unchanged. The offset is a signed 16-bit value and together with the current PC forms the effective address.

Limitations:

The offset must be within the range -32,768 – 32,767.

Examples:

Assembly	**Machine Code**
BDF pass	010001_00000_00000_0000101001011010
BDF fail	010001_00000_00000_1111111111111111

Operand Values	**Result**

No division by 0	PC=PC+0x00000A5A
	C=0,N=0,V=0,Z=0

Division by 0	PC = PC
	C=0,N=0,V=0,Z=0

BDT Branch if Divide by Zero True

I-type

31 26	25 21	20 16	15 0
0x11	rs	rt	immediate

Format:

bdt rs, rt, offset

Purpose:

To branch within program execution if the Divide by Zero flag is set.

Description:

```
If (DBZ == 1)
    PC ← PC + offset
Else
    PC ← PC
```

Checks if a division by zero occurred and loads the PC with PC+offset if true, otherwise the PC is unchanged. The offset is a signed 16-bit value and together with the current PC forms the effective address.

Limitations:

The offset must be within the range -32,768 – 32,767.

Examples:

Assembly	**Machine Code**
`BDT pass`	`010000_00000_00000_0000101001011010`
`BDT fail`	`010000_00000_00000_1111111111111111`

Operand Values	**Result**

Division by 0	PC=PC+0x00000A5A
	C=0,N=0,V=0,Z=0

No division by 0	PC = PC
	C=0,N=0,V=0,Z=0

BEQ Branch if Equal

I-type

31 26	25 21	20 16	15 0
0x04	rs	rt	immediate

Format:

beq rs, rt, offset

Purpose:

To branch within program execution if both operands are equal.

Description:

```
If (rs == rt)
    PC ← PC + offset
Else
    PC ← PC
```

Checks the equivalency of the operands and loads the PC with PC+offset if true, otherwise the PC is unchanged. The offset is a signed 16-bit value and together with the current PC forms the effective address.

Limitations:

The offset must be within the range -32,768 – 32,767.

Examples:

Assembly	**Machine Code**
`BEQ R1, R2, pass`	`001100_00001_00010_0000101001011010`
`BEQ R0, R3, fail`	`001100_00000_00011_1111111111111111`

Operand Values	**Result**

R1 = 0x12345678
R2 = 0x12345678
	PC=PC+0x00000A5A
	C=0,N=0,V=0,Z=1

R0 = 0x00000000
R3 = 0x00000001
	PC = PC
	C=0,N=0,V=0,Z=0

BGTZ Branch if Greater-than Zero

I-type

31 26	25 21	20 16	15 0
0x07	rs	0	offset

Format:

bgtz rs, offset

Purpose:

To branch within current program execution if the operand is greater than zero.

Description:

```
If (rs > 0)
    PC ← PC + offset
Else
    PC ← PC
```

Checks if the operand's sign bit is set or if the register is zero and loads the PC with PC+offset if both conditions are false, otherwise the PC is unchanged. The offset is a signed 16-bit value and together with the current PC forms the effective address.

Limitations:

The offset must be within the range -32,768 – 32,767.

Examples:

Assembly	**Machine Code**
`BGTZ R2, pass`	`000111_00010_00000_0000101001011010`
`BGTZ R3, fail`	`000111_00011_00000_1111111111111111`

Operand Values	**Result**
`R2 = 0x12345678`	`PC=PC+0x00000A5A` `C=x,N=0,V=x,Z=0`
`R3 = 0x80000000`	`PC = PC` `C=x,N=1,V=x,Z=0`

BLEZ — Branch if Less-than or Equal to Zero

I-type

31 26	25 21	20 16	15 0
0x06	rs	0	offset

Format:

blez rs, offset

Purpose:

To branch within current program execution if the operand is less-than or equal-to zero.

Description:

```
If (rs <= 0)
    PC ← PC + offset
Else
    PC ← PC
```

Checks if the operand's sign bit is set or if the register is zero and loads the PC with PC+offset if true, otherwise the PC is unchanged. The offset is a signed 16-bit value and together with the current PC forms the effective address.

Limitations:

The offset must be within the range -32,768 – 32,767.

Examples:

Assembly	**Machine Code**
`BLEZ R2, pass`	`000110_00010_00000_0000101001011010`
`BLEZ R3, fail`	`000110_00011_00000_1111111111111111`
`BLEZ R0, pass`	`000110_00000_00000_0000101001011010`

Operand Values	**Result**
`R2 = 0x80000000`	`PC=PC+0x00000A5A` `C=x,N=1,V=x,Z=0`
`R3 = 0x12345678`	`PC = PC` `C=x,N=0,V=x,Z=0`
`R0 = 0x00000000`	`PC=PC+0x00000A5A` `C=x,N=0,V=x,Z=1`

BNE Branch if Not Equal

I-type

31 26	25 21	20 16	15 0
0x05	rs	rt	immediate

Format:

bne rs, rt, offset

Purpose:

To branch within current program execution if the operands are unequal.

Description:

```
If (rs != rt)
    PC ← PC + offset
Else
    PC ← PC
```

Checks the equivalency of the operands and loads the PC with PC+offset if false, otherwise the PC is unchanged. The offset is a signed 16-bit value and together with the current PC forms the effective address.

Limitations:

The offset must be within the range -32,768 – 32,767.

Examples:

Assembly	**Machine Code**
BNE R1, R2, pass	001100_00001_00010_0000101001011010
BNE R0, R3, fail	001100_00000_00011_1111111111111111

Operand Values	**Result**

R1 = 0x12345679
R2 = 0x12345678
	PC=PC+0x00000A5A
	C=0,N=0,V=0,Z=0

R0 = 0x00000000
R3 = 0x00000000
	PC = PC
	C=0,N=0,V=0,Z=1

BSI

Bit Set Immediate

I-type

31 26	25 21	20 16	15 0
0x24	rs	rt	immediate

Format:

bsi rt, rs, immediate

Purpose:

To set one bit of a register designated by a constant value.

Description:

rs[imm] = 1

rt ← rs

The 16-bit immediate value is used as an index for register rs and the corresponding bit is set to one. The result is stored in rd.

Limitations:

The immediate value must be within the range 0 – 31.

Examples:

Assembly	**Machine Code**
`BSI R4, R12, 16`	`100100_01100_00100_0000000000010000`
`BSI R10, R2, 25`	`100100_00010_01010_0000000000011001`

Operand Values	**Result**
`R12 = 0xFD7863BC`	`0xFD7963BC`
`R2 = 0xDCAD35A0`	`0xDEAD35A0`

LUI Load Upper Immediate

I-type

31 26	25 21	20 16	15 0
0x1F	rs	rt	immediate

Format:

lui rt, rs, immediate

Purpose:

To load the upper half of a register with a constant.

Description:

rs[31:16] ← immediate
rt ← rs

The upper 16-bits of the operands are replaced by the signed, immediate value. The lower half of the word is cleared with zeros. The result is written to register *rt*.

Limitations:

The immediate value must be within the range -32,768 – 32,767.

Examples:

Assembly	Machine Code
`LUI R2,0xFFFF`	`001111_00000_00010_1111111111111111`
`LUI R3, 0x0001`	`001111_00000_00011_0000000000000001`

Operand Values	Result
R2 = 0x12345678	0xFFFF000 C=x,N=1,V=x,Z=0
R3 = 0x00000000	0x00010000 C=x,N=0,V=x,Z=0

LW Load Word

I-type

31 26	25 21	20 16	15 0
0x23	rs	rt	immediate

Format:

lw rt, immediate(rs)

Purpose:

To load a word from memory into a register.

Description:

rt ← M[immediate(rs)]

The contents referenced by the effective address are sign-extended (if applicable) and loaded into the destination register. The effective address is the 16-bit signed immediate value added to the base register, *rs*.

Limitations:

The effective address must be word aligned.

Examples:

Assembly	**Machine Code**
`LW R2,0(R15)`	`100011_01111_00010_0000000000000000`
`LW R3, 4(R15)`	`100011_01111_00011_0000000000000100`

Operand Values	**Result**
M[R15+0] = 0xF0F0F0F0	0xF0F0F0F0 C=x,N=0,V=x,Z=0
M[R15+4] = 0x0000A543	0x0000A543 C=x,N=0,V=x,Z=0

ORI

OR Immediate

I-type

31 … 26	25 … 21	20 … 16	15 … 0
0x0D	rs	rt	immediate

Format:

ori rt, rs, immediate

Purpose:

To logically OR an operand with a constant.

Description:

rt ← rs OR immediate

The source operand is combined with a 16-bit zero-extended constant using a bitwise logical OR. The result is placed in the destination register, *rt*.

Limitations:

The immediate value must be within the range -32,768 – 32,767.

Examples:

Assembly	**Machine Code**
ORI R2,0xFFFF	001101_00000_00010_1111111111111111
ORI R3, 0x0001	001101_00000_00011_0000000000000001

Operand Values	**Result**
R2 = 0xFFFF1234	0xFFFFFFFF C=x,N=1,V=x,Z=0
R3 = 0x00010000	0x00010001 C=x,N=0,V=x,Z=0

SLTI Set if Less-than Immediate

I-type

31 26	25 21	20 16	15 0
0x0A	rs	rt	immediate

Format:

slti rt, rs, immediate

Purpose:

To set or clear a register if the operand is less than a constant.

Description:

```
If (rs < immediate)
    rt ← 1
Else
    rt ← 0
```

Compares the operand with a constant,16-bit signed value and places a Boolean result in the destination register. A value of (1) is true and (0) is false.

Limitations:

The immediate value must be within the range -32,768 – 32,767.

Examples:

Assembly	**Machine Code**
`SLTI R2, R1, 5`	001010_00001_00010_0000000000000101
`SLTI R3, R0, -1`	001010_00000_00011_1111111111111111

Operand Values	**Result**

R1 = 0x00000004	R2=1
	C=x,N=x,V=x,Z=0

R0 = 0x00000000	R3=0
	C=x,N=x,V=x,Z=1

SLTIU

Set if Less-than Immediate Unsigned

I-type

31 26	25 21	20 16	15 0
0x0B	rs	rt	immediate

Format:

sltiu rt, rs, immediate

Purpose:

To set or clear a register if the operand is less than a constant.

Description:

```
If(rs < immediate)
    rt ← 1
Else
    rt ← 0
```

Compares the operand with a constant, 16-bit unsigned value and places a Boolean result in the destination register. A value of (1) is true and (0) is false.

Limitations:

None.

Examples:

Assembly	Machine Code
`SLTI R2, R1, 0x4040`	`001010_00001_00010_0100000001000000`
`SLTI R3, R0, 0x8000`	`001010_00000_00011_1000000000000000`

Operand Values	Result

`R1 = 0xFFFFFFFF`	`R2=0`
	`C=x,N=x,V=x,Z=1`

`R0 = 0x00000000`	`R3=1`
	`C=x,N=x,V=x,Z=0`

SW Store Word

I-type

31 26	25 21	20 16	15 0
0x2B	rs	rt	immediate

Format:

sw rt, immediate(rs)

Purpose:

To store a word to memory.

Description:

M[immediate(rs)] ← rt

The register designated by *rt* is stored at the memory location referenced by *rs* + immediate. Thus, the signed 16-bit immediate value acts as an offset to the base register, *rs*.

Limitations:

The effective address must be word-aligned.

Examples:

Assembly	**Machine Code**
`SW R2,0(R15)`	`100011_01111_00010_0000000000000000`
`SW R3, 4(R15)`	`100011_01111_00011_0000000000000100`

Operand Values	**Result**
R2=0xF0F0F0F0; M[R15+0] = 0xABCDEF01	M[R15+0] = 0xF0F0F0F0 C=x,N=0,V=x,Z=0
R3=0x0000A543; M[R15+4] = 0xAAAAAAAA	M[R15+4] = 0x0000A543 C=x,N=0,V=x,Z=0

XORI

XOR Immediate

I-type

31 26	25 21	20 16	15 0
0x0E	rs	rt	immediate

Format:

xori rt, rs, immediate

Purpose:

To logically exclusive OR an operand with a constant.

Description:

rt ← rs XOR immediate

The source operand is combined with a 16-bit zero-extended constant using a bitwise logical XOR (exclusive OR). The result is placed in the destination register, *rt*.

Limitations:

The immediate value must be within the range -32,768 – 32,767.

Examples:

Assembly	**Machine Code**
XORI R2, R1, 8	001100_00001_00010_0000000000001000
XORI R3, R0, -1	001100_00000_00011_1111111111111111

Operand Values	**Result**
R1 = 0x12345675	0x12345675 C=x,N=0,V=x,Z=0
R0 = 0x00000000	0xFFFFFFFF C=x,N=1,V=x,Z=0

3. Instruction Set - J-Type

J-Type

J Jump

J-type

31 26	25 0
0x02	address

Format:

j addr

Purpose:

Branches to the address specified.

Description:

PC ← addr

The 26-bit address field is loaded directly into the PC. This is not PC-relative and thus branching occurs within the current execution region.

Jumps to the address specified and continues execution.

Limitations:

None.

Examples:

Assembly	**Machine Code**
`J one`	`100000_00000000000000000000000001`
`J two`	`100000_11110011011110111100000001`

Operand Values	**Result**

`one = 0x0000001`	`PC=PC + one`
	`C=x,N=x,V=x,Z=x`

`two = 0x3CDEF01`	`PC=PC + two`
	`C=x,N=x,V=x,Z=x`

JAL Jump and Link

J-type

31 26	25 0
0x03	address

Format:

jal addr

Purpose:

Procedure call within the current execution space.

Description:

PC ← addr

The 26-bit address field is loaded directly into the PC. This is not PC-relative and thus branching occurs within the current execution region. The return address is placed in $ra which points to the instruction after the jump, like a CALL.

Limitations:

None.

Examples:

Assembly	**Machine Code**
`J one`	`110000_00000000000000000000000001`
`J two`	`110000_11110011011110111100000001`

Operand Values	**Result**

`one = 0x0000001`	`R31 = PC`
	`PC=PC + one`
	`C=x,N=x,V=x,Z=x`

`two = 0x3CDEF01`	`R31 = PC`
	`PC=PC + two`
	`C=x,N=x,V=x,Z=x`

4. Instruction Set - Enhancements

Enhancements

INPUT Load Register from I/O

I/O

31 26	25 21	20 16	15 0
0x1C	rs	rt	immediate

Format:

input rt, immediate(rs)

Purpose:

To load a register with an location in I/O memory.

Description:

rt ← ioM[immediate(rs)]

The register designated by *rt* is loaded with the memory location in I/O memory referenced by *rs* + immediate. Thus, the signed 16-bit immediate value acts as an offset to the base register, *rs*.

Limitations:

The effective address must be word-aligned.

Examples:

Assembly	**Machine Code**
`INPUT R2,0(R15)`	`011100_01111_00010_0000000000000000`
`INPUT R3, 4(R15)`	`011100_01111_00011_0000000000000100`

Operand Values	**Result**
M[R15+0] = 0xABCDEF01	0xABCDEF01
M[R15+4] = 0x98765432	0x98765432

OUTPUT Store Register to I/O

I/O

31 26	25 21	20 16	15 0
0x1D	rs	rt	immediate

Format:

output rt, immediate(rs)

Purpose:

To store a register to a location in I/O memory.

Description:

ioM[immediate(rs)] ← rt

The register designated by *rt* is stored into the memory location in I/O memory referenced by *rs* + immediate. Thus, the signed 16-bit immediate value acts as an offset to the base register, *rs*.

Limitations:

The effective address must be word-aligned.

Examples:

Assembly	**Machine Code**
OUTPUT R2,0(R15)	011101_01111_00010_0000000000000000
OUTPUT R3, 4(R15)	011101_01111_00011_0000000000000100

Operand Values	**Result**
R2 = 0xABCDEF01	M[R15+0] = 0xABCDEF01
R3 = 0x98765432	M[R15+4] = 0x98765432

RETI Return from Interrupt

I/O

31 … 26	25 … 21	20 … 16	15 … 0
0x1E	0	0	immediate

Format:

reti

Purpose:

To resume normal program flow after an ISR was serviced.

Description:

PC ← dM[$sp]

The PC is loaded with the return address; the address of the next instruction to be executed after the call. This is the value previously in the PC before an interrupt occurred. Because of this, it is unwise to manipulate the stack pointer ($sp) in an ISR.

Limitations:

Must be in an ISR when executed.

Example:

Assembly	**Machine Code**
RETI	011110_00000_00000_0000000000000000

Operand Values	**Result**
N/A	PC = dM[$sp]

5. Instruction Set - V-Type

V-Type

ADD8 QUAD8 Addition

V-type

31 26	25 21	20 16	15 11	10 6	5 2	1 0
0x1F	rs	rt	rd	0x02	res	0b01

Format:

add8 rd, rs, rt

Purpose:

To add two 32-bit registers representing four 8-bit values and store the result in a 32-bit register representing four 8-bit results.

Description:

$rd \leftarrow rs + rt$

The 32-bit values in rs and rt are divided into four 8-bit sections aligned by their MSBs. The two operands are added together and the four results are stored in rd. There are no flags to update for this operation.

Limitations:

Carry is ignored.

Examples:

Assembly	**Machine Code**
`ADD8 R3, R1, R2`	`011111_00001_00010_00011_00010_0000_01`
`ADD8 R10, R13, R6`	`011111_01101_00110_01010_00010_0000_01`

Operand Values	**Result**

`R1 = 0x1A2B3C4D`
`R2 = 0x62DE00A1`
	`0x7C093CEE`

`R13 = 0x38DE90CA`
`R6 = 0x1298DBE4`
	`0x4A766BAE`

ADD16 DUAL16 Addition

V-type

31 26	25 21	20 16	15 11	10 6	5 2	1 0
0x1F	rs	rt	rd	0x02	res	0b10

Format:

add16 rd, rs, rt

Purpose:

To add two 32-bit registers representing two 16-bit values and store the result in a 32-bit register representing two 16-bit results.

Description:

rd ← rs + rt

The 32-bit values in rs and rt are divided into two 16-bit sections aligned by their MSBs. The two operands are added together and the two results are stored in rd. There are no flags to update for this operation.

Limitations:

Carry is ignored.

Examples:

Assembly	**Machine Code**
ADD16 R10, R3, R2	011111_00011_00010_01010_00010_000_10
ADD16 R4, R11, R8	011111_01011_01000_00100_00010_0000_10

Operand Values	**Result**

R3 = 0x4A68E320
R2 = 0x4A68EC49
	0x94D0CF69

R11 = 0x10BEAD66
R8 = 0x378AE11C
	0x48488E82

AND8 QUAD8 Logical AND

V-type

31 26	25 21	20 16	15 11	10 6	5 2	1 0
0x1F	rs	rt	rd	0x04	res	0b01

Format:

and8 rd, rs, rt

Purpose:

To logically AND two 32-bit registers representing four 8-bit values and store the result in a 32-bit register representing four 8-bit results.

Description:

rd ← rs & rt

The 32-bit values in rs and rt are divided into four 8-bit sections aligned by their MSBs. The two operands are ANDed and the four results are stored in rd. There are no flags to update for this operation.

Limitations:

None.

Examples:

Assembly	**Machine Code**
AND8 R8, R5, R4	011111_00101_00100_01000_00100_0000_01
AND8 R14, R1, R2	011111_00001_00010_01110_00100_0000_01

Operand Values	**Result**

R5 = 0xBB24EA79
R4 = 0x3412ACD1
	0x3000A851

R1 = 0xFC3794BA
R2 = 0x1298DBE4
	0x101090A0

AND16

DUAL16 Logical AND

V-type

31 26	25 21	20 16	15 11	10 6	5 2	1 0
0x1F	rs	rt	rd	0x04	res	0b10

Format:

and16 rd, rs, rt

Purpose:

To logically AND two 32-bit registers representing two 16-bit values and store the result in a 32-bit register representing two 16-bit results.

Description:

rd ← rs & rt

The 32-bit values in rs and rt are divided into two 16-bit sections aligned by their MSBs. The two operands are ANDed and the two results are stored in rd. There are no flags to update for this operation.

Limitations:

None.

Examples:

Assembly	**Machine Code**
`AND16 R3, R9, R1`	`011111_01001_00001_00011_00100_0000_10`
`AND16 R7, R3, R12`	`011111_00011_01100_00111_00100_0000_10`

Operand Values	**Result**

R9 = 0x45FF62AD
R1 = 0xEA3C2915
	0x403C2005

R3 = 0x88DED731
R12 = 0x156ADE62
	0x004AD620

DIV8 QUAD8 Division

V-type

31 26	25 21	20 16	15 11	10 6	5 2	1 0
0x1F	rs	rt	0	0x09	res	0b01

Format:

div8 rs, rt

Purpose:

To multiply two 32-bit registers representing four 8-bit values and store the result in two 32-bit registers representing four 16-bit results.

Description:

rs / rt
lo ← lower 32-bits, rem/quot pairs (result of lower two bytes)
hi ← upper 32-bits, rem/quot pairs (result of upper two bytes)

The 32-bit values in rs and rt are divided into four 8-bit sections aligned by their MSBs. The two operands are multiplied and the four results are stored in *hi/lo*. There are no flags to update for this operation.

Limitations:

None.

Examples:

Assembly	**Machine Code**
DIV8 R13, R8	011111_01101_01000_00000_01001_0000_01
DIV8 R2, R1	011111_00010_00001_00000_01001_0000_01

Operand Values	**Result**

R13 = 0x3B89AE61
R8 = 0x9165DFE0
	0x3B00EEFFF02901FD

R2 = 0xCF7979EA
R1 = 0x333BDE97
	0xCF00030213FDEA00

DIV16 DUAL16 Division

V-type

31 26	25 21	20 16	15 11	10 6	5 2	1 0
0x1F	rs	rt	0	0x09	res	0b10

Format:

div16 rs, rt

Purpose:

To multiply two 32-bit registers representing two 16-bit values and store the result in two 32-bit register representing two 32-bit results.

Description:

rs / rt
lo ← lower 32-bits, rem/quot pairs (result of lower halfword)
hi ← upper 32-bits, rem/quot pairs (result of upper halfword)

The 32-bit values in rs and rt are divided into two 16-bit sections aligned by their MSBs. The two operands are multiplied and the two results are stored in *hi/lo*. There are no flags to update for this operation.

Limitations:

None.

Examples:

Assembly	**Machine Code**
DIV16 R6, R5	011111_00110_00101_00000_01001_0000_10
DIV16 R15, R3	011111_01111_00011_00000_01001_0000_10

Operand Values	**Result**

R6 = 0x69DA137D
R5 = 0xF008A32C
	0x0A0AFFFA137D0000

R15 = 0x0879BCDD
R3 = 0xACE4563B
	0x08790000BCDD0000

MUL8 QUAD8 Multiply

V-type

31 26	25 21	20 16	15 11	10 6	5 2	1 0
0x1F	rs	rt	0	0x08	res	0b01

Format:

mul8 rs, rt

Purpose:

To multiply two 32-bit registers representing four 8-bit values and store the result in two 32-bit registers representing four 16-bit results.

Description:

rs * rt
lo ← lower 32-bits (result of lower two bytes)
hi ← upper 32-bits (result of upper two bytes)

The 32-bit values in rs and rt are divided into four 8-bit sections aligned by their MSBs. The two operands are multiplied and the four results are stored in *hi/lo*. There are no flags to update for this operation.

Limitations:

None.

Examples:

Assembly	**Machine Code**
MUL8 R4, R9	011111_00100_01001_00000_01000_0000_01
MUL8 R1, R6	011111_00001_00110_00000_01000_0000_01

Operand Values	**Result**

R4 = 0xF166D712
R9 = 0xDAD415E3
	0xCD3A547811A30FF6

R1 = 0x197B13BE
R9 = 0xDBDC4573
	0x156369B4051F555A

MUL16 DUAL16 Multiply

V-type

31 26	25 21	20 16	15 11	10 6	5 2	1 0
0x1F	rs	rt	0	0x08	res	0b10

Format:

mul16 rs, rt

Purpose:

To multiply two 32-bit registers representing two 16-bit values and store the result in two 32-bit registers representing two 32-bit results.

Description:

rs * rt
lo ← lower 32-bits (result of lower halfword)
hi ← upper 32-bits (result of upper halfword)

The 32-bit values in rs and rt are divided into two 16-bit sections aligned by their MSBs. The two operands are multiplied and the two results are stored in *hi/lo*. There are no flags to update for this operation.

Limitations:

None.

Examples:

Assembly	**Machine Code**
MUL16 R3, R10	011111_00011_01010_00000_01000_0000_10
MUL16 R9, R7	011111_01001_00111_00000_01000_0000_10

Operand Values	**Result**

R3 = 0xCA76A312
R10 = 0x4968EFF3
	0x3A0DE5F098D89816

R9 = 0x45654094
R7 = 0x42EDAD30
	0x122448812BB01FC0

NOT8

QUAD8 Logical Negation

V-type

31 26	25 21	20 16	15 11	10 6	5 2	1 0
0x1F	rs	0	rd	0x06	res	0b01

Format:

not8 rd, rs

Purpose:

To logically negate one 32-bit register, *rs*, representing four 8-bit values and store the result in a 32-bit register representing four 8-bit results.

Description:

rd ← ~rs

The 32-bit value in *rs* is divided into four 8-bit sections aligned by it's MSB. The operand is negated and the four results are stored in rd. There are no flags to update for this operation.

Limitations:

None.

Examples:

Assembly	**Machine Code**
`NOT8 R3, R6`	`011111_00110_00000_00011_00110_0000_01`
`NOT8 R10, R13`	`011111_01101_00000_01010_00110_0000_01`

Operand Values	**Result**
`R6 = 0x22003EA9`	`0xDDFFC156`
`R13 = 0x38DE90CA`	`0xC7216F35`

NOT16

DUAL16 Logical Negation

V-type

31 26	25 21	20 16	15 11	10 6	5 2	1 0
0x1F	rs	0	rd	0x06	res	0b10

Format:

not16 rd, rs

Purpose:

To logically negate one 32-bit register, *rs*, representing two 16-bit values and store the result in a 32-bit register representing two 16-bit results.

Description:

rd ← ~rs

The 32-bit value in *rs* is divided into two 16-bit sections aligned by it's MSB. The operand is negated and the two results are stored in rd. There are no flags to update for this operation.

Limitations:

None.

Examples:

Assembly	**Machine Code**
`NOT16 R5, R7`	`011111_00111_00000_00101_00110_0000_10`
`NOT16 R12, R3`	`011111_00011_00000_01100_00110_0000_10`

Operand Values	**Result**
`R7 = 0xBC189E3A`	`0x43E761C5`
`R3 = 0xA796DF21`	`0x586920DE`

OR8

QUAD8 Logical OR

V-type

31 26	25 21	20 16	15 11	10 6	5 2	1 0
0x1F	rs	rt	rd	0x05	res	0b01

Format:

or8 rd, rs, rt

Purpose:

To logically OR two 32-bit registers representing four 8-bit values and store the result in a 32-bit register representing four 8-bit results.

Description:

rd ← rs | rt

The 32-bit values in rs and rt are divided into four 8-bit sections aligned by their MSBs. The two operands are ORed and the four results are stored in rd. There are no flags to update for this operation.

Limitations:

None.

Examples:

Assembly	**Machine Code**
OR8 R4, R11, R15	011111_01011_01101_00100_00101_0000_01
OR8 R9, R12, R3	011111_01100_00011_01001_00101_0000_01

Operand Values	**Result**

R11 = 0x36D9AC21
R15 = 0xA5D9EC20
	0xB7D9EC21

R12 = 0xFF6B3A29
R3 = 0x8675309A
	0xFF7F3ABB

OR16

DUAL16 Logical OR

V-type

31 26	25 21	20 16	15 11	10 6	5 2	1 0
0x1F	rs	rt	rd	0x05	res	0b10

Format:

or16 rd, rs, rt

Purpose:

To logically OR two 32-bit registers representing two 16-bit values and store the result in a 32-bit register representing two 16-bit results.

Description:

rd ← rs | rt

The 32-bit values in rs and rt are divided into two 16-bit sections aligned by their MSBs. The two operands are ORed and the two results are stored in rd. There are no flags to update for this operation.

Limitations:

None.

Examples:

Assembly	**Machine Code**
OR16 R5, R8, R6	011111_01000_00110_00101_00101_0000_10
OR16 R1, R1, R10	011111_00001_01010_00001_00101_0000_10

Operand Values	**Result**

R8 = 0x28DB77AC
R6 = 0x13714DEA
	0x3BFB7FEE

R1 = 0x390417AD
R10 = 0xEFDA1D2A
	0xFFDE1FAF

SUB8

QUAD8 Subtraction

V-type

31 26	25 21	20 16	15 11	10 6	5 2	1 0
0x1F	rs	rt	rd	0x03	res	0b01

Format:

sub8 rd, rs, rt

Purpose:

To subtract two 32-bit registers representing four 8-bit values and store the result in a 32-bit register representing four 8-bit results.

Description:

rd ← rs - rt

The 32-bit values in rs and rt are divided into four 8-bit sections aligned by their MSBs. The two operands are subtracted and the four results are stored in rd. There are no flags to update for this operation.

Limitations:

None.

Examples:

Assembly	**Machine Code**
`SUB8 R8, R7, R8`	`011111_00111_01000_01000_00011_0000_01`
`SUB8 R3, R2, R14`	`011111_00010_01110_00011_00011_0000_01`

Operand Values	**Result**

R7 = 0xDEA27A23
R8 = 0x0D543A5B
	0xD14EE4C8

R2 = 0xE77FF352
R14 = 0x123AD32E
	0xD55C2024

SUB16 DUAL16 Subtraction

V-type

31 26	25 21	20 16	15 11	10 6	5 2	1 0
0x1F	rs	rt	rd	0x03	res	0b10

Format:

sub16 rd, rs, rt

Purpose:

To subtract two 32-bit registers representing two 16-bit values and store the result in a 32-bit register representing two 16-bit results.

Description:

rd ← rs - rt

The 32-bit values in rs and rt are divided into two 16-bit sections aligned by their MSBs. The two operands are subtracted and the two results are stored in rd. There are no flags to update for this operation.

Limitations:

None.

Examples:

Assembly	**Machine Code**
SUB16 R10, R9, R8	011111_01001_01000_01010_00011_0000_10
SUB16 R2, R8, R3	011111_01000_00011_00010_00011_0000_10

Operand Values	**Result**

R9 = 0xC0FC5BE8
R8 = 0x21DC79DE
	0x9F20E20A

R8 = 03AFF879ED
R3 = 0xD465ADCF
	0x669ACC1E

XOR8 QUAD8 Logical XOR

V-type

31 26	25 21	20 16	15 11	10 6	5 2	1 0
0x1F	rs	rt	rd	0x07	res	0b01

Format:

xor8 rd, rs, rt

Purpose:

To logically XOR two 32-bit registers representing four 8-bit values and store the result in a 32-bit register representing four 8-bit results.

Description:

rd ← rs ^ rt

The 32-bit values in rs and rt are divided into four 8-bit sections aligned by their MSBs. The two operands are XORed and the four results are stored in rd. There are no flags to update for this operation.

Limitations:

None.

Examples:

Assembly	**Machine Code**
`XOR8 R2, R9, R6`	`011111_01001_00110_00010_00111_0000_01`
`XOR8 R1, R12, R3`	`011111_01100_00011_00001_00111_0000_01`

Operand Values	**Result**

R9 = 0xBAB4637A
R6 = 0xEEFF543C
	0x544B3746

R12 = 0x543200FB
R3 = 0xBA79E796
	0xEE4BE76D

XOR16 DUAL16 Logical XOR

V-type

31–26	25–21	20–16	15–11	10–6	5–2	1–0
0x1F	rs	rt	rd	0x07	res	0b10

Format:

xor16 rd, rs, rt

Purpose:

To logically XOR two 32-bit registers representing two 16-bit values and store the result in a 32-bit register representing two 16-bit results.

Description:

rd ← rs ^ rt

The 32-bit values in rs and rt are divided into two 16-bit sections aligned by their MSBs. The two operands are XORed and the two results are stored in rd. There are no flags to update for this operation.

Limitations:

None.

Examples:

Assembly	**Machine Code**
XOR16 R4, R12, R7	011111_01100_00111_00100_00111_0000_10
XOR16 R5, R9, R8	011111_01001_01000_00101_00111_0000_10

Operand Values	**Result**

R12 = 0x23BDEAFE
R7 = 0x10F897ED
	0x33457D13

R9 = 0x96321EAD
R8 = 0x46EA546B
	0xD0D84AC6

VMFHI Vector Move from HI

V-type

31 26	25 21	20 16	15 11	10 6	5 2	1 0
0x1F	0	0	rd	0x0A	res	0b01/ 0b10

Format:

vmfhi rd

Purpose:

To move the upper 32-bits from a QUAD8/DUAL16 MUL/DIV operation into a destination register, rd.

Description:

rd ← *hi*

The 32-bit value stored in *hi* after a multiply or divide is moved into a GPR specified by rd. The *hi* register mentioned here is separate from the 32-bit operations *hi* register.

Limitations:

None.

Examples:

Assembly	Machine Code
VMFHI R11	011111_00000_00000_01011_01010_0000_01
VMFHI R4	011111_00000_00000_00100_01010_0000_01

Operand Values	Result
hi = 0xAC349DD01	0xAC349DD01
hi = 0xB790035D	0xB790035D

VMFLO

Vector Move from LO

V-type

31 26	25 21	20 16	15 11	10 6	5 2	1 0
0x1F	0	0	rd	0x0A	res	0b01/ 0b10

Format:

vmflo rd

Purpose:

To move the lower 32-bits from a QUAD8/DUAL16 MUL/DIV operation into a destination register, rd.

Description:

rd ← *hi*

The 32-bit value stored in *lo* after a multiply or divide is moved into a GPR specified by rd. The *lo* register mentioned here is separate from the 32-bit operations *lo* register.

Limitations:

None.

Examples:

Assembly	**Machine Code**
VMFLO R10	011111_00000_00000_01010_01011_0000_01
VMFLO R3	011111_00000_00000_00011_01011_0000_01

Operand Values	**Result**
lo = 0x1369D47E2	0x1369D47E2
lo = 0x7BE2AC88	0x7BE2AC88

III. Verilog - Implementation

A. Verilog Implementation

MIPS_TB_1

```
`timescale 100ps / 10ps
// Test bench that asserts reset and establishes a clock for the CPU
// Alec Selfridge/Joshua Hightower

module MIPS_TB_1;

    // Inputs
    reg clk;
    reg reset;


    Highridge_CPU  HR(clk, reset);

  // 1ns system clock (1GHz)
  always
    #5 clk = ~clk;

    initial begin
        $timeformat(-9, 2, " ns", 10);

        //initially load Data Memory with values from data file
        $readmemh("dMem16_Fa15.dat", HR.CPU.dMem.M);
        //initially load Instruction Memory with values from data file
        $readmemh("iMem16_Fa15.dat", HR.CPU.IU.iMem.M);

        // Initialize Inputs
        clk   = 0;
        reset = 0;

        // Wait 100 ns for global reset to finish
        #100;

        //reset all modules in the testbench
        @(negedge clk)
          reset = 1;
        #10;
        @(negedge clk)
          reset = 0;

    end

endmodule
```

Highridge_CPU

```verilog
`timescale 100ps / 10ps
/******************************************************************************************************
*
* Author: Joshua Hightower / Alec Selfridge
* Email: joshuahightower2@gmail.com / aeselfridge@gmail.com
* Filename: MCU.v
* Date: November 14th, 2015
* Version: 1.2
*
* Notes:
*
*******************************************************************************************************
*/
module Highridge_CPU(clk, reset);

    input clk, reset;

    //instantiate wirers to connect MCU with the CPU
    wire       HILO_ld, D_En, im_cs, im_rd, im_wr, dm_cs, dm_rd, dm_wr, T_Sel, C, V, N, Z, DBZ,
               pc_ld, pc_inc, ir_ld, io_cs, io_rd, io_wr, int_ack, intr, S_Sel;
    wire [1:0]  pc_sel, DA_Sel, DO_Sel, SH_Sel, mSel, Ysel;
    wire [3:0]  Y_Sel;
    wire [4:0]  FS;
    wire [31:0] IR, flags, ALU_OUT;


    CPU_MIPS      CPU(.clk(clk),       .reset(reset),     .C(C),             .N(N),
                    .Z(Z),             .V(V),             .DBZ(DBZ),         .IR(IR),
                    .pc_sel(pc_sel),   .pc_ld(pc_ld),     .pc_inc(pc_inc),   .ir_ld(ir_ld),
                    .im_cs(im_cs),     .im_rd(im_rd),     .im_wr(im_wr),     .D_En(D_En),
                    .T_Sel(T_Sel),     .HILO_ld(HILO_ld), .dm_cs(dm_cs),     .dm_rd(dm_rd),
                    .dm_wr(dm_wr),     .io_cs(io_cs),     .io_rd(io_rd),     .io_wr(io_wr),
                    .DA_Sel(DA_Sel),   .Y_Sel(Y_Sel),     .FS(FS),           .flags(flags),
                    .S_Sel(S_Sel),     .DO_Sel(DO_Sel),   .int_ack(int_ack), .intr(intr),
                    .ALU_OUT(ALU_OUT), .Ysel(Ysel),       .mSel(mSel),       .SH_Sel(SH_Sel),
                    .HILO_ld_SIMD(HILO_ld_SIMD));

    MCU           CU(.clk(clk),        .reset(reset),     .IR(IR),           .FS(FS),
                    .pc_sel(pc_sel),   .pc_ld(pc_ld),     .pc_inc(pc_inc),  .ir_ld(ir_ld),
                    .im_cs(im_cs),     .im_rd(im_rd),    .im_wr(im_wr),     .dm_cs(dm_cs),
                    .dm_rd(dm_rd),     .dm_wr(dm_wr),    .D_En(D_En),       .HILO_ld(HILO_ld),
                    .T_sel(T_Sel),     .DA_sel(DA_Sel),  .Y_sel(Y_Sel),     .intr(intr),
                    .int_ack(int_ack), .C(C),            .N(N),             .Z(Z),
                    .V(V),             .io_cs(io_cs),    .io_rd(io_rd),     .io_wr(io_wr),
                    .flags(flags),     .S_Sel(S_Sel),    .DO_Sel(DO_Sel),   .sp_flags(ALU_OUT),
                    .DBZ(DBZ),         .SH_Sel(SH_Sel),  .mSel(mSel),       .Ysel(Ysel),
                    .HILO_ld_SIMD(HILO_ld_SIMD));

endmodule
```

MCU

```
`timescale 100ps / 10ps
/*******************************************************************************************
*
* Author: Joshua Hightower / Alec Selfridge
* Email: joshuahightower2@gmail.com / aeselfridge@gmail.com
* Filename: MCU.v
* Date: October 7th, 2015
* Version: 1.0
*
* Notes: The MCU(MIPS Control Unit) verilog module is a finite moore state machine that controls
* the control signals for the MIPS ISA.  The FSM controls these signals by evaluating the 32 bit
* iput IR and changes states based off that input.  Each clock tick will change to a new state and
* control what the datapath, data memory, and instruction unit are doing.
*
*******************************************************************************************/
module MCU(clk, reset, intr, C, N, Z, V, DBZ, IR, int_ack, pc_sel, pc_ld, pc_inc, ir_ld,
           im_cs, im_rd, im_wr, D_En, T_sel, HILO_ld, HILO_ld_SIMD, dm_cs, dm_rd, dm_wr, io_cs,
           io_rd, io_wr, DA_sel, Y_sel, FS, flags, S_Sel, DO_Sel, sp_flags, SH_Sel, mSel, Ysel);

   input            clk, reset, intr, C, N, Z, V, DBZ;
   input     [31:0]  IR;
   input     [31:0]  sp_flags;
   output reg         int_ack, pc_ld, pc_inc, ir_ld;
   output reg            im_cs, im_rd, im_wr;
   output reg            D_En, T_sel, HILO_ld, HILO_ld_SIMD,S_Sel;
   output reg            dm_cs, dm_rd, dm_wr, io_cs, io_rd, io_wr;
   output reg  [1:0]  pc_sel, DA_sel, DO_Sel, SH_Sel, mSel, Ysel;
   output reg  [3:0]  Y_sel;
   output reg  [4:0]  FS;
   output wire [31:0] flags;

   parameter
      RESET      = 00,  FETCH      = 01,  DECODE       = 02,  SLL       = 03,  SRL    = 04,
      SRA       = 05,  JR1        = 06,  MULT         = 07,  DIV       = 08,  RETI_1 = 09,
      ADD       = 10,  ADDU       = 11,  AND          = 12,  OR        = 13,  NOR    = 14,
      XOR       = 15,  SUB        = 16,  SUBU         = 17,  SLT       = 18,  SLTU   = 19,
      ORI       = 20,  LUI        = 21,  LW           = 22,  SW        = 23,  SETIE  = 24,
      BEQ1      = 25,  BNE1       = 26,  BLEZ1        = 27,  BGTZ1     = 28,  ADDI   = 29,
      WB_alu     = 30,  WB_imm     = 31,  WB_din       = 32,  WB_hi     = 33,  WB_lo   = 34,
      WB_mem     = 35,  SLTI       = 36,  SLTUI        = 37,  ANDI      = 38,  XORI   = 39,
      OUTPUT     = 40,  JAL        = 41,  INPUT        = 42,  JUMP      = 43,  RB      = 44,
      BEQ2      = 45,  BNE2       = 46,  BLTZ1        = 47,  MEM_ACC   = 48,  JR2    = 49,
      BLEZ2      = 50,  BGTZ2      = 51,  WB_io_in     = 52,  WB_io_mem  = 53,  IO_ACC = 54,
      BLTZ2      = 55,  BGEZ1      = 56,  BGEZ2        = 57,  JALR1     = 58,  JALR2   = 59,
      BDT       = 60,  BDF        = 61,  WB_sh        = 62,  BSI       = 63,  BCI     = 64,
      RR         = 65,  E_KEY      = 66,  STND         = 67,  SIMD8     = 68,  SIMD16  = 69,
      ADD8      = 70,  SUB8       = 71,  AND8         = 72,  OR8       = 73,  NOT8    = 74,
      XOR8      = 75,  MUL8       = 76,  DIV8         = 77,  ADD16     = 78,  SUB16   = 79,
      AND16      = 80,  OR16       = 81,  NOT16        = 82,  XOR16     = 83,  MUL16   = 84,
      DIV16      = 85,  WB_SIMD_alu = 86,  WB_SIMD_upper = 87,  VMFLO     = 88,  VMFHI   = 89,
      RETI_3     = 490, RETI_4      = 491, RETI_5       = 492, RETI_6    = 493, RETI_7 = 494,
      RETI_8     = 495, RBO         = 496,
      INTR_1     = 500, INTR_2       = 501, INTR_3       = 502, INTR_4    = 503, INTR_5 = 504,
      INTR_6     = 505, INTR_7 = 506, INTR_8 = 507, INTR_9 = 508, RETI_2 = 509,
      BREAK       = 510,
      ILLEGAL_OP  = 511;

   //state register up to 512 states
   reg [8:0] state;
   reg      psi, psc, psv, psn, psz, psd, nsi, nsc, nsv, nsn, nsz, nsd;
   reg      fl_ld;
```

```
/*
    CONTROL WORD PER STATE
    {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'b00_0_0_0;
   {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
    {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_00_0_0_0000;  FS         = 5'h0;
    {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;       int_ack     = int_ack;
    {io_cs, io_rd, io_wr}               = 3'b0_0_0;           fl_ld       = 1'b0;
    {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
    #1 {nsi, nsc, nsv, nsn, nsz}         = {psi, psc, psv, psn, psz};
    #1 {nsd, nsi, nsc, nsv, nsn, nsz}    = {DBZ, psi, C, V, N, Z};
    state = ;
*/

//*********************************************************************************
// Register File Dump Task
// - The reg dumop task displays registers 0 - 15 to be displayed on the console.
// - Used in Illegal Op state and Break state to show what values got into the regFile.
//*********************************************************************************
task regDump;
 integer      i;
reg    [3:0] j;
    begin
  {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'b00_0_0_0;
  {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
  {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_00_0_0_0000; FS          = 5'h0;
  {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;       int_ack     = 1'b0;
    {io_cs, io_rd, io_wr}               = 3'b0_0_0;           fl_ld       = 1'b0;
    {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
  @(negedge clk)
    for(i = 0; i <= 15; i = i + 1) begin
      j = i;
      #1 $display("time = %t   ||  Register 0x%h = 0x%h ", $time, j,
                    MIPS_TB_1.HR.CPU.ID.registerFile.register[i]);
    end
    end
endtask

//******************************************************************************
// PC/IR Registers Dump Task
// - The pc/ir dump task displays the PC register value and the IR register value.
// - Used in Illegal Op state to see where the illegal op was.
//******************************************************************************
task pc_ir_Dump;
#1 $display("PC = %h IR = %h", MIPS_TB_1.HR.CPU.IU.PC_reg.PC_out,
               MIPS_TB_1.HR.CPU.IU.IR_reg.IR_out);
endtask


//*************************************************************************************
 // Memory Dump Task
 // - The memory at a specified location is displyaed to verify if the SW instruction works.
 //*************************************************************************************
task memDump;
    reg [11:0] r;
    begin
        for(r = 12'h0C0; r < 13'h100; r = r + 4)
        begin
            #1 $display("time = %t   ||  dM[%h] = 0x%h", $time, r,
            {MIPS_TB_1.HR.CPU.dMem.M[r],
             MIPS_TB_1.HR.CPU.dMem.M[r+1],
             MIPS_TB_1.HR.CPU.dMem.M[r+2],
             MIPS_TB_1.HR.CPU.dMem.M[r+3]});
        end
   end
endtask
```

```
//******************************************************************************
// dmemDump Task
// - This task displays the byte location in data memory 0x3f7.
// - The memory location holds the flags saved when an intr occurs.
//******************************************************************************
task dmemDump;
    reg [11:0] r;
    begin
        {dm_cs, dm_rd, dm_wr} = 3'b1_1_0;
        r = 12'h3f7;
         $display("Flags in data memory:");
         $display("time = %t   ||  dM[%h] = 0x%b", $time, r,
        {MIPS_TB_1.HR.CPU.dMem.M[r]});
   end
endtask

//assign statement to map the present state flags to a 32 bit register
assign flags = {26'b0, psd, psi, psc, psv, psn, psz};

//positive clock and reset logic to give the present state flags their input
// if reset, turn all 5 flags to 0
//  if flag load signal is high, give the sp_flags lower 5 bits to respective flags
//   else, the present state gets the next state flags
always@(posedge clk or posedge reset)
    if(reset)
        {psd, psi, psc, psv, psn, psz} <= 6'b0;
    else if(fl_ld)
        {psd, psi, psc, psv, psn, psz} <= sp_flags[5:0];
    else
        {psd, psi, psc, psv, psn, psz} <= {nsd, nsi, nsc, nsv, nsn, nsz};

//STATE MACHINE LOGIC
always@(posedge clk or posedge reset)
begin
    //if reset is high, ALU_Out <- SP_INIT
    if(reset)
    begin
        {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'b00_0_0_0;
        {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
        {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_00_0_0_0000; FS          = 5'h15;
        {dm_cs, dm_rd, dm_wr}              = 3'b0_0_0;        int_ack     = 1'b0;
        {io_cs, io_rd, io_wr}              = 3'b0_0_0;            fl_ld       = 1'b0;
        {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
        {nsd, nsi, nsc, nsv, nsn, nsz}       = 6'b0;
        state = RESET;
    end

    else
        case(state)
            //if(int_ack==1 & intr == 0)
            //    int_ack = 0
            //IR <- iM(PC), PC <- PC + 4
            FETCH:
                //if interrupt, go to INTR_1, else go to DECODE
                if(int_ack == 0 & (intr == 1 & psi == 1))
                begin
                  @(negedge clk)
                    {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
                    {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
                    {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_x_0_xxxx; FS     = 5'h0;
                    {dm_cs, dm_rd, dm_wr}              = 3'b0_0_0;        int_ack = int_ack;
                    {io_cs, io_rd, io_wr}              = 3'b0_0_0;            fl_ld   = 1'b0;
                    {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
                    {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
                    state = INTR_1;
                end
                else if((int_ack == 1 & intr == 1) | intr == 0)
                begin
                  @(negedge clk)
                    {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_1_1;
                    {im_cs, im_rd, im_wr}                 = 3'b1_1_0;
                    {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_x_0_xxxx; FS     = 5'h0;
                    {dm_cs, dm_rd, dm_wr}              = 3'b0_0_0;
                    {io_cs, io_rd, io_wr}              = 3'b0_0_0;           fl_ld   = 1'b0;
                    {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
                    {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
                    //if int_ack and !intr, turn off int_ack, else leave it be
                    if(int_ack == 1 & intr == 0)
                       int_ack = 1'b0;
                    else
                       int_ack = int_ack;
                    state = DECODE;
                end
            //PC <- 32'b0, $sp <- ALU_Out(0x3FC), int_ack <- 0
            RESET:
                begin
                  @(negedge clk)
                    {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'b00_0_0_0;
                    {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
                    {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b1_11_0_0_0000; FS     = 5'h0;
                    {dm_cs, dm_rd, dm_wr}              = 3'b0_0_0;       int_ack = 1'b0;
                    {io_cs, io_rd, io_wr}              = 3'b0_0_0;           fl_ld   = 1'b0;
                    {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
                    {nsd, nsi, nsc, nsv, nsn, nsz}       = 6'b0;
                    state = FETCH;
                end
            //if(IR[31:26==0)
            //   RS <- $rs, RT <- $rt
            //else
            //   RS <- $rs, RT <- DT
            DECODE:
                begin
                    @(negedge clk)
                     if(IR[31:26] == 6'h0) //R type
                       begin
                           {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
                           {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
                           {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_0_0_0000; FS     = 5'h0;
                           {dm_cs, dm_rd, dm_wr}              = 3'b0_0_0;       int_ack = int_ack;
                           {io_cs, io_rd, io_wr}              = 3'b0_0_0;           fl_ld   = 1'b0;
                           {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
                           {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
                           case(IR[5:0])
                               6'h00  : state = SLL;
                               6'h02  : state = SRL;
                               6'h03  : state = SRA;
                               6'h08  : state = JR1;
                               6'h09  : state = JALR1;
                               6'h10  : state = WB_hi; //MFHI
                               6'h12  : state = WB_lo; //MFLO
                               6'h18  : state = MULT;
                               6'h1A  : state = DIV;
                               6'h20  : state = ADD;
                               6'h21  : state = ADDU;
                               6'h22  : state = SUB;
                               6'h23  : state = SUBU;
                               6'h24  : state = AND;
                               6'h25  : state = OR;
                               6'h26  : state = XOR;
                               6'h27  : state = NOR;
                               6'h2A  : state = SLT;
                               6'h2B  : state = SLTU;
                               6'h0D  : state = BREAK;
                               6'h1F  : state = SETIE;
```

```
                6'h2C  : state = RB;
                6'h2D  : state = RBO;
                6'h34  : state = RR;
                default: state = ILLEGAL_OP;
            endcase
        end
     else       //if we are using a BNE/BEQ instruction, RS <- RS & RT <- RT
      if((IR[31:26]==6'h04) | (IR[31:26]==6'h05))
        begin
            {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
            {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
            {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_0_0_xxxx; FS     = 5'h0;
            {dm_cs, dm_rd, dm_wr}              = 3'b0_0_0;       int_ack = int_ack;
            {io_cs, io_rd, io_wr}              = 3'b0_0_0;           fl_ld = 1'b0;
            {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
            {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
            case(IR[31:26])
                6'h01  : state = BGEZ1;
                6'h02  : state = JUMP;
                6'h03  : state = JAL;
                6'h04  : state = BEQ1;
                6'h05  : state = BNE1;
                6'h06  : state = BLEZ1;
                6'h07  : state = BGTZ1;
                6'h08  : state = ADDI;
                6'h09  : state = BLTZ1;
                6'h0A  : state = SLTI;
                6'h0B  : state = SLTUI;
                6'h0C  : state = ANDI;
                6'h0D  : state = ORI;
                6'h0E  : state = XORI;
                6'h0F  : state = LUI;
                6'h10  : state = BDT;
                6'h11  : state = BDF;
                6'h1C  : state = INPUT;
                6'h1D  : state = OUTPUT;
                6'h1E  : state = RETI_1;
                6'h23  : state = LW;
                6'h2B  : state = SW;
                6'h24  : state = BSI;
                6'h25  : state = BCI;
                6'h1F  : state = E_KEY;
                default: state = ILLEGAL_OP;
            endcase
        end
      else   //if RETI, RS <- $sp
       if(IR[31:26]==6'h1E)
         begin
            {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
            {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
            {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_x_0_xxxx; FS     = 5'h0;
            {dm_cs, dm_rd, dm_wr}              = 3'b0_0_0;       int_ack = int_ack;
            {io_cs, io_rd, io_wr}              = 3'b0_0_0;           fl_ld   = 1'b0;
            {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b1_00_00_00_00; HILO_ld_SIMD = 1'b0;
            {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
            case(IR[31:26])
                6'h01  : state = BGEZ1;
                6'h02  : state = JUMP;
                6'h03  : state = JAL;
                6'h04  : state = BEQ1;
                6'h05  : state = BNE1;
                6'h06  : state = BLEZ1;
                6'h07  : state = BGTZ1;
                6'h08  : state = ADDI;
                6'h09  : state = BLTZ1;
                6'h0A  : state = SLTI;
                6'h0B  : state = SLTUI;
```

```
                  6'h0C  : state = ANDI;
                  6'h0D  : state = ORI;
                  6'h0E  : state = XORI;
                  6'h0F  : state = LUI;
                  6'h10  : state = BDT;
                  6'h11  : state = BDF;
                  6'h1C  : state = INPUT;
                  6'h1D  : state = OUTPUT;
                  6'h1E  : state = RETI_1;
                  6'h23  : state = LW;
                  6'h2B  : state = SW;
                  6'h24  : state = BSI;
                  6'h25  : state = BCI;
                  6'h1F  : state = E_KEY;
                  default: state = ILLEGAL_OP;
              endcase
           end
        else  //else, RS <- RS & RT <- S.E.IR[15:0]
         begin
              {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
              {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
              {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_1_0_xxxx; FS     = 5'h0;
              {dm_cs, dm_rd, dm_wr}              = 3'b0_0_0;       int_ack = int_ack;
              {io_cs, io_rd, io_wr}              = 3'b0_0_0;           fl_ld = 1'b0;
              {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
              {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
              case(IR[31:26])
                  6'h01  : state = BGEZ1;
                  6'h02  : state = JUMP;
                  6'h03  : state = JAL;
                  6'h04  : state = BEQ1;
                  6'h05  : state = BNE1;
                  6'h06  : state = BLEZ1;
                  6'h07  : state = BGTZ1;
                  6'h08  : state = ADDI;
                  6'h09  : state = BLTZ1;
                  6'h0A  : state = SLTI;
                  6'h0B  : state = SLTUI;
                  6'h0C  : state = ANDI;
                  6'h0D  : state = ORI;
                  6'h0E  : state = XORI;
                  6'h0F  : state = LUI;
                  6'h10  : state = BDT;
                  6'h11  : state = BDF;
                  6'h1C  : state = INPUT;  //load
                  6'h1D  : state = OUTPUT; //store
                  6'h1E  : state = RETI_1;
                  6'h23  : state = LW;
                  6'h2B  : state = SW;
                  6'h24  : state = BSI;
                  6'h25  : state = BCI;
                  6'h1F  : state = E_KEY;
                  default: state = ILLEGAL_OP;
              endcase
         end
        end
    //ALU_Out <- RT($rt) << shamt
    SLL:
       begin
           @(negedge clk)
            {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
            {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
            {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_x_0_xxxx; FS     = 5'h0C;
            {dm_cs, dm_rd, dm_wr}              = 3'b0_0_0;       int_ack = int_ack;
            {io_cs, io_rd, io_wr}              = 3'b0_0_0;         fl_ld   = 1'b0;
            {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00;    HILO_ld_SIMD = 1'b0;
            #1 {nsd, nsi, nsc, nsv, nsn, nsz}    = {DBZ, psi, C, V, N, Z};
          state = WB_sh;
       end
    //ALU_Out <- RT($rt) >> shamt
    SRL:
       begin
          @(negedge clk)
            {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
            {im_cs, im_rd, im_wr}                  = 3'b0_0_0;
            {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_x_0_xxxx; FS     = 5'h0D;
            {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;       int_ack = int_ack;
            {io_cs, io_rd, io_wr}               = 3'b0_0_0;         fl_ld   = 1'b0;
            {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_01_00_00;   HILO_ld_SIMD = 1'b0;
            #1 {nsd, nsi, nsc, nsv, nsn, nsz}    = {DBZ, psi, C, V, N, Z};
          state = WB_sh;
       end
    //ALU_Out <- RT($rt) >>> shamt
    SRA:
       begin
          @(negedge clk)
            {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
            {im_cs, im_rd, im_wr}                  = 3'b0_0_0;
            {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_x_0_xxxx; FS     = 5'h0E;
            {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;       int_ack = int_ack;
            {io_cs, io_rd, io_wr}               = 3'b0_0_0;         fl_ld   = 1'b0;
            {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_10_00_00;   HILO_ld_SIMD = 1'b0;
            #1 {nsd, nsi, nsc, nsv, nsn, nsz}    = {DBZ, psi, C, V, N, Z};
          state = WB_sh;
       end
    //{HI, LO} <- RS($rs) * RT($rt)
    MULT:
       begin
          @(negedge clk)
            {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
            {im_cs, im_rd, im_wr}                  = 3'b0_0_0;
            {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_x_1_xxxx; FS     = 5'h1E;
            {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;       int_ack = int_ack;
            {io_cs, io_rd, io_wr}               = 3'b0_0_0;         fl_ld   = 1'b0;
            {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00;   HILO_ld_SIMD = 1'b0;
            #1 {nsd, nsi, nsc, nsv, nsn, nsz}    = {DBZ, psi, C, V, N, Z};
          state = FETCH;
       end
    //{HI, LO} <- RS($rs) / RT($rt)
    DIV:
       begin
          @(negedge clk)
            {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
            {im_cs, im_rd, im_wr}                  = 3'b0_0_0;
            {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_x_1_xxxx; FS     = 5'h1F;
            {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;       int_ack = int_ack;
            {io_cs, io_rd, io_wr}               = 3'b0_0_0;         fl_ld   = 1'b0;
            {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00;   HILO_ld_SIMD = 1'b0;
            #1 {nsd, nsi, nsc, nsv, nsn, nsz}    = {DBZ, psi, C, V, N, Z};
          state = FETCH;
       end
    //ALU_Out <- RS($rs) + RT($rt) signed
    ADD:
       begin
          @(negedge clk)
            {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
            {im_cs, im_rd, im_wr}                  = 3'b0_0_0;
            {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_x_0_xxxx; FS     = 5'h02;
            {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;       int_ack = int_ack;
            {io_cs, io_rd, io_wr}               = 3'b0_0_0;         fl_ld   = 1'b0;
            {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00;   HILO_ld_SIMD = 1'b0;
            #1 {nsd, nsi, nsc, nsv, nsn, nsz}    = {DBZ, psi, C, V, N, Z};
          state = WB_alu;
       end
```

```
         //ALU_Out <- RS($rs) + RT($rt) unsigned
         ADDU:
            begin
               @(negedge clk)
                  {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
                  {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
                  {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_x_0_xxxx; FS     = 5'h03;
                  {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;        int_ack = int_ack;
                  {io_cs, io_rd, io_wr}               = 3'b0_0_0;           fl_ld   = 1'b0;
                  {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
                  #1 {nsd, nsi, nsc, nsv, nsn, nsz}    = {DBZ, psi, C, V, N, Z};
                  state = WB_alu;
            end
         //ALU_Out <- RS($rs) - RT($rt) signed
         SUB:
            begin
               @(negedge clk)
                 {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
                 {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
                 {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_x_0_xxxx; FS     = 5'h04;
                 {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;        int_ack = int_ack;
                 {io_cs, io_rd, io_wr}               = 3'b0_0_0;          fl_ld   = 1'b0;
                 {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00;   HILO_ld_SIMD = 1'b0;
                 #1 {nsd, nsi, nsc, nsv, nsn, nsz}    = {DBZ, psi, C, V, N, Z};
               state = WB_alu;
            end
         //ALU_Out <- RS($rs) - RT($rt) unsigned
         SUBU:
            begin
               @(negedge clk)
                  {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
                  {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
                  {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_x_0_xxxx; FS     = 5'h05;
                  {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;        int_ack = int_ack;
                  {io_cs, io_rd, io_wr}               = 3'b0_0_0;           fl_ld   = 1'b0;
                  {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
                  #1 {nsd, nsi, nsc, nsv, nsn, nsz}    = {DBZ, psi, C, V, N, Z};
                  state = WB_alu;
            end
         //ALU_Out <- RS($rs) & RT($rt)
         AND:
            begin
               @(negedge clk)
                  {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
                  {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
                  {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_x_0_xxxx; FS     = 5'h08;
                  {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;        int_ack = int_ack;
                  {io_cs, io_rd, io_wr}               = 3'b0_0_0;           fl_ld   = 1'b0;
                  {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
                  #1 {nsd, nsi, nsc, nsv, nsn, nsz}    = {DBZ, psi, C, V, N, Z};
                  state = WB_alu;
            end
         //ALU_Out <- RS($rs) | RT($rt)
         OR:
            begin
               @(negedge clk)
                  {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
                  {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
                  {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_x_0_xxxx; FS     = 5'h09;
                  {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;        int_ack = int_ack;
                  {io_cs, io_rd, io_wr}               = 3'b0_0_0;           fl_ld   = 1'b0;
                  {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
                  #1 {nsd, nsi, nsc, nsv, nsn, nsz}    = {DBZ, psi, C, V, N, Z};
                  state = WB_alu;
            end
         //ALU_Out <- RS($rs) ^ RT($rt)
         XOR:
```

```
         begin
            @(negedge clk)
               {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
               {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
               {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_x_0_xxxx; FS      = 5'h0A;
               {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;        int_ack = int_ack;
               {io_cs, io_rd, io_wr}               = 3'b0_0_0;            fl_ld   = 1'b0;
               {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
               #1 {nsd, nsi, nsc, nsv, nsn, nsz}    = {DBZ, psi, C, V, N, Z};
               state = WB_alu;
         end
      //ALU_Out <- ~(RS($rs) | RT($rt))
      NOR:
         begin
            @(negedge clk)
               {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
               {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
               {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_x_0_xxxx; FS      = 5'h0B;
               {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;        int_ack = int_ack;
               {io_cs, io_rd, io_wr}               = 3'b0_0_0;            fl_ld   = 1'b0;
               {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
               #1 {nsd, nsi, nsc, nsv, nsn, nsz}    = {DBZ, psi, C, V, N, Z};
               state = WB_alu;
         end
      //if(RS($rs) < RT($rt)), ALU_Out <- 1, else ALU_Out <- 0  signed
      SLT:
         begin
            @(negedge clk)
               {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
               {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
               {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_x_0_xxxx; FS      = 5'h06;
               {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;        int_ack = int_ack;
               {io_cs, io_rd, io_wr}               = 3'b0_0_0;            fl_ld   = 1'b0;
               {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
               #1 {nsd, nsi, nsc, nsv, nsn, nsz}    = {DBZ, psi, C, V, N, Z};
               state = WB_alu;
         end
      //if(RS($rs) < RT($rt)), ALU_Out <- 1, else ALU_Out <- 0  unsigned
      SLTU:
         begin
            @(negedge clk)
               {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
               {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
               {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_x_0_xxxx; FS      = 5'h07;
               {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;        int_ack = int_ack;
               {io_cs, io_rd, io_wr}               = 3'b0_0_0;            fl_ld   = 1'b0;
               {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
               #1 {nsd, nsi, nsc, nsv, nsn, nsz}    = {DBZ, psi, C, V, N, Z};
               state = WB_alu;
         end
      //nsi <- ~psi
      SETIE:
         begin
            @(negedge clk)
               {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
               {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
               {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_x_0_xxxx; FS      = 5'h0;
               {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;        int_ack = int_ack;
               {io_cs, io_rd, io_wr}               = 3'b0_0_0;            fl_ld   = 1'b0;
               {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
               #1 {nsd, nsi, nsc, nsv, nsn, nsz}    = {psd, ~psi, psc, psv, psn, psz};
               state = FETCH;
         end
      //PC <- jump address
      JUMP:
         begin
            @(negedge clk)
               {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'b01_1_0_0;
               {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
               {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_x_0_xxxx; FS     = 5'h0;
               {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;        int_ack = int_ack;
               {io_cs, io_rd, io_wr}               = 3'b0_0_0;           fl_ld   = 1'b0;
               {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
               {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
               state = FETCH;
         end
      //$r31 <- PC + 4; PC <- jump address
      JAL:
         begin
            @(negedge clk)
               {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'b01_1_0_0;
               {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
               {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b1_10_x_0_0100; FS     = 5'h0;
               {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;        int_ack = int_ack;
               {io_cs, io_rd, io_wr}               = 3'b0_0_0;           fl_ld   = 1'b0;
               {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
               {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
               state = FETCH;
         end
      //$r31 <- PC + 4; ALU_Out <- RS
      JALR1:
         begin
            @(negedge clk)
               {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
               {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
               {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b1_10_x_0_0100; FS     = 5'h0;
               {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;        int_ack = int_ack;
               {io_cs, io_rd, io_wr}               = 3'b0_0_0;           fl_ld   = 1'b0;
               {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
               {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
               state = FETCH;
         end
      //PC <- ALU_Out
      JALR2:
         begin
            @(negedge clk)
               {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'b10_1_0_0;
               {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
               {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_x_0_0100; FS     = 5'h0;
               {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;        int_ack = int_ack;
               {io_cs, io_rd, io_wr}               = 3'b0_0_0;           fl_ld   = 1'b0;
               {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
               {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
               state = FETCH;
         end
      //ALU_Out <- R[$rs]
      JR1:
         begin
            @(negedge clk)
               {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
               {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
               {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_x_0_xxxx; FS     = 5'h0;
               {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;        int_ack = int_ack;
               {io_cs, io_rd, io_wr}               = 3'b0_0_0;           fl_ld   = 1'b0;
               {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
               {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
               state = JR2;
         end
      //PC <- ALU_Out
      JR2:
         begin
            @(negedge clk)
               {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'b10_1_0_0;
               {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
               {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_x_0_0000; FS      = 5'h0;
               {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;        int_ack = int_ack;
               {io_cs, io_rd, io_wr}               = 3'b0_0_0;            fl_ld   = 1'b0;
               {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
               {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
               state = FETCH;
           end
       //ALU_Out <- RS($rs) - RT($rt)
       BEQ1:
           begin
               @(negedge clk)
                {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
                {im_cs, im_rd, im_wr}                  = 3'b0_0_0;
                {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_x_0_xxxx; FS      = 5'h04;
                {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;        int_ack = int_ack;
                {io_cs, io_rd, io_wr}               = 3'b0_0_0;          fl_ld   = 1'b0;
                {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
                #1 {nsd, nsi, nsc, nsv, nsn, nsz}    = {DBZ, psi, C, V, N, Z};
               state = BEQ2;
           end
       //if(psz==1) PC <- branch address, else PC <- PC
       BEQ2:
           begin
            @(negedge clk)
               if(psz == 1'b1)
                  {pc_sel, pc_ld, pc_inc, ir_ld}    = 5'b00_1_0_0;
               else
                  {pc_sel, pc_ld, pc_inc, ir_ld}    = 5'bxx_0_0_0;
               {im_cs, im_rd, im_wr}               = 3'b0_0_0;
               {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_x_0_xxxx; FS      = 5'h0;
               {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;        int_ack = int_ack;
               {io_cs, io_rd, io_wr}               = 3'b0_0_0;          fl_ld   = 1'b0;
               {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
               {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
               state = FETCH;
           end
       //ALU_Out <- RS($rs) - RT($rt)
       BNE1:
           begin
               @(negedge clk)
                {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
                {im_cs, im_rd, im_wr}                  = 3'b0_0_0;
                {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_x_0_xxxx; FS      = 5'h04;
                {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;        int_ack = int_ack;
                {io_cs, io_rd, io_wr}               = 3'b0_0_0;          fl_ld   = 1'b0;
                {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
                #1 {nsd, nsi, nsc, nsv, nsn, nsz}    = {DBZ, psi, C, V, N, Z};
               state = BNE2;
           end
       //if(psz==0) PC <- branch address, else PC <- PC
       BNE2:
           begin
            @(negedge clk)
               if(psz == 1'b0)
                  {pc_sel, pc_ld, pc_inc, ir_ld}    = 5'b00_1_0_0;
               else
                  {pc_sel, pc_ld, pc_inc, ir_ld}    = 5'bxx_0_0_0;
               {im_cs, im_rd, im_wr}               = 3'b0_0_0;
               {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_x_0_xxxx; FS      = 5'h0;
               {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;        int_ack = int_ack;
               {io_cs, io_rd, io_wr}               = 3'b0_0_0;          fl_ld   = 1'b0;
               {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
               {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
               state = FETCH;
           end
       //ALU_Out <- RS($rs) - RT($rt)
       BLEZ1:
```

```
    begin
       @(negedge clk)
         {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
         {im_cs, im_rd, im_wr}                  = 3'b0_0_0;
         {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_x_0_xxxx; FS     = 5'h0;
         {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;       int_ack = int_ack;
         {io_cs, io_rd, io_wr}               = 3'b0_0_0;         fl_ld   = 1'b0;
         {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
         #1 {nsd, nsi, nsc, nsv, nsn, nsz}    = {DBZ, psi, C, V, N, Z};
       state = BLEZ2;
    end
 //if(psz==1 | psn==1) PC <- branch address, else PC <- PC
 BLEZ2:
    begin
     @(negedge clk)
       if(psz == 1'b1 | psn == 1'b1)
          {pc_sel, pc_ld, pc_inc, ir_ld}    = 5'b00_1_0_0;
       else
          {pc_sel, pc_ld, pc_inc, ir_ld}    = 5'bxx_0_0_0;
       {im_cs, im_rd, im_wr}                = 3'b0_0_0;
       {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_x_0_xxxx; FS     = 5'h0;
       {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;       int_ack = int_ack;
       {io_cs, io_rd, io_wr}               = 3'b0_0_0;         fl_ld   = 1'b0;
       {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
       {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
       state = FETCH;
    end
 //ALU_Out <- RS($rs) - RT($rt)
 BLTZ1:
    begin
       @(negedge clk)
         {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
         {im_cs, im_rd, im_wr}                  = 3'b0_0_0;
         {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_x_0_xxxx; FS     = 5'h0;
         {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;       int_ack = int_ack;
         {io_cs, io_rd, io_wr}               = 3'b0_0_0;         fl_ld   = 1'b0;
         {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
         #1 {nsd, nsi, nsc, nsv, nsn, nsz}    = {DBZ, psi, C, V, N, Z};
       state = BLTZ2;
    end
 //if(psz==0 & psn==1) PC <- branch address, else PC <- PC
 BLTZ2:
    begin
     @(negedge clk)
       if(psz == 1'b0 & psn == 1'b1)
          {pc_sel, pc_ld, pc_inc, ir_ld}    = 5'b00_1_0_0;
       else
          {pc_sel, pc_ld, pc_inc, ir_ld}    = 5'bxx_0_0_0;
       {im_cs, im_rd, im_wr}                = 3'b0_0_0;
       {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_x_0_xxxx; FS     = 5'h0;
       {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;       int_ack = int_ack;
       {io_cs, io_rd, io_wr}               = 3'b0_0_0;         fl_ld   = 1'b0;
       {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
       {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
       state = FETCH;
    end
 //ALU_Out <- RS($rs) - RT($rt)
 BGTZ1:
    begin
       @(negedge clk)
         {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
         {im_cs, im_rd, im_wr}                  = 3'b0_0_0;
         {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_x_0_xxxx; FS     = 5'h0;
         {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;       int_ack = int_ack;
         {io_cs, io_rd, io_wr}               = 3'b0_0_0;         fl_ld   = 1'b0;
         {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
         #1 {nsd, nsi, nsc, nsv, nsn, nsz}    = {DBZ, psi, C, V, N, Z};
```

```
              state = BGTZ2;
          end
        //if(psz==0 & psn==0) PC <- branch address, else PC <- PC
        BGTZ2:
          begin
           @(negedge clk)
              if(psz == 1'b0 & psn == 1'b0)
                 {pc_sel, pc_ld, pc_inc, ir_ld}    = 5'b00_1_0_0;
              else
                 {pc_sel, pc_ld, pc_inc, ir_ld}    = 5'bxx_0_0_0;
              {im_cs, im_rd, im_wr}               = 3'b0_0_0;
              {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_x_0_xxxx; FS     = 5'h0;
              {dm_cs, dm_rd, dm_wr}              = 3'b0_0_0;       int_ack = int_ack;
              {io_cs, io_rd, io_wr}              = 3'b0_0_0;          fl_ld   = 1'b0;
              {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
              {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
              state = FETCH;
          end
        //ALU_Out <- RS($rs) - RT($rt)
        BGEZ1:
          begin
              @(negedge clk)
               {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
               {im_cs, im_rd, im_wr}                  = 3'b0_0_0;
               {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_x_0_xxxx; FS     = 5'h0;
               {dm_cs, dm_rd, dm_wr}              = 3'b0_0_0;       int_ack = int_ack;
               {io_cs, io_rd, io_wr}              = 3'b0_0_0;          fl_ld   = 1'b0;
               {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
               #1 {nsd, nsi, nsc, nsv, nsn, nsz}    = {DBZ, psi, C, V, N, Z};
              state = BGEZ2;
          end
        //if(psz==1 & psn==0) PC <- branch address, else PC <- PC
        BGEZ2:
          begin
           @(negedge clk)
              if(psz == 1'b1 | psn == 1'b0)
                 {pc_sel, pc_ld, pc_inc, ir_ld}    = 5'b00_1_0_0;
              else
                 {pc_sel, pc_ld, pc_inc, ir_ld}    = 5'bxx_0_0_0;
              {im_cs, im_rd, im_wr}               = 3'b0_0_0;
              {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_x_0_xxxx; FS     = 5'h0;
              {dm_cs, dm_rd, dm_wr}              = 3'b0_0_0;       int_ack = int_ack;
              {io_cs, io_rd, io_wr}              = 3'b0_0_0;          fl_ld   = 1'b0;
              {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
              {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
              state = FETCH;
          end
        //if(psd==1) Pc <- brnahc address, else PC <- PC
        BDT:
          begin
            @(negedge clk)
              if(psd == 1'b1)
                 {pc_sel, pc_ld, pc_inc, ir_ld}    = 5'b00_1_0_0;
              else
                 {pc_sel, pc_ld, pc_inc, ir_ld}    = 5'bxx_0_0_0;
              {im_cs, im_rd, im_wr}               = 3'b0_0_0;
              {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_x_0_xxxx; FS     = 5'h0;
              {dm_cs, dm_rd, dm_wr}              = 3'b0_0_0;       int_ack = int_ack;
              {io_cs, io_rd, io_wr}              = 3'b0_0_0;          fl_ld   = 1'b0;
              {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
              {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
              state = FETCH;
          end
        //if(psd==0) Pc <- brnahc address, else PC <- PC
        BDF:
          begin
            @(negedge clk)
              if(psd == 1'b0)
                 {pc_sel, pc_ld, pc_inc, ir_ld}    = 5'b00_1_0_0;
              else
                 {pc_sel, pc_ld, pc_inc, ir_ld}    = 5'bxx_0_0_0;
              {im_cs, im_rd, im_wr}                = 3'b0_0_0;
              {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_x_0_xxxx; FS     = 5'h0;
              {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;       int_ack = int_ack;
              {io_cs, io_rd, io_wr}               = 3'b0_0_0;          fl_ld   = 1'b0;
              {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
              {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
              state = FETCH;
          end
        //ALU_Out <- RS($rs) + S.E.IR[15:0]
        ADDI:
          begin
              @(negedge clk)
               {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
               {im_cs, im_rd, im_wr}                  = 3'b0_0_0;
               {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_1_0_xxxx; FS     = 5'h02;
               {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;       int_ack = int_ack;
               {io_cs, io_rd, io_wr}               = 3'b0_0_0;          fl_ld   = 1'b0;
               {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
               #1 {nsd, nsi, nsc, nsv, nsn, nsz}    = {DBZ, psi, C, V, N, Z};
              state = WB_imm;
          end
        //ALU_Out <- RS($rs) & {16'b0, RT[15:0]}
        ANDI:
          begin
              @(negedge clk)
               {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
               {im_cs, im_rd, im_wr}                  = 3'b0_0_0;
               {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_x_0_xxxx; FS     = 5'h16;
               {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;       int_ack = int_ack;
               {io_cs, io_rd, io_wr}               = 3'b0_0_0;          fl_ld   = 1'b0;
               {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
               #1 {nsd, nsi, nsc, nsv, nsn, nsz}    = {DBZ, psi, C, V, N, Z};
              state = WB_imm;
          end
        //ALU_Out <- RS($rs) | {16'b0, RT[15:0]}
        ORI:
          begin
              @(negedge clk)
               {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
               {im_cs, im_rd, im_wr}                  = 3'b0_0_0;
               {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_x_0_xxxx; FS     = 5'h17;
               {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;       int_ack = int_ack;
               {io_cs, io_rd, io_wr}               = 3'b0_0_0;          fl_ld   = 1'b0;
               {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
               #1 {nsd, nsi, nsc, nsv, nsn, nsz}    = {DBZ, psi, C, V, N, Z};
              state = WB_imm;
          end
        //ALU_Out <- RS($rs) ^ {16'b0, RT[15:0]}
        XORI:
          begin
              @(negedge clk)
               {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
               {im_cs, im_rd, im_wr}                  = 3'b0_0_0;
               {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_x_0_xxxx; FS     = 5'h18;
               {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;       int_ack = int_ack;
               {io_cs, io_rd, io_wr}               = 3'b0_0_0;          fl_ld   = 1'b0;
               {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
               #1 {nsd, nsi, nsc, nsv, nsn, nsz}    = {DBZ, psi, C, V, N, Z};
              state = WB_imm;
          end
        //ALU_Out <- {RT[15:0], 16'b0}
        LUI:
          begin
```

```
            @(negedge clk)
              {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
              {im_cs, im_rd, im_wr}                  = 3'b0_0_0;
              {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_x_0_xxxx; FS     = 5'h19;
              {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;        int_ack = int_ack;
              {io_cs, io_rd, io_wr}               = 3'b0_0_0;          fl_ld   = 1'b0;
              {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
              #1 {nsd, nsi, nsc, nsv, nsn, nsz}    = {DBZ, psi, C, V, N, Z};
            state = WB_imm;
        end
    //if(R[$rs] < S.E.IR[15:0], ALU_Out <- 1, else ALU_Out <- 0   signed
    SLTI:
        begin
            @(negedge clk)
              {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
              {im_cs, im_rd, im_wr}                  = 3'b0_0_0;
              {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_x_0_xxxx; FS     = 5'h06;
              {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;        int_ack = int_ack;
              {io_cs, io_rd, io_wr}               = 3'b0_0_0;          fl_ld   = 1'b0;
              {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
              #1 {nsd, nsi, nsc, nsv, nsn, nsz}    = {DBZ, psi, C, V, N, Z};
            state = WB_imm;
        end
    //if(R[$rs] < S.E.IR[15:0], ALU_Out <- 1, else ALU_Out <- 0   unsigned
    SLTUI:
        begin
            @(negedge clk)
              {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
              {im_cs, im_rd, im_wr}                  = 3'b0_0_0;
              {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_x_0_xxxx; FS     = 5'h07;
              {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;        int_ack = int_ack;
              {io_cs, io_rd, io_wr}               = 3'b0_0_0;          fl_ld   = 1'b0;
              {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
              #1 {nsd, nsi, nsc, nsv, nsn, nsz}    = {DBZ, psi, C, V, N, Z};
            state = WB_imm;
        end
    //ALU_Out <- RS($rs) + RT(S.E.IR[15:0])
    LW:
        begin
            @(negedge clk)
              {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
              {im_cs, im_rd, im_wr}                  = 3'b0_0_0;
              {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_0_0_xxxx; FS     = 5'h02;
              {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;        int_ack = int_ack;
              {io_cs, io_rd, io_wr}               = 3'b0_0_0;          fl_ld   = 1'b0;
              {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
              #1 {nsd, nsi, nsc, nsv, nsn, nsz}    = {DBZ, psi, C, V, N, Z};
            state = MEM_ACC;
        end
    //Din <- M[ALU_Out]
    MEM_ACC:
        begin
            @(negedge clk)
              {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
              {im_cs, im_rd, im_wr}                  = 3'b0_0_0;
              {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_x_0_0000; FS     = 5'h0;
              {dm_cs, dm_rd, dm_wr}               = 3'b1_1_0;        int_ack = int_ack;
              {io_cs, io_rd, io_wr}               = 3'b0_0_0;          fl_ld   = 1'b0;
              {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
              {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
            state = WB_din;
        end
    //ALU_Out <- RS($rs) + RT(S.E.IR[15:0])
    INPUT:
        begin
            @(negedge clk)
              {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
```

```
          {im_cs, im_rd, im_wr}                   = 3'b0_0_0;
          {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_0_0_xxxx; FS     = 5'h02;
          {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;        int_ack = int_ack;
          {io_cs, io_rd, io_wr}               = 3'b0_0_0;          fl_ld   = 1'b0;
          {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
          #1 {nsd, nsi, nsc, nsv, nsn, nsz}    = {DBZ, psi, C, V, N, Z};
        state = IO_ACC;
    end
 //IO_in <- M[ALU_Out]
 IO_ACC:
    begin
        @(negedge clk)
          {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
          {im_cs, im_rd, im_wr}                   = 3'b0_0_0;
          {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_x_0_0000; FS     = 5'h0;
          {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;        int_ack = int_ack;
          {io_cs, io_rd, io_wr}               = 3'b1_1_0;          fl_ld   = 1'b0;
          {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
          {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
        state = WB_io_in;
    end
 //ALU_Out <- RS($rs) + RT(S.E.IR[15:0])
 SW:
    begin
        @(negedge clk)
          {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
          {im_cs, im_rd, im_wr}                   = 3'b0_0_0;
          {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_0_0_xxxx; FS     = 5'h02;
          {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;        int_ack = int_ack;
          {io_cs, io_rd, io_wr}               = 3'b0_0_0;         fl_ld   = 1'b0;
          {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
          #1 {nsd, nsi, nsc, nsv, nsn, nsz}    = {DBZ, psi, C, V, N, Z};
        state = WB_mem;
    end
 //ALU_Out <- RS($rs) + RT(S.E.IR[15:0])
 OUTPUT:
    begin
        @(negedge clk)
          {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
          {im_cs, im_rd, im_wr}                   = 3'b0_0_0;
          {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_0_0_xxxx; FS     = 5'h02;
          {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;        int_ack = int_ack;
          {io_cs, io_rd, io_wr}               = 3'b0_0_0;          fl_ld   = 1'b0;
          {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
          #1 {nsd, nsi, nsc, nsv, nsn, nsz}    = {DBZ, psi, C, V, N, Z};
        state = WB_io_mem;
    end
 // ALU_Out <- (RS[imm] = 1)
BSI:
    begin
     @(negedge clk)
      {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
      {im_cs, im_rd, im_wr}                   = 3'b0_0_0;
      {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_x_0_xxxx; FS          = 5'h1A;
      {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;        int_ack     = int_ack;
      {io_cs, io_rd, io_wr}               = 3'b0_0_0;          fl_ld       = 1'b0;
      {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
      {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
      state = WB_imm;
    end
// ALU_Out <- (RS[imm] = 0)
BCI:
    begin
     @(negedge clk)
      {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
      {im_cs, im_rd, im_wr}                   = 3'b0_0_0;
      {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_x_0_xxxx; FS          = 5'h1B;
```

```
            {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;        int_ack     = int_ack;
            {io_cs, io_rd, io_wr}               = 3'b0_0_0;           fl_ld       = 1'b0;
            {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
            {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
            state = WB_imm;
          end
        // ALU_Out <- reversed(RS)
        RB:
          begin
           @(negedge clk)
            {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
            {im_cs, im_rd, im_wr}                   = 3'b0_0_0;
            {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_x_0_xxxx; FS          = 5'h1C;
            {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;        int_ack     = int_ack;
            {io_cs, io_rd, io_wr}               = 3'b0_0_0;           fl_ld       = 1'b0;
            {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
            {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
            state = WB_alu;
          end
        // ALU_Out <- reverse endianness(RS)
        RBO:
          begin
           @(negedge clk)
            {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
            {im_cs, im_rd, im_wr}                   = 3'b0_0_0;
            {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_x_0_xxxx; FS          = 5'h1D;
            {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;        int_ack     = int_ack;
            {io_cs, io_rd, io_wr}               = 3'b0_0_0;           fl_ld       = 1'b0;
            {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
            {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
            state = WB_alu;
          end
        // ALU_Out <- RT RR shamt
        RR:
          begin
           @(negedge clk)
            {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
            {im_cs, im_rd, im_wr}                   = 3'b0_0_0;
            {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_x_0_xxxx; FS          = 5'h00;
            {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;        int_ack     = int_ack;
            {io_cs, io_rd, io_wr}               = 3'b0_0_0;           fl_ld       = 1'b0;
            {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_11_00_00; HILO_ld_SIMD = 1'b0;
            {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
            state = WB_sh;
          end
         // used for enhancements to the baseline, a.k.a. E-type
        E_KEY:
          begin
           @(negedge clk)
              if(IR[1:0] == 2'b00) begin // Standard
                   {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'b00_0_0_0;
                   {im_cs, im_rd, im_wr}                   = 3'b0_0_0;
                   {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_00_0_0_0000;  FS          = 5'h0;
                   {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;         int_ack     = int_ack;
                   {io_cs, io_rd, io_wr}               = 3'b0_0_0;             fl_ld       = 1'b0;
                   {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00;  HILO_ld_SIMD = 1'b0;
                   {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
                   state = STND;
              end
              else if(IR[1:0] == 2'b01) begin // SIMD8
                   {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'b00_0_0_0;
                   {im_cs, im_rd, im_wr}                   = 3'b0_0_0;
                   {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_00_0_0_0000;  FS          = 5'h0;
                   {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;         int_ack     = int_ack;
                   {io_cs, io_rd, io_wr}               = 3'b0_0_0;             fl_ld       = 1'b0;
                   {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00;  HILO_ld_SIMD = 1'b0;
                   {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
```

```
                    state = SIMD8;
                end
                else if(IR[1:0] == 2'b10) begin // SIMD16
                    {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'b00_0_0_0;
                    {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
                    {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_00_0_0_0000;  FS          = 5'h0;
                    {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;        int_ack     = int_ack;
                    {io_cs, io_rd, io_wr}               = 3'b0_0_0;            fl_ld       = 1'b0;
                    {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00;  HILO_ld_SIMD = 1'b0;
                    {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
                    state = SIMD16;
                end
            end
        STND:
            begin

            end
        SIMD8:
            begin
              @(negedge clk)
                {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'b00_0_0_0;
                {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
                {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_00_0_0_0000;  FS          = 5'h0;
                {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;        int_ack     = int_ack;
                {io_cs, io_rd, io_wr}               = 3'b0_0_0;            fl_ld       = 1'b0;
                {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00;  HILO_ld_SIMD = 1'b0;
                {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
                case(IR[10:6])
                  5'h02:
                    state = ADD8;
                  5'h03:
                    state = SUB8;
                  5'h04:
                    state = AND8;
                  5'h05:
                    state = OR8;
                  5'h06:
                    state = NOT8;
                  5'h07:
                    state = XOR8;
                  5'h08:
                    state = MUL8;
                  5'h09:
                    state = DIV8;
                  5'h0A:
                    state = VMFHI;
                  5'h0B:
                    state = VMFLO;
                  default:
                    state = ILLEGAL_OP;
                endcase
            end
        SIMD16:
            begin
              @(negedge clk)
                {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'b00_0_0_0;
                {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
                {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_00_0_0_0000;  FS          = 5'h0;
                {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;        int_ack     = int_ack;
                {io_cs, io_rd, io_wr}               = 3'b0_0_0;            fl_ld       = 1'b0;
                {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00;  HILO_ld_SIMD = 1'b0;
                {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
                case(IR[10:6])
                  5'h02:
                    state = ADD16;
                  5'h03:
                    state = SUB16;
            5'h04:
               state = AND16;
            5'h05:
               state = OR16;
            5'h06:
               state = NOT16;
            5'h07:
               state = XOR16;
            5'h08:
               state = MUL16;
            5'h09:
               state = DIV16;
            5'h0A:
               state = VMFHI;
            5'h0B:
               state = VMFLO;
            default:
               state = ILLEGAL_OP;
          endcase
      end
  ADD8:
      begin
          @(negedge clk)
            {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'b00_0_0_0;
            {im_cs, im_rd, im_wr}                      = 3'b0_0_0;
            {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_00_0_0_0000;  FS          = 5'h02;
            {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;         int_ack     = int_ack;
            {io_cs, io_rd, io_wr}               = 3'b0_0_0;           fl_ld       = 1'b0;
            {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_01_00;  HILO_ld_SIMD = 1'b0;
            {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
            state = WB_SIMD_alu;
      end
  SUB8:
      begin
          @(negedge clk)
            {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'b00_0_0_0;
            {im_cs, im_rd, im_wr}                      = 3'b0_0_0;
            {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_00_0_0_0000;  FS          = 5'h03;
            {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;         int_ack     = int_ack;
            {io_cs, io_rd, io_wr}               = 3'b0_0_0;           fl_ld       = 1'b0;
            {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_01_00;  HILO_ld_SIMD = 1'b0;
            {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
            state = WB_SIMD_alu;
      end
  AND8:
      begin
          @(negedge clk)
            {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'b00_0_0_0;
            {im_cs, im_rd, im_wr}                      = 3'b0_0_0;
            {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_00_0_0_0000;  FS          = 5'h04;
            {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;         int_ack     = int_ack;
            {io_cs, io_rd, io_wr}               = 3'b0_0_0;           fl_ld       = 1'b0;
            {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_01_00;  HILO_ld_SIMD = 1'b0;
            {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
            state = WB_SIMD_alu;
      end
 OR8:
      begin
          @(negedge clk)
            {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'b00_0_0_0;
            {im_cs, im_rd, im_wr}                      = 3'b0_0_0;
            {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_00_0_0_0000;  FS          = 5'h05;
            {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;         int_ack     = int_ack;
            {io_cs, io_rd, io_wr}               = 3'b0_0_0;           fl_ld       = 1'b0;
            {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_01_00;  HILO_ld_SIMD = 1'b0;
            {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
            state = WB_SIMD_alu;
      end
 NOT8:
      begin
          @(negedge clk)
            {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'b00_0_0_0;
            {im_cs, im_rd, im_wr}                   = 3'b0_0_0;
            {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_00_0_0_0000;  FS          = 5'h06;
            {dm_cs, dm_rd, dm_wr}                = 3'b0_0_0;         int_ack     = int_ack;
            {io_cs, io_rd, io_wr}                = 3'b0_0_0;           fl_ld       = 1'b0;
            {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_01_00;  HILO_ld_SIMD = 1'b0;
            {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
            state = WB_SIMD_alu;
      end
  XOR8:
   begin
          @(negedge clk)
            {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'b00_0_0_0;
            {im_cs, im_rd, im_wr}                   = 3'b0_0_0;
            {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_00_0_0_0000;  FS          = 5'h07;
            {dm_cs, dm_rd, dm_wr}                = 3'b0_0_0;         int_ack     = int_ack;
            {io_cs, io_rd, io_wr}                = 3'b0_0_0;           fl_ld       = 1'b0;
            {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_01_00;  HILO_ld_SIMD = 1'b0;
            {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
            state = WB_SIMD_alu;
  end
  MUL8:
  begin
      @(negedge clk)
            {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'b00_0_0_0;
            {im_cs, im_rd, im_wr}                   = 3'b0_0_0;
            {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_00_0_0_0000;  FS          = 5'h00;
            {dm_cs, dm_rd, dm_wr}                = 3'b0_0_0;         int_ack     = int_ack;
            {io_cs, io_rd, io_wr}                = 3'b0_0_0;           fl_ld       = 1'b0;
            {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_01_01;  HILO_ld_SIMD = 1'b1;
            {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
            state = FETCH;
  end
  DIV8:
      begin
          @(negedge clk)
            {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'b00_0_0_0;
            {im_cs, im_rd, im_wr}                   = 3'b0_0_0;
            {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_00_0_0_0000;  FS          = 5'h00;
            {dm_cs, dm_rd, dm_wr}                = 3'b0_0_0;         int_ack     = int_ack;
            {io_cs, io_rd, io_wr}                = 3'b0_0_0;           fl_ld       = 1'b0;
            {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_01_10;  HILO_ld_SIMD = 1'b1;
            {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
            state = FETCH;
      end
  ADD16:
      begin
          @(negedge clk)
            {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'b00_0_0_0;
            {im_cs, im_rd, im_wr}                   = 3'b0_0_0;
            {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_00_0_0_0000;  FS          = 5'h02;
            {dm_cs, dm_rd, dm_wr}                = 3'b0_0_0;         int_ack     = int_ack;
            {io_cs, io_rd, io_wr}                = 3'b0_0_0;           fl_ld       = 1'b0;
            {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_10_00;  HILO_ld_SIMD = 1'b0;
            {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
            state = WB_SIMD_alu;
      end
  SUB16:
      begin
          @(negedge clk)
            {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'b00_0_0_0;
            {im_cs, im_rd, im_wr}                   = 3'b0_0_0;
            {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_00_0_0_0000;  FS          = 5'h03;
            {dm_cs, dm_rd, dm_wr}                = 3'b0_0_0;         int_ack     = int_ack;
            {io_cs, io_rd, io_wr}                = 3'b0_0_0;          fl_ld       = 1'b0;
            {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_10_00;  HILO_ld_SIMD = 1'b0;
            {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
            state = WB_SIMD_alu;
      end
  AND16:
      begin
          @(negedge clk)
            {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'b00_0_0_0;
            {im_cs, im_rd, im_wr}                    = 3'b0_0_0;
            {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_00_0_0_0000;  FS          = 5'h04;
            {dm_cs, dm_rd, dm_wr}                = 3'b0_0_0;         int_ack     = int_ack;
            {io_cs, io_rd, io_wr}                = 3'b0_0_0;          fl_ld       = 1'b0;
            {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_10_00;  HILO_ld_SIMD = 1'b0;
            {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
            state = WB_SIMD_alu;
      end
  OR16:
      begin
          @(negedge clk)
            {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'b00_0_0_0;
            {im_cs, im_rd, im_wr}                    = 3'b0_0_0;
            {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_00_0_0_0000;  FS          = 5'h05;
            {dm_cs, dm_rd, dm_wr}                = 3'b0_0_0;         int_ack     = int_ack;
            {io_cs, io_rd, io_wr}                = 3'b0_0_0;          fl_ld       = 1'b0;
            {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_10_00;  HILO_ld_SIMD = 1'b0;
            {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
            state = WB_SIMD_alu;
      end
  NOT16:
      begin
          @(negedge clk)
            {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'b00_0_0_0;
            {im_cs, im_rd, im_wr}                    = 3'b0_0_0;
            {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_00_0_0_0000;  FS          = 5'h06;
            {dm_cs, dm_rd, dm_wr}                = 3'b0_0_0;         int_ack     = int_ack;
            {io_cs, io_rd, io_wr}                = 3'b0_0_0;          fl_ld       = 1'b0;
            {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_10_00;  HILO_ld_SIMD = 1'b0;
            {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
            state = WB_SIMD_alu;
      end
  XOR16:
      begin
          @(negedge clk)
            {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'b00_0_0_0;
            {im_cs, im_rd, im_wr}                    = 3'b0_0_0;
            {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_00_0_0_0000;  FS          = 5'h07;
            {dm_cs, dm_rd, dm_wr}                = 3'b0_0_0;         int_ack     = int_ack;
            {io_cs, io_rd, io_wr}                = 3'b0_0_0;          fl_ld       = 1'b0;
            {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_10_00;  HILO_ld_SIMD = 1'b0;
            {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
            state = WB_SIMD_alu;
      end
  MUL16:
      begin
          @(negedge clk)
            {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'b00_0_0_0;
            {im_cs, im_rd, im_wr}                    = 3'b0_0_0;
            {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_00_0_0_0000;  FS          = 5'h00;
            {dm_cs, dm_rd, dm_wr}                = 3'b0_0_0;         int_ack     = int_ack;
            {io_cs, io_rd, io_wr}                = 3'b0_0_0;          fl_ld       = 1'b0;
            {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_10_01;  HILO_ld_SIMD = 1'b1;
            {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
            state = FETCH;
      end
  DIV16:
      begin
          @(negedge clk)
            {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'b00_0_0_0;
            {im_cs, im_rd, im_wr}                   = 3'b0_0_0;
            {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_00_0_0_0000;  FS          = 5'h00;
            {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;         int_ack     = int_ack;
            {io_cs, io_rd, io_wr}               = 3'b0_0_0;           fl_ld       = 1'b0;
            {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_10_10;  HILO_ld_SIMD = 1'b1;
            {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
            state = FETCH;
      end
  // R[$rd] <- ALU_out(SIMD)
  WB_SIMD_alu:
      begin
          @(negedge clk)
            {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'b00_0_0_0;
            {im_cs, im_rd, im_wr}                   = 3'b0_0_0;
            {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b1_00_0_0_0111;  FS          = 5'h0;
            {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;         int_ack     = int_ack;
            {io_cs, io_rd, io_wr}               = 3'b0_0_0;           fl_ld       = 1'b0;
            {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00;  HILO_ld_SIMD = 1'b0;
            {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
            state = FETCH;
      end
  // R[$rd] <- ALU_out(SIMDHI)
  VMFHI:
      begin
          @(negedge clk)
            {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'b00_0_0_0;
            {im_cs, im_rd, im_wr}                   = 3'b0_0_0;
            {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b1_00_0_0_1000;  FS          = 5'h0;
            {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;         int_ack     = int_ack;
            {io_cs, io_rd, io_wr}               = 3'b0_0_0;           fl_ld       = 1'b0;
            {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00;  HILO_ld_SIMD = 1'b0;
            {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
            state = FETCH;
      end
  // R[$rd] <- ALU_out(SIMDLO)
  VMFLO:
      begin
          @(negedge clk)
            {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'b00_0_0_0;
            {im_cs, im_rd, im_wr}                   = 3'b0_0_0;
            {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b1_00_0_0_1001;  FS          = 5'h0;
            {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;         int_ack     = int_ack;
            {io_cs, io_rd, io_wr}               = 3'b0_0_0;           fl_ld       = 1'b0;
            {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00;  HILO_ld_SIMD = 1'b0;
            {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
            state = FETCH;
      end
  //R[$rd] <- ALU_Out
  WB_alu:
      begin
          @(negedge clk)
            {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
            {im_cs, im_rd, im_wr}                  = 3'b0_0_0;
            {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b1_00_x_0_0000; FS     = 5'h0;
            {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;        int_ack = int_ack;
            {io_cs, io_rd, io_wr}               = 3'b0_0_0;         fl_ld   = 1'b0;
            {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
            {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
          state = FETCH;
      end
  //R[$rt] <- ALU_Out
  WB_imm:
      begin
          @(negedge clk)
```

```
                {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
                {im_cs, im_rd, im_wr}                  = 3'b0_0_0;
                {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b1_01_x_0_0000; FS     = 5'h0;
                {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;       int_ack = int_ack;
                {io_cs, io_rd, io_wr}               = 3'b0_0_0;         fl_ld   = 1'b0;
                {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
                {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
              state = FETCH;
            end
          //R[$rd] <- HI
          WB_hi:
            begin
              @(negedge clk)
                {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
                {im_cs, im_rd, im_wr}                  = 3'b0_0_0;
                {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b1_00_x_0_0001; FS     = 5'h0;
                {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;       int_ack = int_ack;
                {io_cs, io_rd, io_wr}               = 3'b0_0_0;         fl_ld   = 1'b0;
                {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
                {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
              state = FETCH;
            end
          //R[$rd] <- LO
          WB_lo:
            begin
              @(negedge clk)
                {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
                {im_cs, im_rd, im_wr}                  = 3'b0_0_0;
                {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b1_00_x_0_0010; FS     = 5'h0;
                {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;       int_ack = int_ack;
                {io_cs, io_rd, io_wr}               = 3'b0_0_0;         fl_ld   = 1'b0;
                {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
                {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
              state = FETCH;
            end
          //dM[ALU_Out] <- RT
          WB_mem:
            begin
              @(negedge clk)
                {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
                {im_cs, im_rd, im_wr}                  = 3'b0_0_0;
                {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_0_0_0000; FS     = 5'h0;
                {dm_cs, dm_rd, dm_wr}               = 3'b1_0_1;       int_ack = int_ack;
                {io_cs, io_rd, io_wr}               = 3'b0_0_0;         fl_ld   = 1'b0;
                {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
                {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
              state = FETCH;
            end
          //ioM[ALU_Out] <- RT
          WB_io_mem:
            begin
              @(negedge clk)
                {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
                {im_cs, im_rd, im_wr}                  = 3'b0_0_0;
                {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_0_0_0000; FS     = 5'h0;
                {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;       int_ack = int_ack;
                {io_cs, io_rd, io_wr}               = 3'b1_0_1;         fl_ld   = 1'b0;
                {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
                {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
              state = FETCH;
            end
          //R[$rt] <- D_in
          WB_din:
            begin
              @(negedge clk)
                {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
                {im_cs, im_rd, im_wr}                  = 3'b0_0_0;
                {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b1_01_x_0_0011; FS     = 5'h0;
                {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;       int_ack = int_ack;
                {io_cs, io_rd, io_wr}               = 3'b0_0_0;         fl_ld   = 1'b0;
                {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
                {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
              state = FETCH;
            end
          //R[$rt] <- IO_in
          WB_io_in:
            begin
              @(negedge clk)
                {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
                {im_cs, im_rd, im_wr}                  = 3'b0_0_0;
                {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b1_01_x_0_0101; FS     = 5'h0;
                {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;       int_ack = int_ack;
                {io_cs, io_rd, io_wr}               = 3'b0_0_0;         fl_ld   = 1'b0;
                {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
                {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
              state = FETCH;
            end
          // R[$rd] <- SH_out
          WB_sh:
           begin
            @(negedge clk)
              {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
              {im_cs, im_rd, im_wr}                  = 3'b0_0_0;
              {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b1_00_x_0_0110; FS     = 5'h0;
              {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;       int_ack =  int_ack;
              {io_cs, io_rd, io_wr}               = 3'b0_0_0;         fl_ld   = 1'b0;
              {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
              {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
              state = FETCH;
           end
          //DumpReg, DumpMem, finish
          BREAK:
            begin
              $display("BREAK INSTRUCTION FETCHED %t", $time);
              // deassert everything
              @(negedge clk)
                {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'b00_0_0_0;
                {im_cs, im_rd, im_wr}                  = 3'b0_0_0;
                {D_En, DA_sel, T_sel, HILO_ld, Y_sel}  = 9'b0_00_0_0_0000; FS     = 5'h0;
                {dm_cs, dm_rd, dm_wr}                = 3'b0_0_0;       int_ack = int_ack;
                {io_cs, io_rd, io_wr}                = 3'b0_0_0;           fl_ld   = 1'b0;
                {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}    = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
                {nsd, nsi, nsc, nsv, nsn, nsz}        = {psd, psi, psc, psv, psn, psz};
              $display("D I S P L A Y I N G   R E G I S T E R S");
              regDump();
              $display(" ");
              memDump();
              $finish;
            end
          //DumpReg, DumpPC/IR, finish
          ILLEGAL_OP:
            begin
              $display("ILLEGAL OPCODE FETCHED %t", $time);
              // deassert everything
              @(negedge clk)
                {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'b00_0_0_0;
                {im_cs, im_rd, im_wr}                  = 3'b0_0_0;
                {D_En, DA_sel, T_sel, HILO_ld, Y_sel}  = 9'b0_00_0_0_0000; FS     = 5'h0;
                {dm_cs, dm_rd, dm_wr}                = 3'b0_0_0;       int_ack = int_ack;
                {io_cs, io_rd, io_wr}                = 3'b0_0_0;           fl_ld   = 1'b0;
                {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}    = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
                {nsd, nsi, nsc, nsv, nsn, nsz}        = {psd, psi, psc, psv, psn, psz};
              $display("D I S P L A Y I N G   R E G I S T E R S"); $display(" ");
              regDump();
              pc_ir_Dump();
              $finish;
           end
          //RS <- $sp
          INTR_1:
           begin
              @(negedge clk)
                 {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
                 {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
                 {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_x_0_xxxx; FS     = 5'h0;
                 {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;        int_ack = int_ack;
                 {io_cs, io_rd, io_wr}               = 3'b0_0_0;           fl_ld   = 1'b0;
                 {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b1_00_00_00_00; HILO_ld_SIMD = 1'b0;
                 {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
              state = INTR_2;
              #1 $display("Current flags before intr: psd=%b,psi=%b,psc=%b,psv=%b,psn=%b,psz=%b",
                          psd, psi, psc, psv, psn, psz);
           end
          //ALU_Out <- $sp - 4
          INTR_2:
           begin
              @(negedge clk)
                 {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
                 {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
                 {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_x_0_xxxx; FS     = 5'h12;
                 {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;        int_ack = int_ack;
                 {io_cs, io_rd, io_wr}               = 3'b0_0_0;           fl_ld   = 1'b0;
                 {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
                 {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
              state = INTR_3;
           end
          //dM[$sp] <- PC_in, $sp <- ALU_Out
          INTR_3:
           begin
              @(negedge clk)
                 {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
                 {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
                 {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b1_11_x_0_0000; FS     = 5'h0;
                 {dm_cs, dm_rd, dm_wr}               = 3'b1_0_1;        int_ack = int_ack;
                 {io_cs, io_rd, io_wr}               = 3'b0_0_0;           fl_ld   = 1'b0;
                 {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_01_00_00_00; HILO_ld_SIMD = 1'b0;
                 {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
              state = INTR_4;
           end
          //RS <- $sp
          INTR_4:
           begin
              @(negedge clk)
                 {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
                 {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
                 {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_x_0_xxxx; FS     = 5'h0;
                 {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;        int_ack = int_ack;
                 {io_cs, io_rd, io_wr}               = 3'b0_0_0;           fl_ld   = 1'b0;
                 {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b1_00_00_00_00; HILO_ld_SIMD = 1'b0;
                 {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
              state = INTR_5;
           end
          //ALU_Out <- $sp - 4
          INTR_5:
           begin
              @(negedge clk)
                 {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
                 {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
                 {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_x_0_xxxx; FS     = 5'h12;
                 {dm_cs, dm_rd, dm_wr}               = 3'b0_0_0;        int_ack = int_ack;
                 {io_cs, io_rd, io_wr}               = 3'b0_0_0;           fl_ld   = 1'b0;
                 {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
```

```
            {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
         state = INTR_6;
      end
   //dM[$sp] <- flags, $sp <- ALU_Out
   INTR_6:
      begin
         @(negedge clk)
            {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
            {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
            {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b1_11_x_0_0000; FS     = 5'h0;
            {dm_cs, dm_rd, dm_wr}                = 3'b1_0_1;        int_ack = int_ack;
            {io_cs, io_rd, io_wr}                = 3'b0_0_0;           fl_ld   = 1'b0;
            {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_10_00_00_00; HILO_ld_SIMD = 1'b0;
            {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
         state = INTR_7;
      end
   //ALU_Out <- SP_INIT(3FC)
   INTR_7:
      begin
         @(negedge clk)
            {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
            {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
            {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_x_0_xxxx; FS     = 5'h15;
            {dm_cs, dm_rd, dm_wr}                = 3'b0_0_0;        int_ack = int_ack;
            {io_cs, io_rd, io_wr}                = 3'b0_0_0;           fl_ld   = 1'b0;
            {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
            {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
         state = INTR_8;
      end
   //D_in <- dM[3FC]
   INTR_8:
      begin
         @(negedge clk)
            {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
            {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
            {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_x_0_0000; FS     = 5'h0;
            {dm_cs, dm_rd, dm_wr}                = 3'b1_1_0;        int_ack = int_ack;
            {io_cs, io_rd, io_wr}                = 3'b0_0_0;           fl_ld   = 1'b0;
            {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
            {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
         state = INTR_9;
      end
   //PC <- D_in, psi <- 0
   INTR_9:
      begin
         @(negedge clk)
            {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'b10_1_0_0;
            {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
            {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_x_0_0011; FS     = 5'h0;
            {dm_cs, dm_rd, dm_wr}                = 3'b0_0_0;        int_ack = 1'b1;
            {io_cs, io_rd, io_wr}                = 3'b0_0_0;           fl_ld   = 1'b0;
            {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
            #1 {nsd, nsi, nsc, nsv, nsn, nsz}    = {psd, 1'b0, psc, psv, psn, psz};
         state = FETCH;
         dmemDump();
      end
   //ALU_Out <- RS
   RETI_1:
      begin
         @(negedge clk)
            {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
            {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
            {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_x_0_xxxx; FS     = 5'h0;
            {dm_cs, dm_rd, dm_wr}                = 3'b0_0_0;        int_ack = int_ack;
            {io_cs, io_rd, io_wr}                = 3'b0_0_0;           fl_ld   = 1'b0;
            {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
            {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
            #1 $display("Flags before return: psd=%b,psi=%b,psc=%b,psv=%b,psn=%b,psz=%b",
                        psd, psi, psc, psv, psn, psz);
         state = RETI_2;
      end
   //D_in <- dM[ALU_Out], RS <- $sp
   RETI_2:
      begin
         @(negedge clk)
            {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
            {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
            {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_x_0_0000; FS     = 5'h0;
            {dm_cs, dm_rd, dm_wr}              = 3'b1_1_0;        int_ack = int_ack;
            {io_cs, io_rd, io_wr}              = 3'b0_0_0;           fl_ld   = 1'b0;
            {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b1_00_00_00_00; HILO_ld_SIMD = 1'b0;
            {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
         state = RETI_3;
      end
   //flags <- D_in, ALU_Out <- RS + 4
   RETI_3:
      begin
         @(negedge clk)
            {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
            {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
            {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_x_0_0011; FS     = 5'h11;
            {dm_cs, dm_rd, dm_wr}              = 3'b0_0_0;        int_ack = int_ack;
            {io_cs, io_rd, io_wr}              = 3'b0_0_0;           fl_ld   = 1'b1;
            {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
            {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
         state = RETI_4;
      end
   //$sp <- ALU_Out, D_in <- dM[ALU_Out]
   RETI_4:
      begin
         @(negedge clk)
            {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
            {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
            {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b1_11_x_0_0000; FS     = 5'h0;
            {dm_cs, dm_rd, dm_wr}              = 3'b1_1_0;        int_ack = int_ack;
            {io_cs, io_rd, io_wr}              = 3'b0_0_0;           fl_ld   = 1'b0;
            {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
            {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
         state = RETI_5;
      end
   //PC <- D_in, RS <- $sp
   RETI_5:
      begin
         @(negedge clk)
            {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'b10_1_0_0;
            {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
            {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_x_0_0011; FS     = 5'h0;
            {dm_cs, dm_rd, dm_wr}              = 3'b0_0_0;        int_ack = int_ack;
            {io_cs, io_rd, io_wr}              = 3'b0_0_0;           fl_ld   = 1'b0;
            {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b1_00_00_00_00; HILO_ld_SIMD = 1'b0;
            {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
         state = RETI_6;
      end
   //ALU_Out <- RS + 4
   RETI_6:
      begin
         @(negedge clk)
            {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
            {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
            {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_xx_x_0_xxxx; FS     = 5'h11;
            {dm_cs, dm_rd, dm_wr}              = 3'b0_0_0;        int_ack = int_ack;
            {io_cs, io_rd, io_wr}              = 3'b0_0_0;           fl_ld   = 1'b0;
            {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}   = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
            {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
```

```
                    state = RETI_7;
                end
            //$sp <- ALU_Out
            RETI_7:
                begin
                    @(negedge clk)
                        {pc_sel, pc_ld, pc_inc, ir_ld}       = 5'bxx_0_0_0;
                        {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
                        {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b1_11_x_0_0000; FS      = 5'h0;
                        {dm_cs, dm_rd, dm_wr}              = 3'b0_0_0;        int_ack = int_ack;
                        {io_cs, io_rd, io_wr}              = 3'b0_0_0;           fl_ld   = 1'b0;
                        {S_Sel, DO_Sel, SH_Sel, mSel, Ysel}  = 9'b0_00_00_00_00; HILO_ld_SIMD = 1'b0;
                        {nsd, nsi, nsc, nsv, nsn, nsz}       = {psd, psi, psc, psv, psn, psz};
                    state = FETCH;
                    #1 $display("Updated flags after return: psd=%b,psi=%b,psc=%b,psv=%b,psn=%b,psz=%b",
                                psd, psi, psc, psv, psn, psz);
                end
        endcase
    end
endmodule
```

Integer_Datapath_1

```
`timescale 100ps / 10ps
/****************************************************************************************************
*
* Author: Joshua Hightower / Alec Selfridge
* Email: joshuahightower2@gmail.com / aeselfridge@gmail.com
* Filename: Integer_Datapath_1.v
* Date: September 16th, 2015
* Version: 1.0
*
* Notes: The Integer Datapath 1 is a top level verilog module interconnecting the 32x32 register
* file, the 32-bit ALU, and the HILO register modules.  The purpose of the datapath is to take in
* signals and perform operations based off of those signals.  We can break down how this is done
* in a few simple steps:
*                1) The register file takes in an S and T address and changes the S and T 32-bit
*                   output ports based on those 5-bit signals.
*                2) From the register file, the S output conects to the ALU S input and the T output
*                    is in a 2 to 1 mux with DT and the output of the mux connects to the T input of
*                    the ALU.
*                3) After the ALU has its inputs, it performs an operation based on the 5-bit FS
*                    input and outputs its final product through Y_lo.
*                4) Once the ALU is done, a 5 to 1 mux takes in the Y_lo, HI, LO, DY, and PC_in
*                    values and selects which one to output and redirect into the D input of the
*                    register file based off the Y_sel 3-bit signal.
*                5) If the D_En signal is high, the value of the D input of the register file writes
*                    to the register pointed at by the D address 5-bit signal.
*
* Date: September 30th, 2015
* Version: 1.1
*
* Notes: The updated Integer Datapath adds two register between the regfile and ALU and two more
* that corralte with the HI LO registers.  The purpose of adding these additional four registers
* was to begin pipelining and make the datapath more efficient. This makes it take more clocks to
* perform operations listed as such:
*                t1: regfile -> rs/rt registers
*                t2: rs/rt registers -> hi, lo, alu, and d_in registers
*                t3: hi, lo, alu, and d_in registers -> regfile/output modules
*
* Date: October 11th, 2015
* Version: 1.2
*
* Notes: With this version, a D_mux was added to provide whether we are writing with the address
* pointed at by the T or D address.  This is used between I-type and R-type instructions.
*
* Date: October 15th, 2015
* Version: 1.3
*
* Notes: The D_mux is expended for 2 more inputs to be used with jump instructions for later use
* in later builds.
*
****************************************************************************************************/
module Integer_Datapath_1(
   input       clk,
   input       reset,
   input       D_En,
   input       T_Sel,
   input       HILO_ld,
   input       HILO_ld_SIMD,
   input       S_Sel,
   input [1:0]  DA_Sel,
   input [1:0]  DO_Sel,
   input [1:0]  SH_Sel,
   input [1:0]  Ysel,
   input [1:0]  mSel,
   input [3:0]  Y_Sel,
   input [4:0]  D_Addr,
   input [4:0]  S_Addr,
   input [4:0]  T_Addr,
   input [4:0]  shamt,
   input [4:0]  FS,
   input [31:0] DT,
   input [31:0] DY,
   input [31:0] IO,
   input [31:0] PC_in,
   input [31:0] flags,
   output wire  C,
   output wire  V,
   output wire  N,
   output wire  Z,
   output wire  DBZ,
   output wire [31:0] ALU_OUT,
   output wire [31:0] D_OUT
   );

    /*32 bit wires interconnecting modules inside the datapath*/
    // S = regfile -> RS reg
    //  T = regfile -> T_Mux
    //   Y_hi = ALU -> mul/div registers
    //    Y_lo = ALU -> mul/div registers, ALU_Output reg
    //     HI = mul/div registers -> ALU_OUT mux
    //      LO = mul/div registers -> ALU_OUT mux
    //      RS = RS reg -> ALU
    //       T_Mux = T_Mux -> ALU
    //        D_in = D_in reg -> ALU_OUT mux
    //         ALU_Output = ALU_Output reg -> ALU_OUT mux
    //          D_Mux = WriteAddr-> regfile
    wire [63:0] SIMD_in, SIMD_HILO;
    wire [31:0] S, T, Y_hi, Y_lo, HI, LO, RS, RT, T_Mux, D_in, ALU_Output,
                IO_in, SH_out, SH_Output, SIMD_out;
    wire  [4:0] D_Mux, SA;

   //clk, reset, D_En, D, D_Addr, S_Addr, T_Addr, S, T
   reg32x32 registerFile(clk, reset, D_En, ALU_OUT[31:0], D_Mux[4:0], SA[4:0], T_Addr[4:0], S[31:0], T[31:0]);

   //clk, reset, S, T, RS, RT
   Two_32bit_regs RS_RT(clk, reset, S[31:0], T_Mux[31:0], RS[31:0], RT[31:0]);

   //S, T, FS, Y_hi, Y_lo, C, V, N, Z
   ALU_32 ALU(RS[31:0], RT[31:0], FS[4:0], Y_hi[31:0], Y_lo[31:0], C, V, N, Z, DBZ);

   Barrel_Shifter32 BS32(.D_in(RT), .shamt(shamt), .Sel(SH_Sel), .D_out(SH_out));

   Partial_Datapath_SIMD SIMD(.mSel(mSel), .Ysel(Ysel), .FS(FS), .RS(RS), .RT(RT), .Y(SIMD_in));

   HILO_registers mul_div_registers(.clk(clk), .reset(reset), .HILO_ld(HILO_ld), .HILO_ld_SIMD(HILO_ld_SIMD),
                                     .Y_hi(Y_hi), .Y_lo(Y_lo), .ALU(Y_lo), .SH(SH_out), .D_in(DY), .IO_in(IO),
                                     .SIMD_in(SIMD_in), .HI(HI), .LO(LO), .ALU_Output(ALU_Output),
                                     .SH_Output(SH_Output), .SIMD_Output(SIMD_out), .SIMD_HILO(SIMD_HILO),
                                     .D_out(D_in), .IO_out(IO_in));

   //assign D_Out: 00 = RT, 01 = PC_in, 10 = flags
   assign D_OUT =   (DO_Sel==2'b00) ?   RT  :
                    (DO_Sel==2'b01) ? PC_in :
                    (DO_Sel==2'b10) ? flags :
                                        RT ;

   //assign S_Addr: 1 = r29($sp), 0 = S_addr
   assign SA = (S_Sel) ? 5'b11101 : S_Addr;
```

```
   //assign statemet for D_Mux: true = D_addr, false = T_Addr
   assign D_Mux =   (DA_Sel==2'b00) ?   D_Addr :
                    (DA_Sel==2'b01) ?   T_Addr :
                    (DA_Sel==2'b10) ? 5'b11111 :  //$ra
                    (DA_Sel==2'b11) ? 5'b11101 :  //$sp
                                          D_Addr ;

   //assign for T-mux : true = DT, false = T
   assign T_Mux = T_Sel ? DT : T;

   assign ALU_OUT = (Y_Sel==4'b0000) ?   ALU_Output :
                (Y_Sel==4'b0001) ?               HI :
                (Y_Sel==4'b0010) ?               LO :
                (Y_Sel==4'b0011) ?             D_in :
                (Y_Sel==4'b0100) ?            PC_in :
                (Y_Sel==4'b0101) ?            IO_in :
                (Y_Sel==4'b0110) ?        SH_Output :
                (Y_Sel==4'b0111) ?         SIMD_out :
                (Y_Sel==4'b1000) ?  SIMD_HILO[63:32] :
                (Y_Sel==4'b1001) ?   SIMD_HILO[31:0] :
                                      ALU_Output ;

endmodule
```

Mem_4096x32

```
`timescale 1ns / 1ps
/*****************************************************************************************************
*
* Author: Joshua Hightower / Alec Selfridge
* Email: joshuahightower2@gmail.com / aeselfridge@gmail.com
* Filename: mem_4096x32.v
* Date: October 10th, 2015
* Version: 1.0
*
* Notes: The mem_4069x32 creates a byte-addressable memory organized in big-endian format.  To
* access memory(read or write), the cs scalar input must be asserted.  To read or write, each
* operation has its own scalar input and must be asserted to perform these.  Writing is
* synchronous while reading is asnchronous. UPDATED TO 4K ON 10/19/15.
*
*****************************************************************************************************
*/
module mem_4096x32(
   input              clk,
   input              cs,
   input              wr,
   input              rd,
   input       [11:0] addr,
   input       [31:0] D_in,
   output wire [31:0] D_out
   );

   // 4096x32 = 16,380 bytes
   reg [7:0]  M[0:16380];

   //postive clock logic where it checks if cs and wr is asserted
   // if asserted, memory is stored at the locations pointed at by addr
   // stores memory in big-endian format
   always@(posedge clk) begin
     if(cs && wr)
       {M[addr],  M[addr+1], M[addr+2], M[addr+3]} <=
       {D_in[31:24], D_in[23:16], D_in[15:8], D_in[7:0]};
   end

   /*
   *  Tri-state output that assigns the value pointed at by Addr
   *  if the read signal is asserted. If the chip select happens
   *  to be high but we're not reading, then the output should be
   *  zero. Lastly, we'll rest on a Hi-Z state.
   */
   assign D_out = (cs && rd)  ? {M[addr],  M[addr+1], M[addr+2], M[addr+3]} :
                  (cs && !rd) ?  32'h0 :
                                 32'hZ ;

endmodule
```

Instruction_Unit

```
`timescale 1ns / 1ps
/*******************************************************************************************************
*
* Author: Joshua Hightower / Alec Selfridge
* Email: joshuahightower2@gmail.com / aeselfridge@gmail.com
* Filename: Instruction_Unit.v
* Date: October 4th, 2015
* Version: 1.0
*
* Notes: The Instructions Unit is a top level verilog module bringing together the PC and IR
* registers with the instruction memory.  The logic with this module is when PC reads memory, the
* instruction is stored into the IR and PC increments by 4.  Even if the sign extension is not
* used in the instruction, we still sign extend the first 16 bits in case it needs to used for an
* I-type instruction.  On reset, PC and IR get zero'ed out, memory is unaffected.
*
* Date: October 12th, 2015
* Version: 1.1
*
* Notes: In this update, a 2 bit wide pc_sel input is added to select between the branch, jump,
* and PC 32 bit value to be stored into the PC regsiter if pc_ld is enabled.  This will not be
* implemented in Lab 6 for any use, but will see use in future builds.
*
*******************************************************************************************************
*/
module Instruction_Unit(
   input              clk,
   input              reset,
   input              pc_ld,
   input              pc_inc,
   input              im_cs,
   input              im_wr,
   input              im_rd,
   input              ir_ld,
   input        [1:0] pc_sel,
   input       [31:0] PC_in,
   output wire [31:0] PC_out,
   output wire [31:0] IR_out,
   output wire [31:0] SE_16
   );

      wire [31:0] D_in, IR, PC;

      assign D_in = 32'b0;

      assign   PC = (pc_sel==2'b00) ?     {PC_out + {SE_16[29:0], 2'b00}}   : //branch address
                    (pc_sel==2'b01) ? {PC_out[31:28], IR_out[25:0], 2'b00}  : //jump address
                    (pc_sel==2'b10) ?                                PC_in : //jump register
                                                                     PC_in  ; //default

      PC_register PC_reg(.clk(clk),   .reset(reset),    .pc_ld(pc_ld), .pc_inc(pc_inc),
                         .PC_in(PC), .PC_out(PC_out));

      mem_4096x32 iMem(.clk(clk),     .cs(im_cs),   .wr(im_wr),  .rd(im_rd),
                       .addr(PC_out), .D_in(D_in), .D_out(IR));
```

```
	IR_register IR_reg(.clk(clk), .reset(reset), .ir_ld(ir_ld), .IR_in(IR),
	                    .IR_out(IR_out));

	assign SE_16 = {{16{IR_out[15]}}, IR_out[15:0]};

endmodule
```

IO_Mem

```
`timescale 100ps / 10ps
/*****************************************************************************************************
*
* Author: Joshua Hightower / Alec Selfridge
* Email: joshuahightower2@gmail.com / aeselfridge@gmail.com
* Filename: dMem_1024x32.v
* Date: September 30th, 2015
* Version: 1.0
*
* Notes: The dMem_1024x32 creates a byte-addressable memory organized in big-endian format.  To
* access memory(read or write), the dm_cs scalar input must be asserted.  To read or write, each
* operation has its own scalar input and must be asserted to perform these.  Writing is
* synchronous while reading is asnchronous.
*
* Date: October 18th, 2015
* Version: 1.1
*
* Notes: In the updated version, the memory is expanded to accomodate 4k byte address space in
* memory.
*
******************************************************************************************************
*/
module IO_Mem(
   input              clk,
   input              io_cs,
   input              io_wr,
   input              io_rd,
   input              int_ack,
   input       [11:0] addr,
   input       [31:0] D_in,
   output wire [31:0] D_out,
   output reg         intr
   );

   // 1024x32 = 4096 bytes
   reg [7:0]  ioMem[0:16380];

   //postive clock logic where it checks if dm_cs and dm_wr is asserted
   // if asserted, memory is stored at the locations pointed at by addr
   //  stores memory in big-endian format
   always@(posedge clk) begin
     if(io_cs && io_wr)
       {ioMem[addr],  ioMem[addr+1], ioMem[addr+2], ioMem[addr+3]} <=
       {D_in[31:24], D_in[23:16], D_in[15:8], D_in[7:0]};
   end

   /*
   *  Tri-state output that assigns the value pointed at by Addr
   *  if the read signal is asserted. If the chip select happens
   *  to be high but we're not reading, then the output should be
   *  zero. Lastly, we'll rest on a Hi-Z state.
   */
   assign D_out = (io_cs && io_rd)  ? {ioMem[addr],  ioMem[addr+1], ioMem[addr+2], ioMem[addr+3]} :
                  (io_cs && !io_rd) ?  32'h0 : 32'hZ;
   //generate an interrupt
   initial begin
      intr = 1'b0;
      //wait 45ns for intr to trigger
      //#450 intr = 1'b1;
      //wait til int_ack goes high before turning off intr
      //@(posedge int_ack)
         //intr = 1'b0;
   end

endmodule
```

Reg32x32

```
`timescale 1ns / 1ps
/*************************************************************************************************
*
* Author: Joshua Hightower / Alec Selfridge
* Email: joshuahightower2@gmail.com / aeselfridge@gmail.com
* Filename: reg32x32.v
* Date: September 10th, 2015
* Version: 1.0
*
* Notes: The reg32x32 module uses behavioral verilog to create a register file with 32 registers
* that are 32 bits wide.  At each rising edge of the clock or reset, it checks if the reset is
* high, in which case register 0 will get written a 32 bit hex value of zero.  This is the only
* register that changes on reset; every other register retains its values during a reset.  If no
* reset is detected, the register pointed at by the D address gets written the 32 bit D input if
* and only if the data enable is high and it is not register 0.  The S and T outputs use a
* conditional assign statement using the registers pointed at by the S and T addresses,
* respectively.
*
*************************************************************************************************
*/
module reg32x32(clk, reset, D_En, D, D_Addr, S_Addr, T_Addr, S, T);
   input              clk, reset, D_En;        //scalar inputs for clock, reset, and data write enable
   input        [4:0] D_Addr, S_Addr, T_Addr; //5 bit inputs for the write/read addresses
   input       [31:0] D;                       //32 bit input for the write data
   output wire  [31:0] S, T;                   //two 32 bit outputs to display contents of registers

   //initialize 32 registers, each 32 bits wide
   reg [31:0]  register[0:31];

   //at the positive edge of the clock or reset, we are setting register[0] to all zero's if reset
   //is high or we are giving the register pointed at by the D_Addr the 32 bit D input iff D_En is
   //high and the D_Addr is not pointing to register[0]
   always@(posedge clk or posedge reset) begin
      if(reset)
         register[0] <= 32'b0;
      else if((D_Addr != 5'b0) && (D_En))
         register[D_Addr] <= D;
   end

   //S port outputs the register pointed at by the S_Addr
   assign S = register[S_Addr];

   //T port outputs the register pointed at by the T_Addr
   assign T = register[T_Addr];

endmodule
```

Two_32bit_regs

```
`timescale 1ns / 1ps
/**************************************************************************************************
*
* Author: Joshua Hightower / Alec Selfridge
* Email: joshuahightower2@gmail.com / aeselfridge@gmail.com
* Filename: Two_32bit_regs.v
* Date: September 30th, 2015
* Version: 1.0
*
* Notes: The Two 32bit Reg module uses positive edge detection of the clock and reset to take in
* two 32 bit values into two seperate 32 bit registers.  If reset goes high, the two registers get
* zero'ed out, else, the two registers get the according values.
*
***************************************************************************************************
*/
module Two_32bit_regs(
   input           clk,
   input           reset,
   input     [31:0] S,
   input     [31:0] T,
   output reg [31:0] RS,
   output reg [31:0] RT
   );

    //always block using clock logic on positive edge of clk or reset
    // if reset, HI and LO get zero'ed out
    //  else HI <- Y_hi and LO <- Y_lo
    always@(posedge clk or posedge reset)
       if(reset)
          {RS, RT} <= 64'h0;
       else
          {RS, RT} <= {S,T};

endmodule
```

ALU_32

```
`timescale 1ns / 1ps
/*******************************************************************************************************
*
* Author: Joshua Hightower / Alec Selfridge
* Email: joshuahightower2@gmail.com / aeselfridge@gmail.com
* Filename: ALU_32.v
* Date: August 26th, 2015
* Version: 1.0
*
* Notes: The ALU_32 module takes in two 32bit inputs used to perform operations
* and a 5bit opcode input to dictate which operation to perform.  The results
* of these operations output either a 32bit or a 64bit output, depending on which
* operation is performed.  In order to conduct these proceedings, I used one
* divide module, one multiply module, and one MIPS module.  The MIPS is used
* for all operations (listed below) besides multiply and divide.  The multiply
* and divide take care of those two operations, respectively.  The output and
* N flag use 3 to 1 mux's to determine their output while Z, V, and C use
* 2 to 1 mux's.  Below, I listed the available operations with their coresponding
* opcode.
*
*            Opcode Guide (written in hex)
*        Arithmetic          Logic                Other
*        PASS_S = 00          AND  = 08          INC     = 0F
*        PASS_T = 01          OR   = 09          DEC     = 10
*        ADD    = 02          XOR  = 0A          INC4    = 11
*        ADDU   = 03          NOR  = 0B          DEC4    = 12
*        SUB    = 04          SLL  = 0C          ZEROS   = 13
*        SUBU   = 05          SRL  = 0D          ONES     = 14
*      SLT    = 06          SRA  = 0E          SP_INIT = 15
*      SLTU   = 07          ANDI = 16
*        MUL    = 1E          ORI  = 17
*        DIV    = 1F          XORI = 18
*                            LUI  = 19
*
********************************************************************************************************
*/
module ALU_32(S, T, FS, Y_hi, Y_lo, C, V, N, Z, DBZ);

   input        [31:0]  S, T;            //32bit inputs
   input         [4:0]  FS;              //5bit input for FS
   output wire [31:0]  Y_hi;            //32bit wire for Y_hi
   output wire [31:0]  Y_lo;            //32bit wire for Y_lo
   output wire         C, V, N, Z, DBZ; //1 bit wires for C, V, N, Z, and DBZ flags

   wire [31:0] Quot, Rem, Y;     //32bit wires used to output the divide and mips unit to a mux
   wire [63:0] Product;          //64bit wire used to output the multiply unit to a mux
   wire        alu_C, alu_V, dz; //1bit wires to output the mips C and V flags to a mux


   //instatiates 3 modules: one DIV_32, one MPY_32, and one MIPS_32
   //          S          T         Quot       Rem
   DIV_32  div(S[31:0], T[31:0], Quot[31:0], Rem[31:0], dz);

   //          S          T           Product
   MPY_32  mpy(S[31:0], T[31:0], Product[63:0]);
```

```
    //              S        T          FS       Y        V      C
    MIPS_32 mips(S[31:0], T[31:0], FS[4:0], Y[31:0], alu_V, alu_C);

    //3 to 1 mux to select the output based on the function select
    assign   {Y_hi,Y_lo} =    (FS==5'h1E) ? Product    :   //select multiply output
                              (FS==5'h1F) ? {Rem, Quot}:   //select divide output
                                            {32'h0, Y };   //select mips output

    //3 to 1 mux to output the N flag based on certain operations
    assign N = (FS==5'h03 | FS==5'h05 | FS==5'h07) ? 1'bx :    //N = x if unsigned
               (FS==5'h1E)                         ? Y_hi[31]: //N = 64th bit of product
                                                     Y_lo[31]; //N is set to the 32nd bit of output

    //V/C is set to don't care if we are multipling or dividing, else we get V/C from the ALU
    assign {V,C} = (FS==5'h1E | FS==5'h1F) ? 2'bxx : {alu_V, alu_C};

    //Z is set if all 64 bits are zero, else it is not set
    assign Z = ((Y_hi==32'b0) & (Y_lo==32'b0)) ? 1'b1 : 1'b0;

    //assign DBZ flag to the divide ouput if dividing, else don't care
    assign DBZ = (FS==5'h1F) ? dz : 1'bx;

endmodule
```

Barrel_Shifter32

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////
//   Author: Alec Selfridge
//    Email: aeselfridge@gmail.com
// Creation: 10/24/2015
// Last Rev: 10/29/2015
//  Version: 1.1
//
//    Notes: This unit can shift the D_in input up to 31 times
//           specified by shamt. Sel chooses the type of shift.
//      1.1: Added rotate right by shamt.
//
////////////////////////////////////////////////////////////////////
module Barrel_Shifter32(
    input      [31:0] D_in,
    input      [4:0]  shamt,
    input      [1:0]  Sel,
    output reg [31:0] D_out
    );

    reg [31:0] i, tmp; // used in RR

    parameter
      SLL = 2'b00, // left-logical
      SRL = 2'b01, // right-logical
      SRA = 2'b10, // right-arithmetic
      RR  = 2'b11; // rotate right

    always @(*) begin
      case(Sel)
        SLL:
          D_out = D_in << shamt;
        SRL:
          D_out = D_in >> shamt;
        SRA: begin
          D_out     = D_in >> shamt;
          D_out[31] = D_in[31];
          end
        RR: begin
          tmp = 0;
          for(i = 0; i < shamt; i = i + 1)
            tmp[31-(shamt-i)+1] = D_in[i];
          D_out = D_in >> shamt;
          D_out = D_out | tmp;
          end
      endcase
    end

endmodule
```

Partial_Datapath_SIMD

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////
//   Author: Alec Selfridge
//    Email: aeselfridge@gmail.com
// Creation: 11/04/2015
// Last Rev: 11/04/2015
//  Version: 1.0
//
//    Notes: A partial datapath used as a secondary one or for SIMD
//          instructions. It has no register file of its' own, so
//          RS & RT are also used here. Thus, the result is
//          written to the same register file.
//
//     mSel: 00: STND   Ysel: 00: ALU
//          01: SIMD8       01: MPY
//          10: SIMD16      10: DIV
//
////////////////////////////////////////////////////////////////////
module Partial_Datapath_SIMD(mSel, Ysel, FS, RS, RT, Y);

   input [31:0] RS,   RT;
   input [4:0]  FS;
   input [1:0]  mSel, Ysel;

   output wire [63:0] Y;

   wire [31:0] alu;
   wire [63:0] mpy, div;

   ALU32_DM ALU(.FS(FS), .mSel(mSel), .S(RS), .T(RT), .Y(alu), .C(), .V());

   MPY32_DM MPY(         .mSel(mSel), .S(RS), .T(RT), .Y(mpy));

   DIV32_DM DIV(         .mSel(mSel), .S(RS), .T(RT), .Y(div));

   assign Y = (Ysel == 2'b00) ? {32'h0, alu}:
              (Ysel == 2'b01) ? mpy:
              (Ysel == 2'b10) ? div:
                       {32'h0, alu};

endmodule
```

HILO_registers

```
`timescale 1ns / 1ps
/*******************************************************************************************************
*
* Author: Joshua Hightower / Alec Selfridge
* Email: joshuahightower2@gmail.com / aeselfridge@gmail.com
* Filename: HILO_registers.v
* Date: September 16th, 2015
* Version: 1.0
*
* Notes: The HILO_registers module creates two 32-bit registers to hold the multiply and divide
* product / quoietent and remainder.  Using synchronous logic off the positive edge of clk or
* reset, if reset goes high, both registers are zero'ed out. Else, if the HILO_ld is high, HI and
* LO receive the input of their respective registers.
*
* Version 1.1
*
* Notes: The updated version of the HILO registers module adds an additional two 32 bit registers
* for both inputs and outputs.  The logic stays the same, except when the HILO_ld signal is low
* and reset is low, only the two new registers are written to.
*
********************************************************************************************************
*/
module HILO_registers(
   input             clk,
   input             reset,
   input             HILO_ld,
   input             HILO_ld_SIMD,
   input      [31:0] Y_hi,
   input      [31:0] Y_lo,
   input      [31:0] ALU,
   input      [31:0] SH,
   input      [31:0] D_in,
   input      [31:0] IO_in,
   input      [63:0] SIMD_in,
   output reg [31:0] HI,
   output reg [31:0] LO,
   output reg [31:0] ALU_Output,
   output reg [31:0] SH_Output,
   output reg [31:0] SIMD_Output,
   output reg [63:0] SIMD_HILO,
   output reg [31:0] D_out,
   output reg [31:0] IO_out
   );

     //always block using clock logic on positive edge of clk or reset
     // if reset, ALU_Output, D_out, HI, and LO get zero'ed out
     //  else if HILO_ld, ALU_Output <- ALU, D_out <- D_in, HI <- Y_hi, and LO <- Y_lo
     //   else, ALU_Output <- ALU and D_out <- D_in
     always@(posedge clk or posedge reset)
        if(reset) begin
           {HI,LO,ALU_Output,SH_Output}        <= 128'h0;
         {SIMD_HILO,SIMD_Output,D_out,IO_out} <= 160'h0;
      end
        else if(HILO_ld) begin
           {HI,LO,ALU_Output,SH_Output}        <= {Y_hi,Y_lo,ALU,SH};
```

```
          {SIMD_HILO,SIMD_Output,D_out,IO_out} <= {SIMD_HILO,SIMD_in,D_in,IO_in};
      end
    else if(HILO_ld_SIMD) begin
         {HI,LO,ALU_Output,SH_Output}        <= {HI,LO,ALU,SH};
       {SIMD_HILO,SIMD_Output,D_out,IO_out} <= {SIMD_in,SIMD_Output,D_in,IO_in};
      end
        else begin
            {HI,LO,ALU_Output,SH_Output}        <= {HI,LO,ALU,SH};
            {SIMD_Output,D_out,IO_out}          <= {SIMD_in,D_in,IO_in};
                                 SIMD_HILO        <= SIMD_HILO;
      end

endmodule
```

MIPS_32

```
`timescale 1ns / 1ps
/****************************************************************************************************
*
* Author: Joshua Hightower / Alec Selfridge
* Email: joshuahightower2@gmail.com / aeselfridge@gmail.com
* Filename: MIPS_32.v
* Date: August 26th, 2015
* Version: 1.0
*
* Notes: The MIPS_32 module takes in two 32bit inputs and outputs a 32bit
* output.  The output is based upon an operation done with the S and T inputs,
* which the operation is based upon the 5bit FS(function select) input. Every
* operation updates the Y output and the V(overflow) and C(carry) flags. The
* operations that "don't care" about the flags will put a x for them to
* output.  Below is a guide that tells you which opcode does each operation.
*
****************************************************************************************************
*/
module MIPS_32(S, T, FS, Y, V, C);

    input      [31:0]   S, T;    //32bit inputs
    input       [4:0]    FS;       //5bit opcode
    output reg [31:0]   Y;       //32bit output
    output reg         V, C;    //1bit outputs for carry and overflow

    integer int_s, int_t, i;       //integers used for SLT operation

    always@( * ) begin
        case (FS)
            5'h00:   {V,C,Y} = {2'bxx, S};    //output S
            5'h01:   {V,C,Y} = {2'bxx, T};    //output T
            5'h02:   begin
                         {C,Y} = S + T;                      //add S by T, signed
                         V = (S[31]  & T[31]  & ~Y[31]) |    //if two pos numbers produces neg value,V
                             (~S[31] & ~T[31] &  Y[31]);    //if two neg numbers produces pos value,V
                      end
            5'h03:   begin
                         {C,Y} = S + T;                     //add S by T, unsigned
                         V = C;                             //V is set to C bit
                      end
            5'h04:   begin
                         {C,Y} = S - T;                     //minus S by T, signed
                         V = (~S[31] & T[31] & Y[31]) |    //if - minus + number produces + value,V
                             (S[31]  & ~T[31] & ~Y[31]);  //if + minus - number produces - value,V
                      end
            5'h05:   begin
                         {C,Y} = S - T;                     //minus S by T, unsigned
                         V = C;                            //V is set to C bit
                      end
            5'h06:   begin
                         int_s = S;
                         int_t = T;
                         if(int_s < int_t)                 //if S is less than T, set Y to 1
                            {V,C,Y} = {2'bxx, 32'h0000_0001};
                         else                               //else, set Y to 0
```

```
                    {V,C,Y} = {2'bxx, 32'h0};
                end
        5'h07:  begin
                    if(S < T)                               //if S is less than T, set Y to 1
                        {V,C,Y} = {2'bxx, 32'h0000_0001};
                    else                                     //else, set Y to 0
                        {V,C,Y} = {2'bxx, 32'h0};
                end
        5'h08:  {V,C,Y} = {2'bxx, S & T};       //S and'ed with T
        5'h09:  {V,C,Y} = {2'bxx, S | T};       //S or'ed with T
        5'h0A:  {V,C,Y} = {2'bxx, S ^ T};       //S xor'ed with T
        5'h0B:  {V,C,Y} = {2'bxx, ~(S | T)};    //S nor'ed with T
        5'h0C:  begin
                    V  = 1'bx;                  //V is a don't care
                    C  = T[31];                 //C is = to 31st bit
                    Y    = T << 1;              //shifts bits to left by 1
                end
        5'h0D:  begin
                    V  = 1'bx;                   //V is a don't care
                    C  = T[0];                   //C is = to 1st bit
                    Y  = T >> 1;                //shift bits to right by 1
                end
        5'h0E:  begin
                    V     = 1'bx;                //V is a don't care
                    C    = T[0];                 //C is = to 1st bit
                    Y      = T >> 1;             //shift bits right by 1, signed
                    Y[31] = T[31];               //set sign bit to original sign bit
                end
        5'h0F:  begin
                    {C,Y} = {S + 1};             //increment S by 1
                    V    = (~S[31] & Y[31]);     //V is set if a pos incs into a neg
                end
        5'h10:  begin
                    {C,Y} = {S - 1};             //decrement S by 1
                    V    = (S[31] & ~Y[31]);     //V is set if a neg decs into a pos
                end
        5'h11:  begin
                    {C,Y} = {S + 4};             //increment S by 4
                    V    = (~S[31] & Y[31]);     //V is set if a pos incs into a neg
                end
        5'h12:  begin
                    {C,Y} = {S - 4};             //decrement S by 4
                    V    = (S[31] & ~Y[31]);     //V is set if a neg decs into a pos
                end
        5'h13:  {V,C,Y} = {2'bxx, 32'b0};                   //outputs all zeros
        5'h14:  {V,C,Y} = {2'bxx, 32'hFFFF_FFFF};          //outputs all ones
        5'h15:  {V,C,Y} = {2'bxx, 32'h3FC};                //outputs 3FC
        5'h16:  {V,C,Y} = {2'bxx, S & {16'h0,T[15:0]}};  //S and'ed with (0000,T[15:0])
        5'h17:  {V,C,Y} = {2'bxx, S | {16'h0,T[15:0]}};  //S or'ed with (0000,T[15:0])
        5'h18:  {V,C,Y} = {2'bxx, S ^ {16'h0,T[15:0]}};  //S xor'ed with (0000,T[15:0])
        5'h19:  {V,C,Y} = {2'bxx, {T[15:0],16'h0}};           //output (T[15:0],0000
        5'h1A: begin                              // bit set: S[imm]
                {V,C} = 2'bxx;
                Y    = S;
                // clear sign ext
                Y[(T << 26) >> 26]  = 1'b1;
              end
        5'h1B: begin                              // bit clear: S[imm]
              {V,C} = 2'bxx;
              Y    = S;
```

```
                    // clear sign ext
                    Y[(T << 26) >> 26]  = 0;
                    end
            5'h1C: begin                                  // reverse bits in S
                     for(i = 0; i < 32; i = i + 1)
                        Y[i] = S[(31 - i)];
                    {V,C} = 2'bxx;
                    end
            5'h1D: begin                                  // reverse endianness
                     Y[7:0]   = S[31:24];
                     Y[15:8]  = S[23:16];
                     Y[23:16] = S[15:8];
                     Y[31:24] = S[7:0];
                     {V,C} = 2'bxx;
                    end
              default:   {V,C,Y} = {2'bxx, S};        //output S as default
       endcase   //end case statement

    end  // end always

endmodule
```

DIV_32

```
`timescale 1ns / 1ps
/****************************************************************************************************
*
* Author: Joshua Hightower / Alec Selfridge
* Email: joshuahightower2@gmail.com / aeselfridge@gmail.com
* Filename: DIV_32.v
* Date: August 26th, 2015
* Version: 1.0
*
* Notes: The DIV_32 module takes in two 32bit inputs and divides them to get
* a 32bit quotient and a 32bit remainder.  In order to do this, for efficiency
* sake, I used two integers to store the 32bit inputs and used them for the
* divide and modulus operations to get a signed 32bit quotient and remainder.
*
****************************************************************************************************
*/
module DIV_32(S, T, Quot, Rem, DBZ);

   input       [31:0]   S, T;      //32bit inputs
   output reg            DBZ;      //divide by zero flag
   output reg [31:0]   Quot, Rem; //32bit quotient and remainder

   integer int_s, int_t;           //instatiate two integer variables

   always@( S or T ) begin

      int_s = S;                   //cast S input as a signed 32bit integer
      int_t = T;                   //cast T input as a signed 32bit integer

      if(T == 32'b0) begin
         DBZ = 1'b1;
      end
      else begin
         DBZ  = 1'b0;
         Quot = int_s / int_t;      //divide S by T to get quotient
         Rem  = int_s % int_t;      //modulus S by T to get remainder
      end

   end   //end always

endmodule
```

MPY_32

```
`timescale 1ns / 1ps
/****************************************************************************************************
*
* Author: Joshua Hightower / Alec Selfridge
* Email: joshuahightower2@gmail.com / aeselfridge@gmail.com
* Filename: MPY_32.v
* Date: August 26th, 2015
* Version: 1.0
*
* Notes: The MPY_32 module takes in two 32bit inputs and outputs a 64bit
* product.  In order to do this in the most efficient way, I store the 32bit
* inputs into signed 32bit integer variables.  After which, I store the
* product of S multiplied by T into a 64bit register output.
*
****************************************************************************************************
*/
module MPY_32(S, T, Product);

   input       [31:0]   S, T;       //32bit inputs
   output reg [63:0]     Product;  //64bit output

   integer int_s, int_t;           //instatiate two interger variables

   always@( S or T ) begin

      int_s = S;                   //cast S input as a signed 32bit integer
      int_t = T;                   //cast T input as a signed 32bit integer
      Product = int_s*int_t;       //store the 64bit product of S * T

   end   //end always

endmodule
```

ALU32_DM

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////
//   Author: Alec Selfridge
//    Email: aeselfridge@gmail.com
// Creation: 10/30/2015
// Last Rev: 10/30/2015
//  Version: 1.0
//
//    Notes: Dual-mode ALU capable of 32-bit SIMD or standalone
//          modes. In SIMD mode, a sub-mode (Quad8/Dual16)
//          identifies the data type. The other mode puts the ALU
//          into, effectively, parallel mode. In standalone mode,
//          the data type is 32-bit integer and functions like a
//          standard ALU. FS is used to select one of many
//          operations in either case.
//
//////////////////////////////////////////////////////////////////
module ALU32_DM(FS, mSel, S, T, Y, C, V);

    input  [4:0]     FS;
    input  [1:0]   mSel;
    input  [31:0]  S, T;
    output reg [31:0] Y;
    output reg    C, V;

    parameter
      // modes
      mSTND   = 2'b00, mSIMD8  = 2'b01, mSIMD16 = 2'b10,
      // SIMD8
      ADD8    = 5'h02,
      SUB8    = 5'h03,
      AND8    = 5'h04,
      OR8     = 5'h05,
      NOT8    = 5'h06,
      XOR8    = 5'h07,
      // SIMD16
      ADD16   = 5'h02,
      SUB16   = 5'h03,
      AND16   = 5'h04,
      OR16    = 5'h05,
      NOT16   = 5'h06,
      XOR16   = 5'h07,
      // Standalone
      PASS_S  = 5'h00,
      PASS_T  = 5'h01,
      ADD     = 5'h02,
      ADDU    = 5'h03,
      SUB     = 5'h04,
      SUBU    = 5'h05,
      AND     = 5'h06,
      OR     = 5'h07,
      NOT     = 5'h08,
      XOR     = 5'h09;

      always@(*) begin
      case(mSel)
        mSIMD8: begin
          case(FS)
            PASS_S: begin
              {C, V} = 2'bxx;
              Y = S;
            end
            PASS_T: begin
              {C, V} = 2'bxx;
              Y = T;
            end
            ADD8: begin
              {C, V} = 2'bxx;
              Y[7:0]   = S[7:0]   + T[7:0];
              Y[15:8]  = S[15:8]  + T[15:8];
              Y[23:16] = S[23:16] + T[23:16];
              Y[31:24] = S[31:24] + T[31:24];
            end
            SUB8: begin
              {C, V} = 2'bxx;
              Y[7:0]   = S[7:0]   - T[7:0];
              Y[15:8]  = S[15:8]  - T[15:8];
              Y[23:16] = S[23:16] - T[23:16];
              Y[31:24] = S[31:24] - T[31:24];
            end
            AND8: begin
              {C, V} = 2'bxx;
              Y[7:0]   = S[7:0]   & T[7:0];
              Y[15:8]  = S[15:8]  & T[15:8];
              Y[23:16] = S[23:16] & T[23:16];
              Y[31:24] = S[31:24] & T[31:24];
            end
            OR8: begin
              {C, V} = 2'bxx;
              Y[7:0]   = S[7:0]   | T[7:0];
              Y[15:8]  = S[15:8]  | T[15:8];
              Y[23:16] = S[23:16] | T[23:16];
              Y[31:24] = S[31:24] | T[31:24];
            end
            NOT8: begin
              {C, V} = 2'bxx;
              Y[7:0]   = ~S[7:0];
              Y[15:8]  = ~S[15:8];
              Y[23:16] = ~S[23:16];
              Y[31:24] = ~S[31:24];
            end
            XOR8: begin
              {C, V} = 2'bxx;
              Y[7:0]   = S[7:0]   ^ T[7:0];
              Y[15:8]  = S[15:8]  ^ T[15:8];
              Y[23:16] = S[23:16] ^ T[23:16];
              Y[31:24] = S[31:24] ^ T[31:24];
            end
            default:
              // pass S
              {C, V, Y} = {2'bxx, S};
          endcase
        end
        mSIMD16: begin
          case(FS)
            PASS_S: begin
              {C, V, Y} = {2'bxx, S};
            end
            PASS_T: begin
              {C, V, Y} = {2'bxx, T};
            end
            ADD16: begin
              {C, V} = 2'bxx;
              Y[15:0]  = S[15:0]  + T[15:0];
              Y[31:16] = S[31:16] + T[31:16];
            end
            SUB16: begin
              {C, V} = 2'bxx;
              Y[15:0]  = S[15:0]  - T[15:0];
              Y[31:16] = S[31:16] - T[31:16];
            end
            AND16: begin
              {C, V} = 2'bxx;
              Y[15:0]  = S[15:0]  & T[15:0];
              Y[31:16] = S[31:16] & T[31:16];
            end
            OR16: begin
              {C, V} = 2'bxx;
              Y[15:0]  = S[15:0]  | T[15:0];
              Y[31:16] = S[31:16] | T[31:16];
            end
            NOT16: begin
              {C, V} = 2'bxx;
              Y[15:0]  = ~S[15:0];
              Y[31:16] = ~S[31:16];
            end
            XOR16: begin
              {C, V} = 2'bxx;
              Y[15:0]  = S[15:0]  ^ T[15:0];
              Y[31:16] = S[31:16] ^ T[31:16];
            end
            default:
              // pass S
              {C, V, Y} = {2'bxx, S};
          endcase
        end
        mSTND: begin
          case(FS)
            PASS_S:
              {C, V, Y} = {2'bxx, S};
            PASS_T:
              {C, V, Y} = {2'bxx, T};
            ADD: begin
              {C, Y} = S + T;
              V      = (S[31]  & T[31]  & ~Y[31]) |
                       (~S[31] & ~T[31] &  Y[31]);
            end
            ADDU: begin
              {C, Y} = S + T;
               V     = C;
            end
            SUB: begin
              {C, Y} = S - T;
               V     = (~S[31] & T[31] & Y[31]) |
                       (S[31]  & ~T[31] & ~Y[31]);
            end
            SUBU: begin
              {C, Y} = S - T;
              V      = C;
            end
            AND:
              {C, V, Y} = {2'bxx, S & T};
            OR:
              {C, V, Y} = {2'bxx, S | T};
            NOT:
              {C, V, Y} = {2'bxx, ~S};
            XOR:
              {C, V, Y} = {2'bxx, S ^ T};
            default:
              // pass S
              {C, V, Y} = {2'bxx, S};
          endcase
        end
      endcase
    end

  endmodule
```

DIV32_DM

```verilog
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////
//   Author: Alec Selfridge
//    Email: aeselfridge@gmail.com
// Creation: 11/07/2015
// Last Rev: 11/07/2015
//  Version: 1.0
//
//    Notes: A dual-mode divider that will do 32/32, quad 8/8,
//           or dual 16/16.
//
//                 SIMD8 - 16-bit quotient/rem
//         [   op3  |   op2  |   op1  |   op0  ]
//                 v                v
//         [      REM       |      QUOT      ]
//
//                SIMD16 - 32-bit quotient/rem
//         [                |                ]
//                          v
//         [            REM/QUOT             ]
//
////////////////////////////////////////////////////////////////////
module DIV32_DM(S, T, Y, mSel);

  input      [31:0] S, T;
  input      [1:0]  mSel;
  output reg [63:0] Y;

  integer s, t;

  parameter
    mSTND   = 2'b00,
    mSIMD8  = 2'b01,
    mSIMD16 = 2'b10;

    always@(*) begin
    s = S;
    t = T;
    case(mSel)
      mSTND: begin
        Y[63:32] = s % t;
        Y[31:0]  = s / t;
      end
      mSIMD8: begin
        Y[7:0]   = s[7:0]   / t[7:0];
        Y[15:8]  = s[7:0]   % t[7:0];
        Y[23:16] = s[15:8]  / t[15:8];
        Y[31:24] = s[15:8]  % t[15:8];
        Y[39:32] = s[23:16] / t[23:16];
        Y[47:40] = s[23:16] % t[23:16];
        Y[55:48] = s[31:24] / t[31:24];
        Y[63:56] = s[31:24] % t[31:24];
      end
      mSIMD16: begin
        Y[15:0]  = s[15:0]  / t[15:0];
        Y[31:16]  = s[15:0]  % t[15:0];
        Y[47:32]  = s[31:16] / t[31:16];
        Y[63:48]  = s[31:16] % t[31:16];
      end
    endcase
    end

endmodule
```

MPY32_DM

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////
//   Author: Alec Selfridge
//    Email: aeselfridge@gmail.com
// Creation: 11/04/2015
// Last Rev: 11/04/2015
//  Version: 1.0
//
//    Notes: A dual-mode multiplier that will do 32x32, quad 8x8,
//          or dual 16x16.
//
//                 SIMD8 - 16-bit product
//        [   op3  |   op2  |   op1  |   op0  ]
//                 v                v
//        [       HI      |       LO      ]
//
//                SIMD16 - 32-bit product
//        [               |               ]
//                        v
//        [             HI/LO             ]
//
////////////////////////////////////////////////////////////////////
module MPY32_DM(S, T, Y, mSel);

  input      [31:0] S, T;
  input      [1:0]  mSel;
  output reg [63:0] Y;

  integer s, t;

  parameter
    mSTND   = 2'b00,
    mSIMD8  = 2'b01,
    mSIMD16 = 2'b10;

    always@(*) begin
    s = S;
    t = T;
    case(mSel)
      mSTND:
        Y = s * t;
      mSIMD8: begin
        Y[15:0]  = s[7:0]   * t[7:0];
        Y[31:16] = s[15:8]  * t[15:8];
        Y[47:32] = s[23:16] * t[23:16];
        Y[63:48] = s[31:24] * t[31:24];
      end
      mSIMD16: begin
        Y[31:0]  = s[15:0]  * t[15:0];
        Y[63:32] = s[31:16] * t[31:16];
      end
    endcase
    end

endmodule
```

PC_register

```
`timescale 1ns / 1ps
/****************************************************************************************************
*
* Author: Joshua Hightower / Alec Selfridge
* Email: joshuahightower2@gmail.com / aeselfridge@gmail.com
* Filename: PC_register.v
* Date: October 4th, 2015
* Version: 1.0
*
* Notes: The PC register verilog module creates a 32 bit register that is loaded if the scalar
* input(pc_ld) is high and increments by 4 if the control signal for that is high(pc_inc).  On
* reset, the register is set back to zero.
*
****************************************************************************************************
*/
module PC_register(
   input            clk,
   input            reset,
   input            pc_ld,
   input            pc_inc,
   input     [31:0] PC_in,
   output reg [31:0] PC_out
   );

    always@(posedge clk or posedge reset)
       if(reset)
          PC_out <= 32'b0;
       else if(pc_ld)
          PC_out <= PC_in;
       else if(pc_inc)
          PC_out <= PC_out + 4;

endmodule
```

IR_register

```
`timescale 1ns / 1ps
/****************************************************************************************************
*
* Author: Joshua Hightower / Alec Selfridge
* Email: joshuahightower2@gmail.com / aeselfridge@gmail.com
* Filename: IR_register.v
* Date: October 4th, 2015
* Version: 1.0
*
* Notes: The IR register creates a 32 bit register that is loaded with the input value if the load
* signal is set high.  Else if reset goes high, zero out the register.
*
****************************************************************************************************
*/
module IR_register(
   input             clk,
   input             reset,
   input             ir_ld,
   input      [31:0] IR_in,
   output reg [31:0] IR_out
   );

   always@(posedge clk or posedge reset)
      if(reset)
         IR_out <= 32'b0;
      else if(ir_ld)
         IR_out <= IR_in;

endmodule
```

B. Annotated Memory Modules

III. Verilog - Implementation

Memory Module 1 Instruction Memory

```
@0
3c 01 12 34     R1 <- 0x12345678
34 21 56 78
3c 02 87 65     R2 <- 0x87654321
34 42 43 21
00 01 18 20     R3 <- R1

10 22 00 01     BEQ comparing R1 w/ R2. Should NOT branch to no_eq.
10 23 00 03     BEQ comparing R1 w/ R3. Should branch to yes_eq.
no_eq:
3c 0e ff ff     R14 <- 0xFFFF_FFFF, fail flag
35 ce ff ff
00 00 00 0d     Break

yes_eq:         R14 <- 0x0000_0000, pass
00 00 70 20

14 23 00 01     BNE comparing R1 w/ R3. Should NOT branch to no_ne.
14 22 00 03     BNE comparing R1 w/ R2. Should branch to yes_ne.

no_ne:
3c 0f ff ff     R15 <- 0xFFFF_FFFF, fail flag
35 ef ff ff
00 00 00 0d     Break

yes_ne:
00 00 78 20     R15 <- 0x0000_0000, pass flag
3c 0d 10 01     R13 <- 0x1001_00C0, dMem pointer
35 ad 00 c0
ad a1 00 00     dM[R13] <- R1
00 00 00 0d
                Break
```

III. Verilog - Implementation

Memory Module 1 Results

```
BREAK INSTRUCTION FETCHED   70.50 ns
D I S P L A Y I N G    R E G I S T E R S
time =   72.10 ns   ||  Register 0x0 = 0x00000000
time =   72.20 ns   ||  Register 0x1 = 0x12345678
time =   72.30 ns   ||  Register 0x2 = 0x87654321
time =   72.40 ns   ||  Register 0x3 = 0x12345678
time =   72.50 ns   ||  Register 0x4 = 0xxxxxxxxx
time =   72.60 ns   ||  Register 0x5 = 0xxxxxxxxx
time =   72.70 ns   ||  Register 0x6 = 0xxxxxxxxx
time =   72.80 ns   ||  Register 0x7 = 0xxxxxxxxx
time =   72.90 ns   ||  Register 0x8 = 0xxxxxxxxx
time =   73.00 ns   ||  Register 0x9 = 0xxxxxxxxx
time =   73.10 ns   ||  Register 0xa = 0xxxxxxxxx
time =   73.20 ns   ||  Register 0xb = 0xxxxxxxxx
time =   73.30 ns   ||  Register 0xc = 0xxxxxxxxx
time =   73.40 ns   ||  Register 0xd = 0x100100c0
time =   73.50 ns   ||  Register 0xe = 0x00000000
time =   73.60 ns   ||  Register 0xf = 0x00000000

time =   73.70 ns   ||  dM[0c0] = 0x12345678
time =   73.80 ns   ||  dM[0c4] = 0xxxxxxxxx
time =   73.90 ns   ||  dM[0c8] = 0xxxxxxxxx
time =   74.00 ns   ||  dM[0cc] = 0xxxxxxxxx
time =   74.10 ns   ||  dM[0d0] = 0xxxxxxxxx
time =   74.20 ns   ||  dM[0d4] = 0xxxxxxxxx
time =   74.30 ns   ||  dM[0d8] = 0xxxxxxxxx
time =   74.40 ns   ||  dM[0dc] = 0xxxxxxxxx
time =   74.50 ns   ||  dM[0e0] = 0xxxxxxxxx
time =   74.60 ns   ||  dM[0e4] = 0xxxxxxxxx
time =   74.70 ns   ||  dM[0e8] = 0xxxxxxxxx
time =   74.80 ns   ||  dM[0ec] = 0xxxxxxxxx
time =   74.90 ns   ||  dM[0f0] = 0xxxxxxxxx
time =   75.00 ns   ||  dM[0f4] = 0xxxxxxxxx
time =   75.10 ns   ||  dM[0f8] = 0xxxxxxxxx
time =   75.20 ns   ||  dM[0fc] = 0xxxxxxxxx
```

R1-R3 loaded with specified values.

dMem pointer, location 0x0c0

Pass flags for BNE and BEQ.

Store R1 specified by R13.

III. Verilog - Implementation

Memory Module 2 Instruction Memory

```
@0
3c 01 ff ff
34 21 ff ff
20 02 00 10
3c 0f 10 01
35 ef 00 c0

Top:
00 01 08 42
ad e1 00 00
21 ef 00 04
20 42 ff ff
14 40 ff fb
08 10 00 0c
00 00 00 0d

Exit:
3c 0e 5a 5a
35 ce 3c 3c

00 00 00 0d
```

Load R1 with 0xFFFFFFFF

Load R2 with 0x10

Load R15 with 0x100100C0

Step 1: R1 shifted right by 1 and stored back
Step 2: Store R1 into dM pointed @ by R15
Step 3: Increment R15 by 4
Step 4: Subtract 1 from loop counter (R2)
Step 5: Check if R2 is zero (loop finished)
Go to "Exit" if finished. Else, go to "Top"
Step 6: Break if jump failed

Load R14 with 0x5A5A3C3C

Break

III. Verilog - Implementation

Memory Module 2 Results

```
BREAK INSTRUCTION FETCHED  370.00 ns
D I S P L A Y I N G   R E G I S T E R S
time =  372.00 ns   ||  Register 0x00 = 0x00000000
time =  373.00 ns   ||  Register 0x01 = 0x00000000
time =  374.00 ns   ||  Register 0x02 = 0x00000000
time =  375.00 ns   ||  Register 0x03 = 0xxxxxxxxx
time =  376.00 ns   ||  Register 0x04 = 0xxxxxxxxx
time =  377.00 ns   ||  Register 0x05 = 0xxxxxxxxx
time =  378.00 ns   ||  Register 0x06 = 0xxxxxxxxx
time =  379.00 ns   ||  Register 0x07 = 0xxxxxxxxx
time =  380.00 ns   ||  Register 0x08 = 0xxxxxxxxx
time =  381.00 ns   ||  Register 0x09 = 0xxxxxxxxx
time =  382.00 ns   ||  Register 0x0a = 0xxxxxxxxx
time =  383.00 ns   ||  Register 0x0b = 0xxxxxxxxx
time =  384.00 ns   ||  Register 0x0c = 0xxxxxxxxx
time =  385.00 ns   ||  Register 0x0d = 0xxxxxxxxx
time =  386.00 ns   ||  Register 0x0e = 05a5a3c3c
time =  387.00 ns   ||  Register 0x0f = 0x10010100
```

-R1 ends with 0 after being shifted multiple times
-R2 ends with 0 after being decremented in the loop
-R14 is loaded with A5A53C3C
-R15 contains the pointer to the last memory location referenced+4

```
time =  412.00 ns   ||  dM[0fc] = 0x0000ffff
time =  412.00 ns   ||  dM[0f8] = 0x0001ffff
time =  412.00 ns   ||  dM[0f4] = 0x0003ffff
time =  412.00 ns   ||  dM[0f0] = 0x0007ffff
time =  412.00 ns   ||  dM[0ec] = 0x000ffff
time =  412.00 ns   ||  dM[0e8] = 0x001fffff
time =  412.00 ns   ||  dM[0e4] = 0x003fffff
time =  412.00 ns   ||  dM[0e0] = 0x007fffff
time =  412.00 ns   ||  dM[0dc] = 0x00ffffff
time =  412.00 ns   ||  dM[0d8] = 0x01ffffff
time =  412.00 ns   ||  dM[0d4] = 0x03ffffff
time =  412.00 ns   ||  dM[0d0] = 0x07ffffff
time =  412.00 ns   ||  dM[0cc] = 0x0fffffff
time =  412.00 ns   ||  dM[0c8] = 0x1fffffff
time =  412.00 ns   ||  dM[0c4] = 0x3fffffff
time =  412.00 ns   ||  dM[0c0] = 0x7fffffff
```

Each location, starting with 0x0C0, holds the result of each shift.
For example, 0x0C0 contains the first shift, 0x0C4 contains the second shift, and so on.

III. Verilog - Implementation

Memory Module 3 Instruction Memory

```
@0
3c 01 ff ff
34 21 ff ff
20 02 00 10
3c 0f 10 01
35 ef 00 c0

Top:
00 01 08 43
ad e1 00 00
21 ef 00 04
20 42 ff ff
14 40 ff fb
08 10 00 0c
00 00 00 0d

Exit:
3c 0e 5a 5a
35 ce 3c 3c

00 00 00 0d
```

Load R1 with 0xFFFFFFFF

Load R2 with 0x10

Load R15 with 0x100100C0

Step 1: R1 shifted right (arithmetic) by 1 and stored back
Step 2: Store R1 into dM pointed @ by R15
Step 3: Increment R15 by 4
Step 4: Subtract 1 from loop counter (R2)
Step 5: Check if R2 is zero (loop finished)
Go to “Exit” if finished. Else, go to “Top”

Load R14 with 0x5A5A3C3C

Break

III. Verilog - Implementation

Memory Module 3 Results

```
BREAK INSTRUCTION FETCHED  370.00 ns
D I S P L A Y I N G   R E G I S T E R S
time =  372.00 ns   ||  Register 0x00 = 0x00000000
time =  373.00 ns   ||  Register 0x01 = 0x00000000
time =  374.00 ns   ||  Register 0x02 = 0x00000000
time =  375.00 ns   ||  Register 0x03 = 0xxxxxxxxx
time =  376.00 ns   ||  Register 0x04 = 0xxxxxxxxx
time =  377.00 ns   ||  Register 0x05 = 0xxxxxxxxx
time =  378.00 ns   ||  Register 0x06 = 0xxxxxxxxx
time =  379.00 ns   ||  Register 0x07 = 0xxxxxxxxx
time =  380.00 ns   ||  Register 0x08 = 0xxxxxxxxx
time =  381.00 ns   ||  Register 0x09 = 0xxxxxxxxx
time =  382.00 ns   ||  Register 0x0a = 0xxxxxxxxx
time =  383.00 ns   ||  Register 0x0b = 0xxxxxxxxx
time =  384.00 ns   ||  Register 0x0c = 0xxxxxxxxx
time =  385.00 ns   ||  Register 0x0d = 0xxxxxxxxx
time =  386.00 ns   ||  Register 0x0e = 05a5a3c3c
time =  387.00 ns   ||  Register 0x0f = 0x10010100
```

-R1 ends with 0 after being shifted multiple times
-R2 ends with 0 after being decremented in the loop
-R14 if loaded with 5A5A3C3C
-R15 contains the pointer to the last memory location referenced+4

```
time =  412.00 ns   ||  dM[0fc] = 0xffff8000
time =  412.00 ns   ||  dM[0f8] = 0xffff0001
time =  412.00 ns   ||  dM[0f4] = 0xfffe0003
time =  412.00 ns   ||  dM[0f0] = 0xfffc0007
time =  412.00 ns   ||  dM[0ec] = 0xfff8000f
time =  412.00 ns   ||  dM[0e8] = 0xfff0001f
time =  412.00 ns   ||  dM[0e4] = 0xffe0003f
time =  412.00 ns   ||  dM[0e0] = 0xffc0007f
time =  412.00 ns   ||  dM[0dc] = 0xff8000ff
time =  412.00 ns   ||  dM[0d8] = 0xff0001ff
time =  412.00 ns   ||  dM[0d4] = 0xfe0003ff
time =  412.00 ns   ||  dM[0d0] = 0xfc0007ff
time =  412.00 ns   ||  dM[0cc] = 0xf8000fff
time =  412.00 ns   ||  dM[0c8] = 0xf0001fff
time =  412.00 ns   ||  dM[0c4] = 0xe0003fff
time =  412.00 ns   ||  dM[0c0] = 0xc0007fff
```

Each location, starting with 0x0C0, holds the result of each shift.
For example, 0x0C0 contains the first shift, 0x0C4 contains the second shift, and so on.

III. Verilog - Implementation

Memory Module 4 Instruction Memory

```
@0
3c 01 ff ff
34 21 ff ff
20 02 00 10
3c 0f 10 01
35 ef 00 c0
```

Load R1 with 0xFFFFFFFF

Load R2 with 0x10

Load R15 with 0x100100C0

```
Top:
00 01 08 40
ad e1 00 00
21 ef 00 04
20 42 ff ff
14 40 ff fb
08 10 00 0c
00 00 00 0d
```

Step 1: R1 shifted left by 1 and stored back
Step 2: Store R1 into dM pointed @ by R15
Step 3: Increment R15 by 4
Step 4: Subtract 1 from loop counter (R2)
Step 5: Check if R2 is zero (loop finished)
Go to “Exit” if finished. Else, go to “Top”
Step 6: Break if jump failed

```
Exit:
3c 0e 5a 5a
35 ce 3c 3c
```

Load R14 with 0x5A5A3C3C

```
00 00 00 0d
```

Break

III. Verilog - Implementation

Memory Module 4 Results

```
BREAK INSTRUCTION FETCHED  370.00 ns
D I S P L A Y I N G   R E G I S T E R S
time =  372.00 ns   ||  Register 0x00 = 0x00000000
time =  373.00 ns   ||  Register 0x01 = 0xffff0000
time =  374.00 ns   ||  Register 0x02 = 0x00000000
time =  375.00 ns   ||  Register 0x03 = 0xxxxxxxxx
time =  376.00 ns   ||  Register 0x04 = 0xxxxxxxxx
time =  377.00 ns   ||  Register 0x05 = 0xxxxxxxxx
time =  378.00 ns   ||  Register 0x06 = 0xxxxxxxxx
time =  379.00 ns   ||  Register 0x07 = 0xxxxxxxxx
time =  380.00 ns   ||  Register 0x08 = 0xxxxxxxxx
time =  381.00 ns   ||  Register 0x09 = 0xxxxxxxxx
time =  382.00 ns   ||  Register 0x0a = 0xxxxxxxxx
time =  383.00 ns   ||  Register 0x0b = 0xxxxxxxxx
time =  384.00 ns   ||  Register 0x0c = 0xxxxxxxxx
time =  385.00 ns   ||  Register 0x0d = 0xxxxxxxxx
time =  386.00 ns   ||  Register 0x0e = 05a5a3c3c
time =  387.00 ns   ||  Register 0x0f = 0x10010100
```

-R1 ends with 0xFFFF0000 after being shifted multiple times
-R2 ends with 0 after being decremented in the loop
-R14 if loaded with 5A5A3C3C
-R15 contains the pointer to the last memory location referenced+4

```
time =  412.00 ns   ||  dM[0fc] = 0xffff0000
time =  412.00 ns   ||  dM[0f8] = 0xffff8000
time =  412.00 ns   ||  dM[0f4] = 0xffffc000
time =  412.00 ns   ||  dM[0f0] = 0xffffe000
time =  412.00 ns   ||  dM[0ec] = 0xfffff000
time =  412.00 ns   ||  dM[0e8] = 0xfffff800
time =  412.00 ns   ||  dM[0e4] = 0xfffffc00
time =  412.00 ns   ||  dM[0e0] = 0xfffffe00
time =  412.00 ns   ||  dM[0dc] = 0xffffff00
time =  412.00 ns   ||  dM[0d8] = 0xffffff80
time =  412.00 ns   ||  dM[0d4] = 0xffffffc0
time =  412.00 ns   ||  dM[0d0] = 0xffffffe0
time =  412.00 ns   ||  dM[0cc] = 0xfffffff0
time =  412.00 ns   ||  dM[0c8] = 0xfffffff8
time =  412.00 ns   ||  dM[0c4] = 0xfffffffc
time =  412.00 ns   ||  dM[0c0] = 0xfffffffe
```

Each location, starting with 0x0C0, holds the result of each shift.
For example, 0x0C0 contains the first shift, 0x0C4 contains the second shift, and so on.

III. Verilog - Implementation

Memory Module 5 Instruction Memory

```
@0
3c 01 ff ff
34 21 ff ff
20 02 00 10
3c 0f 10 01
35 ef 00 c0

Top:
00 01 08 40
ad e1 00 00
21 ef 00 04
20 42 00 01
28 43 00 00
14 60 ff fa
08 10 00 0d
00 00 00 0d

Exit:
3c 0e 5a 5a
35 ce 3c 3c

00 00 00 0d
```

Load R1 with 0xFFFFFFFF

Load R2 with -16

Load R15 with 0x100100C0

Step 1: R1 shifted left by 1 and stored back
Step 2: Store R1 into dM pointed @ by R15
Step 3: Increment R15 by 4
Step 4: Add 1 to loop counter (R2)
Step 5: Check if R2 is under zero (loop finished) (R3 = 1)
Go to "Exit" if finished. Else, go to "Top"
Step 6: Break if jump failed

Load R14 with 0x5A5A3C3C

Break

III. Verilog - Implementation

Memory Module 5 Results

```
BREAK INSTRUCTION FETCHED  370.00 ns
D I S P L A Y I N G   R E G I S T E R S
time =  372.00 ns  ||  Register 0x00 = 0x00000000
time =  373.00 ns  ||  Register 0x01 = 0xffff0000
time =  374.00 ns  ||  Register 0x02 = 0x00000000
time =  375.00 ns  ||  Register 0x03 = 0xxxxxxxxx
time =  376.00 ns  ||  Register 0x04 = 0xxxxxxxxx
time =  377.00 ns  ||  Register 0x05 = 0xxxxxxxxx
time =  378.00 ns  ||  Register 0x06 = 0xxxxxxxxx
time =  379.00 ns  ||  Register 0x07 = 0xxxxxxxxx
time =  380.00 ns  ||  Register 0x08 = 0xxxxxxxxx
time =  381.00 ns  ||  Register 0x09 = 0xxxxxxxxx
time =  382.00 ns  ||  Register 0x0a = 0xxxxxxxxx
time =  383.00 ns  ||  Register 0x0b = 0xxxxxxxxx
time =  384.00 ns  ||  Register 0x0c = 0xxxxxxxxx
time =  385.00 ns  ||  Register 0x0d = 0xxxxxxxxx
time =  386.00 ns  ||  Register 0x0e = 05a5a3c3c
time =  387.00 ns  ||  Register 0x0f = 0x10010100
```

-R1 ends with 0xFFFF0000 after being shifted multiple times
-R2 ends with 0 after being decremented in the loop
-R14 if loaded with 5A5A3C3C
-R15 contains the pointer to the last memory location referenced+4

```
time =  412.00 ns  ||  dM[0fc] = 0xffff0000
time =  412.00 ns  ||  dM[0f8] = 0xffff8000
time =  412.00 ns  ||  dM[0f4] = 0xffffc000
time =  412.00 ns  ||  dM[0f0] = 0xffffe000
time =  412.00 ns  ||  dM[0ec] = 0xfffff000
time =  412.00 ns  ||  dM[0e8] = 0xfffff800
time =  412.00 ns  ||  dM[0e4] = 0xfffffc00
time =  412.00 ns  ||  dM[0e0] = 0xfffffe00
time =  412.00 ns  ||  dM[0dc] = 0xffffff00
time =  412.00 ns  ||  dM[0d8] = 0xffffff80
time =  412.00 ns  ||  dM[0d4] = 0xffffffc0
time =  412.00 ns  ||  dM[0d0] = 0xffffffe0
time =  412.00 ns  ||  dM[0cc] = 0xfffffff0
time =  412.00 ns  ||  dM[0c8] = 0xfffffff8
time =  412.00 ns  ||  dM[0c4] = 0xfffffffc
time =  412.00 ns  ||  dM[0c0] = 0xfffffffe
```

Each location, starting with 0x0C0, holds the result of each shift.
For example, 0x0C0 contains the first shift, 0x0C4 contains the second shift, and so on.

III. Verilog - Implementation

Memory Module 6 Instruction Memory

```
@0
3c 0f 10 01
35 ef 00 00
```

R15 <- 0x10010000

```
3c 0e 10 01
35 ce 00 c0
```

R14 <- 0x100100C0

```
20 0d 00 10
```

R13 <- 16

```
8d e1 00 04
8d e2 00 08
8d e3 00 0c
8d e4 00 10
8d e5 00 14
8d e6 00 18
8d e7 00 1c
8d e8 00 20
8d e9 00 24
8d ea 00 28
8d eb 00 2c
8d ec 00 30
```

Load words starting w/ R1 through R12. The pointer for memory is stored inside R15 w/ each lw offsets it by 4, incrementing by 4 for each instruction.

```
mem2mem:
8d f1 00 00
ad d1 00 00
21 ef 00 04
21 ce 00 04
21 ad ff ff
15 a0 ff fa
```

Step 1: R17 <- dMem[R15]
Step 2: dMem[R14] <- R17
Step 3: R15 <- R15 + 4
Step 4: R14 <- R14 + 4
Step 5: R13 <- R13 + -1
Step 6: BNE comparing R13 w/ R0, jumping to mem2me label

Break

```
00 00 00 0d
```

Data Memory

```
@00          // Big Endian Format

C3 C3 C3 C3    // 0x00:03
12 34 56 78    // 0x04:07
89 AB CD EF    // 0x08:0B
A5 A5 A5 A5    // 0x0C:0F
5A 5A 5A 5A    // 0x10:13  //word 4
24 68 AC E0    // 0x14:17
13 57 9B DF    // 0x18:1B
0F 0F 0F 0F    // 0x1C:1F
F0 F0 F0 F0    // 0x20:23  //word 8
00 00 00 09    // 0x24:27
00 00 00 0A    // 0x28:2B
00 00 00 0B    // 0x2C:2F
00 00 00 0C    // 0x30:33  //word 12
00 00 00 0D    // 0x34:37
FF FF FF F8    // 0x38:3B
00 00 75 CC    // 0x3C:3F
@1CC AB CD EF 01    // 0x1CC:1CF
@3F8 00 00 00 00    // 0x3F8:3FB
```

Memory Module 6 Results

```
BREAK INSTRUCTION FETCHED  494.50 ns
D I S P L A Y I N G   R E G I S T E R S
time =  496.10 ns   ||  Register 0x0 = 0x00000000
time =  496.20 ns   ||  Register 0x1 = 0x12345678
time =  496.30 ns   ||  Register 0x2 = 0x89abcdef
time =  496.40 ns   ||  Register 0x3 = 0xa5a5a5a5
time =  496.50 ns   ||  Register 0x4 = 0x5a5a5a5a
time =  496.60 ns   ||  Register 0x5 = 0x2468ace0
time =  496.70 ns   ||  Register 0x6 = 0x13579bdf
time =  496.80 ns   ||  Register 0x7 = 0x0f0f0f0f
time =  496.90 ns   ||  Register 0x8 = 0xf0f0f0f0
time =  497.00 ns   ||  Register 0x9 = 0x00000009
time =  497.10 ns   ||  Register 0xa = 0x0000000a
time =  497.20 ns   ||  Register 0xb = 0x0000000b
time =  497.30 ns   ||  Register 0xc = 0x0000000c
time =  497.40 ns   ||  Register 0xd = 0x00000000
time =  497.50 ns   ||  Register 0xe = 0x10010100
time =  497.60 ns   ||  Register 0xf = 0x10010040

time =  497.70 ns   ||  dM[0c0] = 0xc3c3c3c3
time =  497.80 ns   ||  dM[0c4] = 0x12345678
time =  497.90 ns   ||  dM[0c8] = 0x89abcdef
time =  498.00 ns   ||  dM[0cc] = 0xa5a5a5a5
time =  498.10 ns   ||  dM[0d0] = 0x5a5a5a5a
time =  498.20 ns   ||  dM[0d4] = 0x2468ace0
time =  498.30 ns   ||  dM[0d8] = 0x13579bdf
time =  498.40 ns   ||  dM[0dc] = 0x0f0f0f0f
time =  498.50 ns   ||  dM[0e0] = 0xf0f0f0f0
time =  498.60 ns   ||  dM[0e4] = 0x00000009
time =  498.70 ns   ||  dM[0e8] = 0x0000000a
time =  498.80 ns   ||  dM[0ec] = 0x0000000b
time =  498.90 ns   ||  dM[0f0] = 0x0000000c
time =  499.00 ns   ||  dM[0f4] = 0x0000000d
time =  499.10 ns   ||  dM[0f8] = 0xfffffff8
time =  499.20 ns   ||  dM[0fc] = 0x000075cc
```

R1 through R12 loaded in with values from Data Memory.

Loop counter variable

Data Memory pointers.

Data Memory loaded in from another part of Data Memory using the memory to memory transfer loop. Location of original data dMem[000] to dMem[03F].

III. Verilog - Implementation

Memory Module 7 Instruction Memory

```
@0
3c 0f 10 01
35 ef 00 00
```

R15 <- 0x10010000

```
3c 0e 10 01
35 ce 00 c0
```

R14 <- 0x100100C0

```
20 0d 00 10
```

R13 <- 16

```
8d e1 00 04
8d e2 00 08
8d e3 00 0c
8d e4 00 10
8d e5 00 14
8d e6 00 18
8d e7 00 1c
8d e8 00 20
8d e9 00 24
8d ea 00 28
8d eb 00 2c
8d ec 00 30
```

Load words starting w/ R1 through R12. The pointer for memory is stored inside R15 w/ each lw offsets it by 4, incrementing by 4 for each instruction.

```
0c 10 00 15
```

Jump and link to mem2mem.

```
3c 0f ff ff
35 ef ff ff
```

R15 <- FFFFFFFF

```
00 00 00 0d
```

Break

```
mem2mem:
8d f1 00 00
ad d1 00 00
21 ef 00 04
21 ce 00 04
21 ad ff ff
15 a0 ff fa
```

Step 1: R17 <- dMem[R15]
Step 2: dMem[R14] <- R17
Step 3: R15 <- R15 + 4
Step 4: R14 <- R14 + 4
Step 5: R13 <- R13 + -1
Step 6: BNE comparing R13 w/ R0, jumping to mem2me

```
03 e0 00 08
```

Jump register using R31

```
00 00 00 0d
```

Break

Data Memory

```
@0           // Big Endian Format
C3 C3 C3 C3   // 0x00:03
12 34 56 78   // 0x04:07
89 AB CD EF   // 0x08:0B
A5 A5 A5 A5   // 0x0C:0F
5A 5A 5A 5A   // 0x10:13  //word 4
24 68 AC E0   // 0x14:17
13 57 9B DF   // 0x18:1B
0F 0F 0F 0F   // 0x1C:1F
F0 F0 F0 F0   // 0x20:23  //word 8
00 00 00 09   // 0x24:27
00 00 00 0A   // 0x28:2B
00 00 00 0B   // 0x2C:2F
00 00 00 0C   // 0x30:33  //word 12
00 00 00 0D   // 0x34:37
```

```
FF FF FF F8    // 0x38:3B
00 00 75 CC    // 0x3C:3F
@1CC
AB CD EF 01    // 0x1CC:1CF
@3F8
00 00 00 00    // 0x3F8:3FB
```

Memory Module 7 Results

```
BREAK INSTRUCTION FETCHED  509.50 ns
D I S P L A Y I N G   R E G I S T E R S
time =  511.10 ns   ||  Register 0x0 = 0x00000000
time =  511.20 ns   ||  Register 0x1 = 0x12345678
time =  511.30 ns   ||  Register 0x2 = 0x89abcdef
time =  511.40 ns   ||  Register 0x3 = 0xa5a5a5a5
time =  511.50 ns   ||  Register 0x4 = 0x5a5a5a5a
time =  511.60 ns   ||  Register 0x5 = 0x2468ace0
time =  511.70 ns   ||  Register 0x6 = 0x13579bdf
time =  511.80 ns   ||  Register 0x7 = 0x0f0f0f0f
time =  511.90 ns   ||  Register 0x8 = 0xf0f0f0f0
time =  512.00 ns   ||  Register 0x9 = 0x00000009
time =  512.10 ns   ||  Register 0xa = 0x0000000a
time =  512.20 ns   ||  Register 0xb = 0x0000000b
time =  512.30 ns   ||  Register 0xc = 0x0000000c
time =  512.40 ns   ||  Register 0xd = 0x00000000
time =  512.50 ns   ||  Register 0xe = 0x10010100
time =  512.60 ns   ||  Register 0xf = 0xffffffff
```

R1 through R12 loaded in with values from Data Memory.

Loop counter variable

Data Memory pointer.

Pass flag from return from JAL.

```
time =  512.70 ns   ||  dM[0c0] = 0xc3c3c3c3
time =  512.80 ns   ||  dM[0c4] = 0x12345678
time =  512.90 ns   ||  dM[0c8] = 0x89abcdef
time =  513.00 ns   ||  dM[0cc] = 0xa5a5a5a5
time =  513.10 ns   ||  dM[0d0] = 0x5a5a5a5a
time =  513.20 ns   ||  dM[0d4] = 0x2468ace0
time =  513.30 ns   ||  dM[0d8] = 0x13579bdf
time =  513.40 ns   ||  dM[0dc] = 0x0f0f0f0f
time =  513.50 ns   ||  dM[0e0] = 0xf0f0f0f0
time =  513.60 ns   ||  dM[0e4] = 0x00000009
time =  513.70 ns   ||  dM[0e8] = 0x0000000a
time =  513.80 ns   ||  dM[0ec] = 0x0000000b
time =  513.90 ns   ||  dM[0f0] = 0x0000000c
time =  514.00 ns   ||  dM[0f4] = 0x0000000d
time =  514.10 ns   ||  dM[0f8] = 0xfffffff8
time =  514.20 ns   ||  dM[0fc] = 0x000075cc
```

Data Memory loaded in from another part of Data Memory using the memory to memory transfer loop. Location of original data dMem[000] to dMem[03F].

III. Verilog - Implementation

Memory Module 8 Instruction Memory

```
@0
3c 0f 10 01
35 ef 00 00
```

R15 <- 0x1001_0000

```
8d e1 00 00
8d e2 00 04
8d e3 00 08
8d e4 00 0c
8d e5 00 10
8d e6 00 14
8d e7 00 18
```

Loads words starting w/ R1 through R7 from Data Memory pointed at by R15. The first lw is offset by 0, then incremented by 4 every other lw.

```
00 22 00 18
00 00 40 12
14 a8 00 10
```

Multiply R1 w/ R2

R8 <- LO

BNE compare R5 w/ R8. Should NOT branch to fail1.

```
00 62 00 18
00 00 48 12
00 00 50 10
14 c9 00 0f
14 ea 00 11
```

Multiply R3 w/ R2

R9 <- LO, R8 <- HI

BNE compare R6 and R9. Should NOT branch to fail2L.

BNE compare R7 and R10. Should NOT branch to fail2H.

```
00 24 00 18
00 00 58 12
00 00 60 10
14 cb 00 10
14 ec 00 12
```

Multiply R1 w/ R4

R11 <- LO, R10 <- HI

BNE compare R6 and R11. Should NOT branch to fail3L.

BNE compare R7 and R12. Should NOT branch to fail3H.

```
00 64 00 18
00 00 68 12
14 ad 00 12
```

Multiply R3 w/ R4

R13 <- LO

BNE compare R5 w/ R13. Should NOT branch to fail4.

```
pass:
3c 0e 00 00
35 ce 00 00
00 00 00 0d
```

R14 <- 0x0000_0000, pass flag. Break.

```
fail1:
3c 0e ff ff
35 ce ff ff
00 00 00 0d
```

R14 <- 0xFFFF_FFFF, fail flag. Break.

```
fail2L:
3c 0e ff ff
35 ce ff fe
00 00 00 0d
```

R14 <- 0xFFFF_FFFE, fail flag. Break.

```
fail2H:
3c 0e ff ff
35 ce ff fd
00 00 00 0d
```

R14 <- 0xFFFF_FFFD, fail flag. Break.

```
fail3L:
3c 0e ff ff
35 ce ff fc
00 00 00 0d
```

R14 <- 0xFFFF_FFFC, fail flag. Break.

III. Verilog - Implementation

```
fail3L:
3c 0e ff ff
35 ce ff fb
00 00 00 0d
```

R14 <- 0xFFFF_FFFB, fail flag. Break.

```
fail4:
3c 0e ff ff
35 ce ff fa
00 00 00 0d
```

R14 <- 0xFFFF_FFFA, fail flag. Break.

Data Memory

```
@0          // Big Endian Format
00 00 00 19  //0x00:03   00 =   25
00 00 03 E8  //0x04:07   01 =   1000
FF FF FF E7  //0x08:0B   02 =    -25
FF FF FC 18  //0x0C:0F   03 =  -1000
00 00 61 A8  //0x10:13   04 =  25000
FF FF 9E 58  //0x14:17   05 = -25000
FF FF FF FF  //0x18:1B   06 =     -1
00 00 00 07  //0x1C:1F
00 00 00 08  //0x20:23
00 00 00 09  //0x24:27
00 00 00 0A  //0x28:2B
00 00 00 0B  //0x2C:2F
00 00 00 0C  //0x30:33
00 00 00 0D  //0x34:37
00 00 00 0E  //0x38:3B
00 00 00 0F  //0x3C:3F

@1CC
AB CD EF 01  // 0x1CC:1CF

@3F8
00 00 00 00  // 0x3F8:3FB
```

Memory Module 8 Results

```
BREAK INSTRUCTION FETCHED  119.50 ns
D I S P L A Y I N G   R E G I S T E R S
time =  121.10 ns  ||  Register 0x0 = 0x00000000
time =  121.20 ns  ||  Register 0x1 = 0x00000019
time =  121.30 ns  ||  Register 0x2 = 0x000003e8
time =  121.40 ns  ||  Register 0x3 = 0xffffffe7
time =  121.50 ns  ||  Register 0x4 = 0xfffffc18
time =  121.60 ns  ||  Register 0x5 = 0x000061a8
time =  121.70 ns  ||  Register 0x6 = 0xffff9e58
time =  121.80 ns  ||  Register 0x7 = 0xffffffff
time =  121.90 ns  ||  Register 0x8 = 0x000061a8
time =  122.00 ns  ||  Register 0x9 = 0xffff9e58
time =  122.10 ns  ||  Register 0xa = 0xffffffff
time =  122.20 ns  ||  Register 0xb = 0xffff9e58
time =  122.30 ns  ||  Register 0xc = 0xffffffff
time =  122.40 ns  ||  Register 0xd = 0x000061a8
time =  122.50 ns  ||  Register 0xe = 0x00000000
time =  122.60 ns  ||  Register 0xf = 0x10010000

time =  122.70 ns  ||  dM[0c0] = 0xxxxxxxxx
time =  122.80 ns  ||  dM[0c4] = 0xxxxxxxxx
time =  122.90 ns  ||  dM[0c8] = 0xxxxxxxxx
time =  123.00 ns  ||  dM[0cc] = 0xxxxxxxxx
time =  123.10 ns  ||  dM[0d0] = 0xxxxxxxxx
time =  123.20 ns  ||  dM[0d4] = 0xxxxxxxxx
time =  123.30 ns  ||  dM[0d8] = 0xxxxxxxxx
time =  123.40 ns  ||  dM[0dc] = 0xxxxxxxxx
time =  123.50 ns  ||  dM[0e0] = 0xxxxxxxxx
time =  123.60 ns  ||  dM[0e4] = 0xxxxxxxxx
time =  123.70 ns  ||  dM[0e8] = 0xxxxxxxxx
time =  123.80 ns  ||  dM[0ec] = 0xxxxxxxxx
time =  123.90 ns  ||  dM[0f0] = 0xxxxxxxxx
time =  124.00 ns  ||  dM[0f4] = 0xxxxxxxxx
time =  124.10 ns  ||  dM[0f8] = 0xxxxxxxxx
time =  124.20 ns  ||  dM[0fc] = 0xxxxxxxxx
```

R1 through R7 get loaded with values from Data

R8 through R13 are results with various multiples.

Pass Flag = 0x00000000

Data Memory pointer

III. Verilog - Implementation

Memory Module 9 Instruction Memory

```
@0
3c 0f 10 01
35 ef 00 00
```

R15 <- 0x1001_0000

```
8d e1 00 00
8d e2 00 04
8d e3 00 08
8d e4 00 0c
8d e5 00 10
8d e6 00 14
8d e7 00 18
8d e8 00 1c
```

Loads words starting w/ R1 through R8 from Data Memory pointed at by R15. The first lw is offset by 0, then incremented by 4 every

```
00 22 00 1a
00 00 48 12
00 00 50 10
15 25 00 16
15 46 00 18
```

Divide R1 by R2. R9 <- LO, R10 <- HI.
BNE comparing R9 w/ R5 and R10 w/ R6.
Should NOT branch to either fail1Q or fail1R.

```
00 62 00 1a
00 00 48 12
00 00 50 10
15 27 00 17
15 48 00 19
```

Divide R3 by R2. R9 <- LO, R10 <- HI.
BNE comparing R9 w/ R7 and R10 w/ R8.
Should NOT branch to either fail2Q or fail2R.

```
00 24 00 1a
00 00 48 12
00 00 50 10
15 27 00 18
15 46 00 1a
```

Divide R1 by R4. R9 <- LO, R10 <- HI.
BNE comparing R9 w/ R7 and R10 w/ R6.
Should NOT branch to either fail3Q or fail3R.

```
00 64 00 1a
00 00 48 12
00 00 50 10
15 25 00 19
15 48 00 1b
```

Divide R3 by R4. R9 <- LO, R10 <- HI.
BNE comparing R9 w/ R5 and R10 w/ R8.
Should NOT branch to either fail4Q or fail4R.

```
3c 0b 00 00
35 6b 00 00
00 0b 60 20
00 0b 68 20
00 0b 70 20
00 00 00 0d
```

pass: R11 through R14 <- 0x0000_0000
Break.

```
3c 0e ff ff
35 ce ff ff
00 00 00 0d
```

fail1Q: R14 <- 0xFFFF_FFFF and break.

```
3c 0e ff ff
35 ce ff fe
00 00 00 0d
```

fail1R: R14 <- 0xFFFF_FFFE and break.

```
3c 0e ff ff
35 ce ff fd
00 00 00 0d
```

fail2Q: R14 <- 0xFFFF_FFFD and break.

```
3c 0e ff ff
35 ce ff fc
00 00 00 0d
```

fail2R: R14 <- 0xFFFF_FFFC and break.

```
3c 0e ff ff
35 ce ff fb
00 00 00 0d
```

fail3Q: R14 <- 0xFFFF_FFFB and break.

```
3c 0e ff ff
35 ce ff fa
```

fail3R: R14 <- 0xFFFF_FFFA and break.

Data Memory

```
@0          // Big Endian Format

00 04 09 11// 0x00:03  264465
00 00 03 E8// 0x04:07  1000
FF FB F6 EF// 0x08:0B  -264465
FF FF FC 18// 0x0C:0F  -1000
00 00 01 08// 0x10:13  264
00 00 01 D1// 0x14:17  465
FF FF FE F8// 0x18:1B  -264
FF FF FE 2F// 0x1C:1F  -465
00 00 00 08// 0x20:23
00 00 00 09// 0x24:27
00 00 00 0A// 0x28:2B
00 00 00 0B// 0x2C:2F
00 00 00 0C// 0x30:33
00 00 00 0D// 0x34:37
00 00 00 0E// 0x38:3B
00 00 00 0F// 0x3C:3F

@1CC
AB CD EF 01    // 0x1CC:1CF

@3F8
00 00 00 00    // 0x3F8:3FB
```

```
00 00 00 0d
3c 0e ff ff
35 ce ff f9
00 00 00 0d
3c 0e ff ff
35 ce ff f8
00 00 00 0d
```

fail4Q: R14 <- 0xFFFF_FFF9 and break.

fail4R: R14 <- 0xFFFF_FFF8 and break.

Memory Module 9 Results

```
BREAK INSTRUCTION FETCHED  150.50 ns
D I S P L A Y I N G   R E G I S T E R S
time =  152.10 ns  ||  Register 0x0 = 0x00000000
time =  152.20 ns  ||  Register 0x1 = 0x00040911
time =  152.30 ns  ||  Register 0x2 = 0x000003e8
time =  152.40 ns  ||  Register 0x3 = 0xfffbf6ef
time =  152.50 ns  ||  Register 0x4 = 0xfffffc18
time =  152.60 ns  ||  Register 0x5 = 0x00000108
time =  152.70 ns  ||  Register 0x6 = 0x000001d1
time =  152.80 ns  ||  Register 0x7 = 0xfffffef8
time =  152.90 ns  ||  Register 0x8 = 0xfffffe2f
time =  153.00 ns  ||  Register 0x9 = 0x00000108
time =  153.10 ns  ||  Register 0xa = 0xfffffe2f
time =  153.20 ns  ||  Register 0xb = 0x00000000
time =  153.30 ns  ||  Register 0xc = 0x00000000
time =  153.40 ns  ||  Register 0xd = 0x00000000
time =  153.50 ns  ||  Register 0xe = 0x00000000
time =  153.60 ns  ||  Register 0xf = 0x10010000

time =  153.70 ns  ||  dM[0c0] = 0xxxxxxxxx
time =  153.80 ns  ||  dM[0c4] = 0xxxxxxxxx
time =  153.90 ns  ||  dM[0c8] = 0xxxxxxxxx
time =  154.00 ns  ||  dM[0cc] = 0xxxxxxxxx
time =  154.10 ns  ||  dM[0d0] = 0xxxxxxxxx
time =  154.20 ns  ||  dM[0d4] = 0xxxxxxxxx
time =  154.30 ns  ||  dM[0d8] = 0xxxxxxxxx
time =  154.40 ns  ||  dM[0dc] = 0xxxxxxxxx
time =  154.50 ns  ||  dM[0e0] = 0xxxxxxxxx
time =  154.60 ns  ||  dM[0e4] = 0xxxxxxxxx
time =  154.70 ns  ||  dM[0e8] = 0xxxxxxxxx
time =  154.80 ns  ||  dM[0ec] = 0xxxxxxxxx
time =  154.90 ns  ||  dM[0f0] = 0xxxxxxxxx
time =  155.00 ns  ||  dM[0f4] = 0xxxxxxxxx
time =  155.10 ns  ||  dM[0f8] = 0xxxxxxxxx
time =  155.20 ns  ||  dM[0fc] = 0xxxxxxxxx
```

R1 through R8 are loaded with specific values from Data Memory.

R9 and R10 loaded with last divide with registers R3 and R4.

R11 through R15 <- 0x0000_0000 to indicate pass.

Data Memory pointer

III. Verilog - Implementation

Memory Module 10 Memory Contents

```
@0
3c 0f 10 01
35 ef 00 c0
20 01 ff 8a
20 02 00 8a
0c 10 00 22
3c 0d 77 88
35 ad 77 88
3c 0c 88 77
35 8c 88 77
3c 0b ff ff
35 6b ff ff
```

Load R15 with 0x100100C0
Load R1 with 0xFFFFFF8A
Load R2 with 0x0000008A
Load R13 with 0x77887788
Load R12 with 0x88778877
Load R11 with 0xFFFFFFFF
Jumps to subroutine

```
01 ac 50 26
11 4b 00 02
20 0e ff fb
00 00 00 0d
```

-R10 gets 0xFFFFFFFF and is compared to R11
-Set R14 to 0xFFFFFFFB (fail flag)
-Branches to XOR_pass when successful

```
XOR_pass:
01 ac 48 24
11 20 00 02
20 0e ff fa
00 00 00 0d
```

-R9 gets 0x00000000 and is compared to R0
-Set R14 to 0xFFFFFFFA (fail flag)
-Branches to AND_pass when successful

```
AND_pass:
01 e2 48 25
3c 08 10 01
35 08 00 ca
11 09 00 02
20 0e ff f9
00 00 00 0d
```

-R9/R8 get 0x100100CA/0x100100CA and are compared
-Set R14 to 0xFFFFFFF9 (fail flag)
-Branches to OR_pass when successful

```
OR_pass:
01 e2 48 27
3c 08 ef fe
35 08 ff 35
11 09 00 02
20 0e ff f8
00 00 00 0d
```

-R9/R8 get 0xEFFEFF35/0xEFFEFF35and are compared
-Set R14 to 0xFFFFFFF8 (fail flag)
-Branches to NOR_pass when successful

```
NOR_pass:
ad e8 00 10
00 00 70 20
00 00 00 0d
```

-R8 is stored into dM @ R15
-R14 is cleared, indicating tests passed

```
SUBROUTINE:
00 22 18 2a
14 60 00 02
20 0e ff ff
00 00 00 0d SLT1:
20 04 00 c0
ad e4 00 00
```

-Compare R1 < R2 and branch to SLT1
-Set R14 to 0xFFFFFFFF (fail flag)
-Set M[C0] to pass flag (0xC0)

III. Verilog - Implementation

```
00 41 18 2b
14 60 00 02
20 0e ff fe
00 00 00 0d SLT2:
20 05 00 c4
ad e5 00 04
```

-Compare R2 < R1 and branch to SLT2
-Set R14 to 0xFFFFFFFE (fail flag)
-Set M[C4] to pass flag (0xC4)

```
00 41 18 2a
10 60 00 02
20 0e ff fd
00 00 00 0d SLT3:
20 06 00 c8
ad e6 00 08
```

-Compare R2 !< R1 and branch to SLT3
-Set R14 to 0xFFFFFFFD (fail flag)

```
00 22 18 2b
10 60 00 02
20 0e ff fc
00 00 00 0d SLT4:
20 07 00 cc
ad e7 00 0c
03 e0 00 08
```

-Compare R2 !< R1 and branch to SLT4
-Set R14 to 0xFFFFFFFC (fail flag)

Memory Module 10 Results

```
BREAK INSTRUCTION FETCHED  182.00 ns
D I S P L A Y I N G   R E G I S T E R S
time =  184.00 ns   ||  Register 0x00 = 0x00000000
time =  185.00 ns   ||  Register 0x01 = 0xffffff8a
time =  186.00 ns   ||  Register 0x02 = 0x0000008a
time =  187.00 ns   ||  Register 0x03 = 0x00000000
time =  188.00 ns   ||  Register 0x04 = 0x000000c0
time =  189.00 ns   ||  Register 0x05 = 0x000000c4
time =  190.00 ns   ||  Register 0x06 = 0x000000c8
time =  191.00 ns   ||  Register 0x07 = 0x000000cc
time =  192.00 ns   ||  Register 0x08 = 0xeffeff35
time =  193.00 ns   ||  Register 0x09 = 0xeffeff35
time =  194.00 ns   ||  Register 0x0a = 0xffffffff
time =  195.00 ns   ||  Register 0x0b = 0xffffffff
time =  196.00 ns   ||  Register 0x0c = 0x88778877
time =  197.00 ns   ||  Register 0x0d = 0x77887788
time =  198.00 ns   ||  Register 0x0e = 0x00000000
time =  199.00 ns   ||  Register 0x0f = 0x100100c0
```

-R1-R3 & R8-R13: scratch registers used in op's
-R4-R7: pass flags

Eventually cleared when all tests pass

Memory pointer

```
time =  412.00 ns   ||  dM[0c0] = 0x000000c0
time =  412.00 ns   ||  dM[0c4] = 0x000000c4
time =  412.00 ns   ||  dM[0c8] = 0x000000c8
time =  412.00 ns   ||  dM[0cc] = 0x000000cc
time =  412.00 ns   ||  dM[0d0] = 0xeffeff35
time =  412.00 ns   ||  dM[0d4] = 0xxxxxxxxx
time =  412.00 ns   ||  dM[0d8] = 0xxxxxxxxx
time =  412.00 ns   ||  dM[0dc] = 0xxxxxxxxx
time =  412.00 ns   ||  dM[0e0] = 0xxxxxxxxx
time =  412.00 ns   ||  dM[0e4] = 0xxxxxxxxx
time =  412.00 ns   ||  dM[0e8] = 0xxxxxxxxx
time =  412.00 ns   ||  dM[0ec] = 0xxxxxxxxx
time =  412.00 ns   ||  dM[0f0] = 0xxxxxxxxx
time =  412.00 ns   ||  dM[0f4] = 0xxxxxxxxx
time =  412.00 ns   ||  dM[0f8] = 0xxxxxxxxx
time =  412.00 ns   ||  dM[0fc] = 0xxxxxxxxx
```

Pass flags

III. Verilog - Implementation

Memory Module 11 Memory Contents

```
@0
3c 0f 10 01
35 ef 00 c0
20 01 ff 8a
20 02 00 8a
0c 10 00 08
ad e1 00 18
ad e2 00 1c
00 00 00 0d
```

Load R15 with 0x100100C0
Load R1 with 0xFFFFFF8A
Load R2 with 0x0000008A
-Jump to subroutine and when we return:
dM[D8] <- R1, dM[DC] <- R2, then break

```
SUBROUTINE:
18 20 00 02
20 0e ff ff
00 00 00 0d BLEZP1:
```

Branch if(R1 <= 0) to BLEP1
(passes)

```
20 03 00 c0
ad e3 00 00
18 40 00 03
20 04 00 c4
ad e4 00 04
08 10 00 13 BLEZF2:
20 0e ff fe
```

Branch if(R2 <= 0) to BLEZF2 (fails)
-dM[C0] <- 0xC0, dm[C4] <- 0xC4
-R14 gets 0xFFFFFFFE (fail flag)

```
00 00 00 0d BLEZP2:
18 00 00 02
20 0e ff fd
00 00 00 0d BLEZP3:
20 05 00 c8
ad e5 00 08
```

Branch if(R0 <= 0) to BLEZP3
(passes)
-dM[C8] <- 0xC8
-R14 gets 0xFFFFFFFD (fail flag)

```
1c 40 00 02
20 0e ff fc
00 00 00 0d BGTZP1:
```

Branch if(R2 > 0) to BGTZP1
(passes)

```
20 06 00 cc
ad e6 00 0c
1c 20 00 03
20 07 00 d0
ad e7 00 10
08 10 00 23 BGTZF2:
20 0e ff fb
00 00 00 0d BGTZP2:
```

Branch if(R1 > 0) to BGTZF2 (fails)
-dM[CC] <- 0xCC, dm[D0] <- 0xD0
-R14 gets 0xFFFFFFFB (fail flag)

```
1c 20 00 03
20 08 00 d4
ad e8 00 14
08 10 00 29 BGTZF3:
20 0e ff fa
00 00 00 0d BGTZP3:
```

Branch if(R1 > 0) to BGTZF2 (fails)
-dM[D4] <- 0xD4
-R14 gets 0xFFFFFFFA (fail flag)

```
20 0e 00 00
03 e0 00 08
```

Return from subroutine
-R14 gets cleared to signify tests passed

III. Verilog - Implementation

Memory Module 11 Results

```
BREAK INSTRUCTION FETCHED  131.00 ns
D I S P L A Y I N G   R E G I S T E R S
time =  133.00 ns  ||  Register 0x00 = 0x00000000
time =  134.00 ns  ||  Register 0x01 = 0xffffff8a
time =  135.00 ns  ||  Register 0x02 = 0x0000008a
time =  136.00 ns  ||  Register 0x03 = 0x000000c0
time =  137.00 ns  ||  Register 0x04 = 0x000000c4
time =  138.00 ns  ||  Register 0x05 = 0x000000c8
time =  139.00 ns  ||  Register 0x06 = 0x000000cc
time =  140.00 ns  ||  Register 0x07 = 0x000000d0
time =  141.00 ns  ||  Register 0x08 = 0x000000d4
time =  142.00 ns  ||  Register 0x09 = 0xxxxxxxxx
time =  143.00 ns  ||  Register 0x0a = 0xxxxxxxxx
time =  144.00 ns  ||  Register 0x0b = 0xxxxxxxxx
time =  145.00 ns  ||  Register 0x0c = 0xxxxxxxxx
time =  146.00 ns  ||  Register 0x0d = 0xxxxxxxxx
time =  147.00 ns  ||  Register 0x0e = 0x00000000
time =  148.00 ns  ||  Register 0x0f = 0x100100c0
```

Values used in comparisons

Pass flags

Memory pointer

```
time =  412.00 ns  ||  dM[0c0] = 0x000000c0
time =  412.00 ns  ||  dM[0c4] = 0x000000c4
time =  412.00 ns  ||  dM[0c8] = 0x000000c8
time =  412.00 ns  ||  dM[0cc] = 0x000000cc
time =  412.00 ns  ||  dM[0d0] = 0x000000d0
time =  412.00 ns  ||  dM[0d4] = 0x000000d4
time =  412.00 ns  ||  dM[0d8] = 0xffffff8a
time =  412.00 ns  ||  dM[0dc] = 0x0000008a
time =  412.00 ns  ||  dM[0e0] = 0xxxxxxxxx
time =  412.00 ns  ||  dM[0e4] = 0xxxxxxxxx
time =  412.00 ns  ||  dM[0e8] = 0xxxxxxxxx
time =  412.00 ns  ||  dM[0ec] = 0xxxxxxxxx
time =  412.00 ns  ||  dM[0f0] = 0xxxxxxxxx
time =  412.00 ns  ||  dM[0f4] = 0xxxxxxxxx
time =  412.00 ns  ||  dM[0f8] = 0xxxxxxxxx
time =  412.00 ns  ||  dM[0fc] = 0xxxxxxxxx
```

R1/R2 shown here after subroutine finishes

Pass flags

R1/R2 stored here to signify tests passed

III. Verilog - Implementation

Memory Module 12 Memory Contents

```
@0
3c 0f 10 01
35 ef 00 c0
20 01 ff 8a
20 02 00 8a
0c 10 00 1a
```

Load R15 with 0x100100C0
Load R1 with 0xFFFFFF8A
Load R2 with 0x0000008A
Jump to subroutine

```
3c 0d ff ff
35 ad 55 55
3c 0c ff ff
35 8c fa f5
3c 0b ff ff
35 6b ff ff
3c 0a 00 00
35 4a f0 f0
```

Load R13 with 0xFFFF5555
Load R12 with 0xFFFFFAF5
Load R11 with 0xFFFFFFFF
Load R10 with 0x0000F0F0

```
39 a9 aa aa
01 2b 40 22
11 00 00 02
20 0e ff f9
00 00 00 0d XOR1:
```

If(R8 == 0) branch to XOR1 (passes)
Load R9 with 0xFFFFFFFF
Load R8 with: R9 - R11
-R14 gets 0xFFFFFFF9 (fail flag)

```
31 87 f5 fa
00 ea 40 22
11 00 00 02
20 0e ff f8
00 00 00 0d XOR2:
```

If(R8 == 0) branch to XOR2 (passes)
Load R7 with 0x0000F0F0
Load R8 with: R7 - R10
-R14 gets 0xFFFFFFF8 (fail flag)

```
ad e1 00 18
00 00 00 0d
00 00 00 0d
```

dM[D8] <- 0xFFFFFF8A and break

```
SUBROUTINE
2c 23 ff 8b
14 60 00 02
20 0e ff ff
00 00 00 0d SLT1:
20 04 00 c0
ad e4 00 00
```

If(R1 < 0xFF8B) branch to SLT1 (passes)
dM[C0] <- 0xC0
-R14 gets 0xFFFFFFFF (fail flag)

```
2c 23 ff 89
10 60 00 02
20 0e ff fe
00 00 00 0d SLT2:
20 05 00 c4
ad e5 00 04
```

If(R1 < 0xFF89) branch to SLT2 (passes)
dM[C4] <- 0xC4
-R14 gets 0xFFFFFFFE (fail flag)

```
2c 23 ff 8a
10 60 00 02
20 0e ff fd
00 00 00 0d SLT3:
20 06 00 c8
ad e6 00 08
```

If(R1 < 0xFF8A) branch to SLT3 (passes)
dM[C8] <- 0xC8
-R14 gets 0xFFFFFFFD (fail flag)

```
2c 43 00 8b
14 60 00 02
20 0e ff fc
00 00 00 0d SLT4:
20 07 00 cc
ad e7 00 0c
```

If(R2 < 0x008B) branch to SLT4 (passes)
dM[CC] <- 0xCC
-R14 gets 0xFFFFFFFC (fail flag)

```
2c 43 00 89
10 60 00 02
20 0e ff fb
00 00 00 0d SLT5:
20 08 00 d0
ad e8 00 10
```

If(R2 < 0x0089) branch to SLT5 (passes)
dM[D0] <- 0xD0
-R14 gets 0xFFFFFFFB (fail flag)

```
2c 43 00 8a
10 60 00 02
20 0e ff fa
00 00 00 0d SLT6:
20 06 00 d4
ad e6 00 14
```

If(R2 < 0x008A) branch to SLT6 (passes)
dM[D4] <- 0xD4

```
20 0e 00 00
03 e0 00 08
```

Set R14 to 0 and return from subroutine

Memory Module 12 Results

```
BREAK INSTRUCTION FETCHED  198.00 ns
D I S P L A Y I N G   R E G I S T E R S
time =  200.00 ns   ||  Register 0x00 = 0x00000000
time =  201.00 ns   ||  Register 0x01 = 0xffffff8a
time =  202.00 ns   ||  Register 0x02 = 0x0000008a
time =  203.00 ns   ||  Register 0x03 = 0x00000000
time =  204.00 ns   ||  Register 0x04 = 0x000000c0
time =  205.00 ns   ||  Register 0x05 = 0x000000c4
time =  206.00 ns   ||  Register 0x06 = 0x000000d4
time =  207.00 ns   ||  Register 0x07 = 0x0000f0f0
time =  208.00 ns   ||  Register 0x08 = 0x00000000
time =  209.00 ns   ||  Register 0x09 = 0xffffffff
time =  210.00 ns   ||  Register 0x0a = 0x0000f0f0
time =  211.00 ns   ||  Register 0x0b = 0xffffffff
time =  212.00 ns   ||  Register 0x0c = 0xfffffaf5
time =  213.00 ns   ||  Register 0x0d = 0xffff5555
time =  214.00 ns   ||  Register 0x0e = 0x00000000
time =  215.00 ns   ||  Register 0x0f = 0x100100c0
```

Values used in comparisons (Registers 0x01–0x02)

Used for pass flags & arithmetic (Registers 0x03–0x09)

Start patterns (Registers 0x0a–0x0d)

Memory pointer (Register 0x0f)

```
time =  412.00 ns   ||  dM[0c0] = 0x000000c0
time =  412.00 ns   ||  dM[0c4] = 0x000000c4
time =  412.00 ns   ||  dM[0c8] = 0x000000c8
time =  412.00 ns   ||  dM[0cc] = 0x000000cc
time =  412.00 ns   ||  dM[0d0] = 0x000000d0
time =  412.00 ns   ||  dM[0d4] = 0x000000d4
time =  412.00 ns   ||  dM[0d8] = 0xffffff8a
time =  412.00 ns   ||  dM[0dc] = 0xxxxxxxxx
time =  412.00 ns   ||  dM[0e0] = 0xxxxxxxxx
time =  412.00 ns   ||  dM[0e4] = 0xxxxxxxxx
time =  412.00 ns   ||  dM[0e8] = 0xxxxxxxxx
```

Pass flags (dM[0c0]–dM[0d4])

R1 stored here to signify all tests passed (dM[0d8])

```
time =  412.00 ns   ||  dM[0ec] = 0xxxxxxxxx
time =  412.00 ns   ||  dM[0f0] = 0xxxxxxxxx
time =  412.00 ns   ||  dM[0f4] = 0xxxxxxxxx
time =  412.00 ns   ||  dM[0f8] = 0xxxxxxxxx
time =  412.00 ns   ||  dM[0fc] = 0xxxxxxxxx
```

III. Verilog - Implementation

Memory Module 13 Instruction Memory

```
@0
00 00 00 1f
```

Set interrupt flag to 1

```
3c 01 12 34
34 21 56 78
3c 02 87 65
34 42 43 21
3c 03 ab cd
34 63 ef 01
3c 04 01 fe
34 84 dc ba
3c 05 5a 5a
34 a5 5a 5a
3c 06 ff ff
34 c6 ff ff
3c 07 ff ff
34 e7 ff 00
```

R1<-0x1234_5678, R2<-0x8765_4321, R3<-0xABCD_EF01, R4<-0x01FE_DCBA, R5<-0x5A5A_5A5A,R6<-0xFFFF_FFFF, R7<-0xFFFF_FF00

```
00 c7 40 20
00 c8 48 20
00 c9 50 20
00 ca 58 20
00 cb 60 20
00 cc 68 20
00 cd 70 20
00 ce 78 20
```

Add instruction that take R6 and R(n-1) and store it into Rn, starting with n=8 to n=15. Result is one less than the previous result.

```
3c 07 10 01
34 e7 03 f0
ac ef 00 00
00 00 00 0d
```

R7 gets 0x1001_03F0, then store R15 into location dM[R7], then break.

```
@200
```

Interrupt vector.

```
3c 10 10 01
36 10 00 c0
3c 11 80 00
36 31 ff ff
20 12 00 10
```

R16<-0x1001_00C0, R17<-0x8000_FFFF, R18<- 0x0000_0010

```
76 11 00 00 out_IO:
00 11 88 43
22 10 00 04
22 52 ff ff
16 40 ff fb
```

Loop that outputs R17 to ioM[R16], shifts right arithmetically R17 by 1, increments R16 by 4 and R18 by -1, then compares R18 with R0 in a BNE. Jumps to out_IO if true.

```
3c 10 10 01
36 10 00 c0
```

R16 <- 1001_00C0

```
72 13 00 00
72 14 00 04
72 15 00 08
72 16 00 0c
72 17 00 10
72 18 00 14
```

Input instructions that load ioM[R16] into R19 and goes to R24. Each instruction adds 4 to the offset, starting at 0.

```
03 e0 00 08
```

Jump register R31 to return from interrupt

III. Verilog - Implementation

Memory Module 13 Results

```
BREAK INSTRUCTION FETCHED  504.00 ns
D I S P L A Y I N G   R E G I S T E R S
time =  506.00 ns   ||  Register 0x00 = 0x00000000
time =  507.00 ns   ||  Register 0x01 = 0x12345678
time =  508.00 ns   ||  Register 0x02 = 0x87654321
time =  509.00 ns   ||  Register 0x03 = 0xabcdef01
time =  510.00 ns   ||  Register 0x04 = 0x01fedcba
time =  511.00 ns   ||  Register 0x05 = 0x5a5a5a5a
time =  512.00 ns   ||  Register 0x06 = 0xffffffff
time =  513.00 ns   ||  Register 0x07 = 0x100103f0
time =  514.00 ns   ||  Register 0x08 = 0xfffffeff
time =  515.00 ns   ||  Register 0x09 = 0xfffffefe
time =  516.00 ns   ||  Register 0x0a = 0xfffffefd
time =  517.00 ns   ||  Register 0x0b = 0xfffffefc
time =  518.00 ns   ||  Register 0x0c = 0xfffffefb
time =  519.00 ns   ||  Register 0x0d = 0xfffffefa
time =  520.00 ns   ||  Register 0x0e = 0xfffffef9
time =  521.00 ns   ||  Register 0x0f = 0xfffffef8
time =  522.00 ns   ||  Register 0x10 = 0x100100c0
time =  523.00 ns   ||  Register 0x11 = 0xffff8000
time =  524.00 ns   ||  Register 0x12 = 0x00000000
time =  525.00 ns   ||  Register 0x13 = 0x8000ffff
time =  526.00 ns   ||  Register 0x14 = 0xc0007fff
time =  527.00 ns   ||  Register 0x15 = 0xe0003fff
time =  528.00 ns   ||  Register 0x16 = 0xf0001fff
time =  529.00 ns   ||  Register 0x17 = 0xf8000fff
time =  530.00 ns   ||  Register 0x18 = 0xfc0007ff
```

R1 through R6 loaded with specific values.

Results from the add instructions. Results were one less than the previous register.

Memory pointer

R17 result from shifting

Loop counter variable.

Result from Input instructions. Loaded from ioMemory 0x0c0 through 0x0d4.

```
time =  531.00 ns   ||  ioM[0c0] = 0x8000ffff
time =  532.00 ns   ||  ioM[0c4] = 0xc0007fff
time =  533.00 ns   ||  ioM[0c8] = 0xe0003fff
time =  534.00 ns   ||  ioM[0cc] = 0xf0001fff
time =  535.00 ns   ||  ioM[0d0] = 0xf8000fff
time =  536.00 ns   ||  ioM[0d4] = 0xfc0007ff
time =  537.00 ns   ||  ioM[0d8] = 0xfe0003ff
time =  538.00 ns   ||  ioM[0dc] = 0xff0001ff
time =  539.00 ns   ||  ioM[0e0] = 0xff8000ff
time =  540.00 ns   ||  ioM[0e4] = 0xffc0007f
time =  541.00 ns   ||  ioM[0e8] = 0xffe0003f
time =  542.00 ns   ||  ioM[0ec] = 0xfff0001f
time =  543.00 ns   ||  ioM[0f0] = 0xfff8000f
time =  544.00 ns   ||  ioM[0f4] = 0xfffc0007
time =  545.00 ns   ||  ioM[0f8] = 0xfffe0003
time =  546.00 ns   ||  ioM[0fc] = 0xffff0001
```

Results from Output loop with shift right arithmetic instructions into ioMemory.

III. Verilog - Implementation

Memory Module 14 Instruction Memory

```
@0
00 00 00 1f
```

Set interrupt flag to 1

```
3c 01 12 34
34 21 56 78
3c 02 87 65
34 42 43 21
3c 03 ab cd
34 63 ef 01
3c 04 01 fe
34 84 dc ba
3c 05 5a 5a
34 a5 5a 5a
3c 06 ff ff
34 c6 ff ff
3c 07 ff ff
34 e7 ff 00
```

R1<-0x1234_5678,
R2<-0x8765_4321,
R3<-0xABCD_EF01,
R4<-0x01FE_DCBA,
R5<-0x5A5A_5A5A,R6<-0xFFFF_

```
00 c7 40 20
00 c8 48 20
00 c9 50 20
00 ca 58 20
00 cb 60 20
00 cc 68 20
00 cd 70 20
00 ce 78 20
```

Add instruction that take R6 and R(n-1) and store it into Rn, starting with n=8 to n=15. Result is one less than the previous result.

```
3c 07 10 01
34 e7 03 f0
ac ef 00 00
00 00 00 0d
```

R7 gets 0x1001_03F0, then store R15 into location dM[R7], then break.

```
@200
```

Interrupt vector.

```
3c 10 10 01
36 10 00 c0
3c 11 80 00
36 31 ff ff
20 12 00 10
```

R16<-0x1001_00C0, R17<-0x8000_FFFF, R18<- 0x0000_0010

```
76 11 00 00 out_IO:
00 11 88 43
22 10 00 04
22 52 ff ff
16 40 ff fb
```

Loop that outputs R17 to ioM[R16], shifts right arithmetically R17 by 1, increments R16 by 4 and R18 by -1, then compares R18 with R0 in a BNE. Jumps to out_IO if true.

```
3c 10 10 01
36 10 00 c0
```

R16 <- 1001_00C0

```
72 13 00 00
72 14 00 04
72 15 00 08
72 16 00 0c
72 17 00 10
72 18 00 14
```

Input instructions that load ioM[R16] into R19 and goes to R24. Each instruction adds 4 to the offset, starting at 0.

```
03 e0 00 08
```

Return from interrupt instruction.

Memory Module 14 Results

```
Current flags before intr: psd=x,psi=1,psc=x,psv=x,psn=0,psz=0
```

Showing flags before interrupt states are saved in memory/

```
Flags in data memory:
time =   57.10 ns  ||  dM[3f7] = 0x00x1xx00
```

Shows where flags are in memory.

```
Flags before return: psd=x,psi=0,psc=0,psv=0,psn=0,psz=0
Updated flags after return: psd=x,psi=1,psc=x,psv=x,psn=0,psz=0
```

Showing the flags before the POP and after. Match with flags before intr.

```
BREAK INSTRUCTION FETCHED  504.00 ns
D I S P L A Y I N G   R E G I S T E R S
time =  506.00 ns   ||  Register 0x00 = 0x00000000
time =  507.00 ns   ||  Register 0x01 = 0x12345678
time =  508.00 ns   ||  Register 0x02 = 0x87654321
time =  509.00 ns   ||  Register 0x03 = 0xabcdef01
time =  510.00 ns   ||  Register 0x04 = 0x01fedcba
time =  511.00 ns   ||  Register 0x05 = 0x5a5a5a5a
time =  512.00 ns   ||  Register 0x06 = 0xffffffff
```

R1 through R6 loaded with specific values.

```
time =  513.00 ns   ||  Register 0x07 = 0x100103f0
time =  514.00 ns   ||  Register 0x08 = 0xfffffeff
time =  515.00 ns   ||  Register 0x09 = 0xfffffefe
time =  516.00 ns   ||  Register 0x0a = 0xfffffefd
time =  517.00 ns   ||  Register 0x0b = 0xfffffefc
time =  518.00 ns   ||  Register 0x0c = 0xfffffefb
time =  519.00 ns   ||  Register 0x0d = 0xfffffefa
time =  520.00 ns   ||  Register 0x0e = 0xfffffef9
time =  521.00 ns   ||  Register 0x0f = 0xfffffef8
```

Results from the add instructions. Results were one less than the previous register.

Memory pointer

```
time =  522.00 ns   ||  Register 0x10 = 0x100100c0
time =  523.00 ns   ||  Register 0x11 = 0xffff8000
time =  524.00 ns   ||  Register 0x12 = 0x00000000
```

R17 result from shifting

Loop counter variable.

```
time =  525.00 ns   ||  Register 0x13 = 0x8000ffff
time =  526.00 ns   ||  Register 0x14 = 0xc0007fff
time =  527.00 ns   ||  Register 0x15 = 0xe0003fff
time =  528.00 ns   ||  Register 0x16 = 0xf0001fff
time =  529.00 ns   ||  Register 0x17 = 0xf8000fff
time =  530.00 ns   ||  Register 0x18 = 0xfc0007ff
```

Result from Input instructions.   Loaded from ioMemory 0x0c0 through 0x0d4.

```
time =  531.00 ns   ||  ioM[0c0] = 0x8000ffff
time =  532.00 ns   ||  ioM[0c4] = 0xc0007fff
time =  533.00 ns   ||  ioM[0c8] = 0xe0003fff
time =  534.00 ns   ||  ioM[0cc] = 0xf0001fff
time =  535.00 ns   ||  ioM[0d0] = 0xf8000fff
time =  536.00 ns   ||  ioM[0d4] = 0xfc0007ff
time =  537.00 ns   ||  ioM[0d8] = 0xfe0003ff
time =  538.00 ns   ||  ioM[0dc] = 0xff0001ff
time =  539.00 ns   ||  ioM[0e0] = 0xff8000ff
time =  540.00 ns   ||  ioM[0e4] = 0xffc0007f
time =  541.00 ns   ||  ioM[0e8] = 0xffe0003f
time =  542.00 ns   ||  ioM[0ec] = 0xfff0001f
time =  543.00 ns   ||  ioM[0f0] = 0xfff8000f
time =  544.00 ns   ||  ioM[0f4] = 0xfffc0007
time =  545.00 ns   ||  ioM[0f8] = 0xfffe0003
time =  546.00 ns   ||  ioM[0fc] = 0xffff0001
```

Results from Output loop with shift right arithmetic instructions into ioMemory.

Memory Module 15a Memory Contents

```
@0
3c 01 ff ff
34 21 ff ff
```

Load R1 with 0xFFFFFFFF

```
20 02 00 10
```

Load R2 with 0x10

```
3c 0f 10 01
35 ef 00 c0
```

Load R15 with 0x100100C0

```
Top:
00 01 08 82
ad e1 00 00
21 ef 00 04
20 42 ff ff
14 40 ff fb
08 10 00 0c
00 00 00 0d
```

Step 1: R1 shifted right by 2 and stored back
Step 2: Store R1 into dM pointed @ by R15
Step 3: Increment R15 by 4
Step 4: Subtract 1 from loop counter (R2)
Step 5: Check if R2 is zero (loop finished)
Go to “Exit” if finished. Else, go to “Top”
Step 6: Break if jump failed

```
Exit:
3c 0e 5a 5a
35 ce 3c 3c
```

Load R14 with 0x5A5A3C3C

```
00 0e 70 b4
```

Rotate R14 right by 2

```
00 00 00 0d
```

Break

III. Verilog - Implementation

Memory Module 15a Results

```
BREAK INSTRUCTION FETCHED  370.00 ns
D I S P L A Y I N G   R E G I S T E R S
time =  372.00 ns  ||  Register 0x00 = 0x00000000
time =  373.00 ns  ||  Register 0x01 = 0x00000000
time =  374.00 ns  ||  Register 0x02 = 0x00000000
time =  375.00 ns  ||  Register 0x03 = 0xxxxxxxxx
time =  376.00 ns  ||  Register 0x04 = 0xxxxxxxxx
time =  377.00 ns  ||  Register 0x05 = 0xxxxxxxxx
time =  378.00 ns  ||  Register 0x06 = 0xxxxxxxxx
time =  379.00 ns  ||  Register 0x07 = 0xxxxxxxxx
time =  380.00 ns  ||  Register 0x08 = 0xxxxxxxxx
time =  381.00 ns  ||  Register 0x09 = 0xxxxxxxxx
time =  382.00 ns  ||  Register 0x0a = 0xxxxxxxxx
time =  383.00 ns  ||  Register 0x0b = 0xxxxxxxxx
time =  384.00 ns  ||  Register 0x0c = 0xxxxxxxxx
time =  385.00 ns  ||  Register 0x0d = 0xxxxxxxxx
time =  386.00 ns  ||  Register 0x0e = 0x16968f0f
time =  387.00 ns  ||  Register 0x0f = 0x10010100
```

-R1 ends with 0 after being shifted multiple times
-R2 ends with 0 after being decremented in the loop
-R14 contains the result of the rotate right.
-R15 contains the pointer to the last memory location referenced+4

```
time =  412.00 ns  ||  dM[0c0] = 0x3fffffff
time =  412.00 ns  ||  dM[0c4] = 0x0fffffff
time =  412.00 ns  ||  dM[0c8] = 0x03ffffff
time =  412.00 ns  ||  dM[0cc] = 0x00ffffff
time =  412.00 ns  ||  dM[0d0] = 0x003fffff
time =  412.00 ns  ||  dM[0d4] = 0x000fffff
time =  412.00 ns  ||  dM[0d8] = 0x0003ffff
time =  412.00 ns  ||  dM[0dc] = 0x0000ffff
time =  412.00 ns  ||  dM[0e0] = 0x00003fff
time =  412.00 ns  ||  dM[0e4] = 0x00000fff
time =  412.00 ns  ||  dM[0e8] = 0x000003ff
time =  412.00 ns  ||  dM[0ec] = 0x000000ff
time =  412.00 ns  ||  dM[0f0] = 0x0000003f
time =  412.00 ns  ||  dM[0f4] = 0x0000000f
time =  412.00 ns  ||  dM[0f8] = 0x00000003
time =  412.00 ns  ||  dM[0fc] = 0x00000000
```

Each location, starting with 0x0C0, holds the result of each shift.
For example, 0x0C0 contains the first shift, 0x0C4 contains the second shift, and so on.

III. Verilog - Implementation

Memory Module 15b Memory Contents

```
@0
3c 01 80 00
34 21 ff ff
20 02 00 10
3c 0f 10 01
35 ef 00 c0
Top:
00 01 08 83
ad e1 00 00
21 ef 00 04
20 42 ff ff
14 40 ff fb
08 10 00 0c
00 00 00 0d


Exit:
3c 0e 5a 5a
35 ce 3c 3c
00 0e 70 b4

00 00 00 0d
```

Load R1 with 0x8000FFFF

Load R2 with 0x10

Load R15 with 0x100100C0

Step 1: R1 shifted right (arithmetic) by 2 and stored back
Step 2: Store R1 into dM pointed @ by R15
Step 3: Increment R15 by 4
Step 4: Subtract 1 from loop counter (R2)
Step 5: Check if R2 is zero (loop finished)
Go to "Exit" if finished. Else, go to "Top"
Step 6: Break if jump failed

Load R14 with 0x5A5A3C3C

Rotate R14 right by 2

Break

III. Verilog - Implementation

Memory Module 15b Results

```
BREAK INSTRUCTION FETCHED  370.00 ns
D I S P L A Y I N G   R E G I S T E R S
time =  372.00 ns   ||  Register 0x00 = 0x00000000
time =  373.00 ns   ||  Register 0x01 = 0xaaaaaaaa
time =  374.00 ns   ||  Register 0x02 = 0x00000000
time =  375.00 ns   ||  Register 0x03 = 0xxxxxxxxx
time =  376.00 ns   ||  Register 0x04 = 0xxxxxxxxx
time =  377.00 ns   ||  Register 0x05 = 0xxxxxxxxx
time =  378.00 ns   ||  Register 0x06 = 0xxxxxxxxx
time =  379.00 ns   ||  Register 0x07 = 0xxxxxxxxx
time =  380.00 ns   ||  Register 0x08 = 0xxxxxxxxx
time =  381.00 ns   ||  Register 0x09 = 0xxxxxxxxx
time =  382.00 ns   ||  Register 0x0a = 0xxxxxxxxx
time =  383.00 ns   ||  Register 0x0b = 0xxxxxxxxx
time =  384.00 ns   ||  Register 0x0c = 0xxxxxxxxx
time =  385.00 ns   ||  Register 0x0d = 0xxxxxxxxx
time =  386.00 ns   ||  Register 0x0e = 0x16968f0f
time =  387.00 ns   ||  Register 0x0f = 0x10010100
```

-R1 ends with 0xAAAAAAAA after being shifted multiple times
-R2 ends with 0 after being decremented in the loop
-R14 contains the result of the rotate right.
-R15 contains the pointer to the last memory location referenced+4

```
time =  412.00 ns   ||  dM[0c0] = 0xa0003fff
time =  412.00 ns   ||  dM[0c4] = 0xa8000fff
time =  412.00 ns   ||  dM[0c8] = 0xaa000fff
time =  412.00 ns   ||  dM[0cc] = 0xaa8000ff
time =  412.00 ns   ||  dM[0d0] = 0xaaa0003f
time =  412.00 ns   ||  dM[0d4] = 0xaaa8000f
time =  412.00 ns   ||  dM[0d8] = 0xaaaa8003
time =  412.00 ns   ||  dM[0dc] = 0xaaaa8000
time =  412.00 ns   ||  dM[0e0] = 0xaaaaa000
time =  412.00 ns   ||  dM[0e4] = 0xaaaaa800
time =  412.00 ns   ||  dM[0e8] = 0xaaaaaa00
time =  412.00 ns   ||  dM[0ec] = 0xaaaaaa80
time =  412.00 ns   ||  dM[0f0] = 0xaaaaaaa0
time =  412.00 ns   ||  dM[0f4] = 0xaaaaaaa8
time =  412.00 ns   ||  dM[0f8] = 0xaaaaaaaa
time =  412.00 ns   ||  dM[0fc] = 0xaaaaaaaa
```

Each location, starting with 0x0C0, holds the result of each shift.
For example, 0x0C0 contains the first shift, 0x0C4 contains the second shift, and so on.

Memory Module 15c Memory Contents

```
@0
3c 01 80 00
34 21 ff ff
20 02 00 10
3c 0f 10 01
35 ef 00 c0
```

Load R1 with 0x8000FFFF

Load R2 with 0x10

Load R15 with 0x100100C0

```
Top:
00 01 08 80
ad e1 00 00
21 ef 00 04
20 42 ff ff
14 40 ff fb
08 10 00 0c
00 00 00 0d
```

Step 1: R1 shifted left by 2 and stored back
Step 2: Store R1 into dM pointed @ by R15
Step 3: Increment R15 by 4
Step 4: Subtract 1 from loop counter (R2)
Step 5: Check if R2 is zero (loop finished)
Go to “Exit” if finished. Else, go to “Top”
Step 6: Break if jump failed

```
Exit:
3c 0e 5a 5a
35 ce 3c 3c
00 0e 70 b4

00 00 00 0d
```

Load R14 with 0x5A5A3C3C

Rotate R14 right by 2

Break

III. Verilog - Implementation

Memory Module 15c Results

```
BREAK INSTRUCTION FETCHED  370.00 ns
D I S P L A Y I N G   R E G I S T E R S
time =  372.00 ns  ||  Register 0x00 = 0x00000000
time =  373.00 ns  ||  Register 0x01 = 0x00000000
time =  374.00 ns  ||  Register 0x02 = 0x00000000
time =  375.00 ns  ||  Register 0x03 = 0xxxxxxxxx
time =  376.00 ns  ||  Register 0x04 = 0xxxxxxxxx
time =  377.00 ns  ||  Register 0x05 = 0xxxxxxxxx
time =  378.00 ns  ||  Register 0x06 = 0xxxxxxxxx
time =  379.00 ns  ||  Register 0x07 = 0xxxxxxxxx
time =  380.00 ns  ||  Register 0x08 = 0xxxxxxxxx
time =  381.00 ns  ||  Register 0x09 = 0xxxxxxxxx
time =  382.00 ns  ||  Register 0x0a = 0xxxxxxxxx
time =  383.00 ns  ||  Register 0x0b = 0xxxxxxxxx
time =  384.00 ns  ||  Register 0x0c = 0xxxxxxxxx
time =  385.00 ns  ||  Register 0x0d = 0xxxxxxxxx
time =  386.00 ns  ||  Register 0x0e = 0x16968f0f
time =  387.00 ns  ||  Register 0x0f = 0x10010100
```

-R1 ends with 0 after being shifted multiple times
-R2 ends with 0 after being decremented in the loop
-R14 contains the result of the rotate right.
-R15 contains the pointer to the last memory location referenced+4

```
time =  412.00 ns  ||  dM[0c0] = 0xffffffff
time =  412.00 ns  ||  dM[0c4] = 0xfffffff0
time =  412.00 ns  ||  dM[0c8] = 0xffffffc0
time =  412.00 ns  ||  dM[0cc] = 0xffffff00
time =  412.00 ns  ||  dM[0d0] = 0xfffffc00
time =  412.00 ns  ||  dM[0d4] = 0xfffff000
time =  412.00 ns  ||  dM[0d8] = 0xffffc000
time =  412.00 ns  ||  dM[0dc] = 0xffff0000
time =  412.00 ns  ||  dM[0e0] = 0xfffc0000
time =  412.00 ns  ||  dM[0e4] = 0xfff00000
time =  412.00 ns  ||  dM[0e8] = 0xffc00000
time =  412.00 ns  ||  dM[0ec] = 0xff000000
time =  412.00 ns  ||  dM[0f0] = 0xfc000000
time =  412.00 ns  ||  dM[0f4] = 0xf0000000
time =  412.00 ns  ||  dM[0f8] = 0xc0000000
time =  412.00 ns  ||  dM[0fc] = 0x00000000
```

Each location, starting with 0x0C0, holds the result of each shift.
For example, 0x0C0 contains the first shift, 0x0C4 contains the second shift, and so on.

III. Verilog - Implementation

Memory Module 16 Memory Contents

```
@0
3c 01 ff ff
94 22 00 1f


90 43 00 00

00 60 20 2c


00 60 28 2d

00 00 00 0d
```

Load R1 with 0xFFFF0000

Clear 31st bit of R1 and store into R2

Set 0th bit of R2 and store into R3

Reverse the bits of R3 and store into R4

Reverse endianess of R4 and store into R5

Break

III. Verilog - Implementation

Memory Module 16 Results

```
BREAK INSTRUCTION FETCHED   35.00 ns
D I S P L A Y I N G   R E G I S T E R S
time =   37.00 ns  ||  Register 0x00 = 0x00000000
time =   38.00 ns  ||  Register 0x01 = 0xffff0000

time =   39.00 ns  ||  Register 0x02 = 0x7fff0000
time =   40.00 ns  ||  Register 0x03 = 0x7fff0001

time =   41.00 ns  ||  Register 0x04 = 0x8000fffe
time =   42.00 ns  ||  Register 0x05 = 0x0100ff7f
time =   43.00 ns  ||  Register 0x06 = 0xxxxxxxxx
time =   44.00 ns  ||  Register 0x07 = 0xxxxxxxxx
time =   45.00 ns  ||  Register 0x08 = 0xxxxxxxxx
time =   46.00 ns  ||  Register 0x09 = 0xxxxxxxxx
time =   47.00 ns  ||  Register 0x0a = 0xxxxxxxxx
time =   48.00 ns  ||  Register 0x0b = 0xxxxxxxxx
time =   49.00 ns  ||  Register 0x0c = 0xxxxxxxxx
time =   50.00 ns  ||  Register 0x0d = 0xxxxxxxxx
time =   51.00 ns  ||  Register 0x0e = 0xxxxxxxxx
time =   52.00 ns  ||  Register 0x0f = 0xxxxxxxxx
```

Initial Data

Cleared 31st bit of R1 and stored into R2

Set 0th bit of R2 and stored into R3

Reversed bit order of R3 stored into R4

Reversed endianness of R4 stored into R5

III. Verilog - Implementation

Memory Module 17 Memory Contents

```
@0
3d ce 32 54
35 ce ba dc
3c 0d 77 88
35 ad 77 88

7d ae 60 81
7d ac 58 c1

7d 8b 51 01
7d 6a 49 41

7d 49 41 81
7d 28 39 c1

7c e8 02 01
7c 00 32 81

7c 00 2a c1
7c c5 02 41

7c 00 22 81
7c 00 1a c1

00 00 00 0d
```

Load R14 with 0x3254BADC

Load R13 with 0x77887788

8bit add: R12 gets R13 + R14

8bit sub: R11 gets R13 - R12

8bit AND: R10 gets R12 & R11

8bit OR: R9 gets R11 | R10

8bit NOT: R8 gets ~R10

8bit XOR: R7 gets R9 ^ R8

8bit MUL: R12 gets R8 * R7

Move [63:32] to R6

Move [31:0] to R5

8bit DIV: R12 gets R6 / R5

Move [63:32] to R4

Move [31:0] to R3

Break

III. Verilog - Implementation

Memory Module 17 Results

```
BREAK INSTRUCTION FETCHED   97.00 ns
D I S P L A Y I N G   R E G I S T E R S
time =   99.00 ns  ||  Register 0x00 = 0x00000000
time =  100.00 ns  ||  Register 0x01 = 0xxxxxxxxx
time =  101.00 ns  ||  Register 0x02 = 0xxxxxxxxx
time =  102.00 ns  ||  Register 0x03 = 0x64000801
time =  103.00 ns  ||  Register 0x04 = 0x55002a03
time =  104.00 ns  ||  Register 0x05 = 0xb847da25
time =  105.00 ns  ||  Register 0x06 = 0x55ff642d
time =  106.00 ns  ||  Register 0x07 = 0xb9dfb9ff
time =  107.00 ns  ||  Register 0x08 = 0x7773ffdb
time =  108.00 ns  ||  Register 0x09 = 0xceac4624
time =  109.00 ns  ||  Register 0x0a = 0x888c0024
time =  110.00 ns  ||  Register 0x0b = 0xceac4624
time =  111.00 ns  ||  Register 0x0c = 0xa9dc3164
time =  112.00 ns  ||  Register 0x0d = 0x77887788
time =  113.00 ns  ||  Register 0x0e = 0x3254badc
time =  114.00 ns  ||  Register 0x0f = 0xxxxxxxxx
```

R3: result of 8bit DIV (lower)
R4: result of 8bit DIV (upper)
R5: result of 8bit MUL (lower)
R6: result of 8bit MUL (upper)
R7: result of 8bit XOR
R8: result of 8bit NOT
R9: result of 8bit OR
R10: result of 8bit AND
R11: result of 8bit sub
R12: result of 8bit add
R13: start value
R14: start value

III. Verilog - Implementation

Memory Module 18 Memory Contents

```
@0
3d ce 32 54
35 ce ba dc
3c 0d 77 88
35 ad 77 88

7d ae 60 82
7d ac 58 c2

7d 8b 51 02
7d 6a 49 42

7d 49 41 82
7d 28 39 c2

7c e8 02 02
7c 00 32 82

7c 00 2a c2
7c c5 02 42

7c 00 22 82
7c 00 1a c2

00 00 00 0d
```

Load R14 with 0x3254BADC

Load R13 with 0x77887788

16bit add: R12 gets R13 + R14

16bit sub: R11 gets R13 - R12

16bit AND: R10 gets R12 &

16bit OR: R9 gets R11 | R10

16bit NOT: R8 gets ~R10

16bit XOR: R7 gets R9 ^ R8

16bit MUL: R12 gets R8 * R7

Move [63:32] to R6

Move [31:0] to R5

16bit DIV: R12 gets R6 / R5

Move [63:32] to R4

Move [31:0] to R3

Break

III. Verilog - Implementation

<u>**Memory Module 18 Results**</u>

```
BREAK INSTRUCTION FETCHED   97.00 ns
D I S P L A Y I N G   R E G I S T E R S
time =   99.00 ns  ||  Register 0x00 = 0x00000000
time =  100.00 ns  ||  Register 0x01 = 0xxxxxxxxx
time =  101.00 ns  ||  Register 0x02 = 0xxxxxxxxx
time =  102.00 ns  ||  Register 0x03 = 0x2f2d0000
time =  103.00 ns  ||  Register 0x04 = 0x56ed0000
time =  104.00 ns  ||  Register 0x05 = 0xbae3f925
time =  105.00 ns  ||  Register 0x06 = 0x56ed2f2d
time =  106.00 ns  ||  Register 0x07 = 0xbbdfbaff
time =  107.00 ns  ||  Register 0x08 = 0x7673ffdb
time =  108.00 ns  ||  Register 0x09 = 0xcdac4524
time =  109.00 ns  ||  Register 0x0a = 0x898c0024
time =  110.00 ns  ||  Register 0x0b = 0xcdac4524
time =  111.00 ns  ||  Register 0x0c = 0xa9dc3264
time =  112.00 ns  ||  Register 0x0d = 0x77887788
time =  113.00 ns  ||  Register 0x0e = 0x3254badc
time =  114.00 ns  ||  Register 0x0f = 0xxxxxxxxx
```

R3: result of 16bit DIV (lower)
R4: result of 16bit DIV (upper)
R5: result of 16bit MUL (lower)
R6: result of 16bit MUL (upper)
R7: result of 16bit XOR
R8: result of 16bit NOT
R9: result of 16bit OR
R10: result of 16bit AND
R11: result of 16bit sub
R12: result of 16bit add
R13: start value
R14: start value

Memory Module 19 Memory Contents

```
@0
3c 01 12 34
34 21 56 78
3c 02 87 65
34 42 43 21
```

R1 <- 0x1234_5678
R2<- 0x8765_4321

```
00 41 00 1a
40 00 00 01
44 00 00 03
```

1: Divide R1 by R2
2: Branch if divide by zero flag is true. (fail_1)
3: Branch if divide by zero flag is false (pass_1)

```
3c 0e ff ff fail_1:
35 ce ff ff
00 00 00 0d
```

R14 <- 0xFFFF_FFFF (fail flag), then break.

```
00 00 70 20 pass_1:
```

R14<- 0 (pass flag)

```
00 40 00 1a
44 00 00 01
40 00 00 03
```

1: Divide R1 by R2
2: Branch if divide by zero flag is false(fail_2).
3: Branch if divide by zero flag is true (pass_2).

```
3c 0f ff ff fail_2:
35 ef ff ff
00 00 00 0d
```

R15<-0xFFFF_FFFF(fail flag), then break.

```
00 00 78 20 pass_2:
00 00 00 0d
```

R15 <- 0 (pass flag), then break.

Memory Module 19 Results

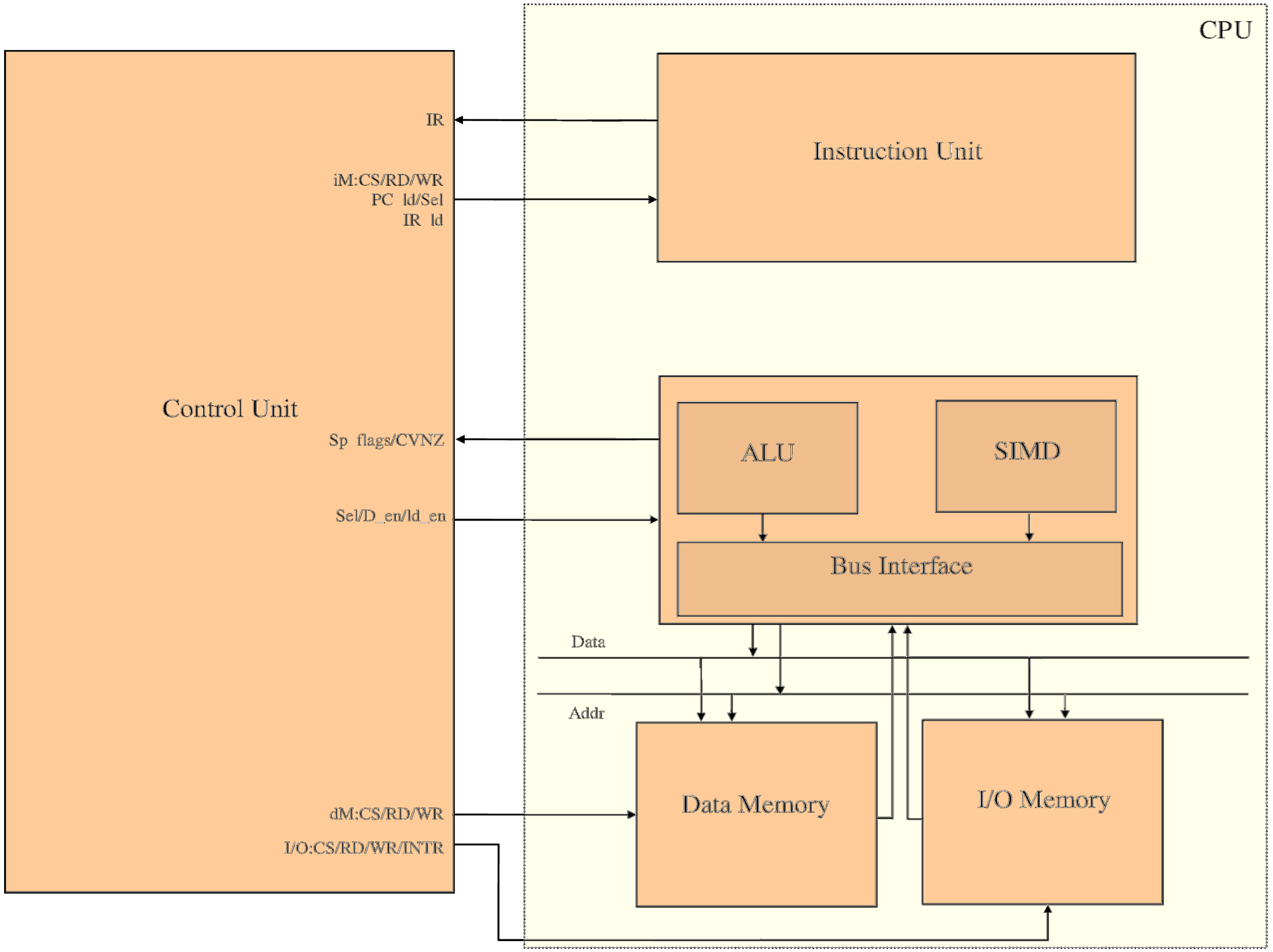
```
BREAK INSTRUCTION FETCHED   56.50 ns
D I S P L A Y I N G   R E G I S T E R S
time =   58.10 ns  ||  Register 0x0 = 0x00000000
time =   58.20 ns  ||  Register 0x1 = 0x12345678
time =   58.30 ns  ||  Register 0x2 = 0x87654321
time =   58.40 ns  ||  Register 0x3 = 0xxxxxxxxx
time =   58.50 ns  ||  Register 0x4 = 0xxxxxxxxx
time =   58.60 ns  ||  Register 0x5 = 0xxxxxxxxx
time =   58.70 ns  ||  Register 0x6 = 0xxxxxxxxx
time =   58.80 ns  ||  Register 0x7 = 0xxxxxxxxx
time =   58.90 ns  ||  Register 0x8 = 0xxxxxxxxx
time =   59.00 ns  ||  Register 0x9 = 0xxxxxxxxx
time =   59.10 ns  ||  Register 0xa = 0xxxxxxxxx
time =   59.20 ns  ||  Register 0xb = 0xxxxxxxxx
time =   59.30 ns  ||  Register 0xc = 0xxxxxxxxx
time =   59.40 ns  ||  Register 0xd = 0xxxxxxxxx
time =   59.50 ns  ||  Register 0xe = 0x00000000
time =   59.60 ns  ||  Register 0xf = 0x00000000
```

R1 and R2 loaded with specified values.

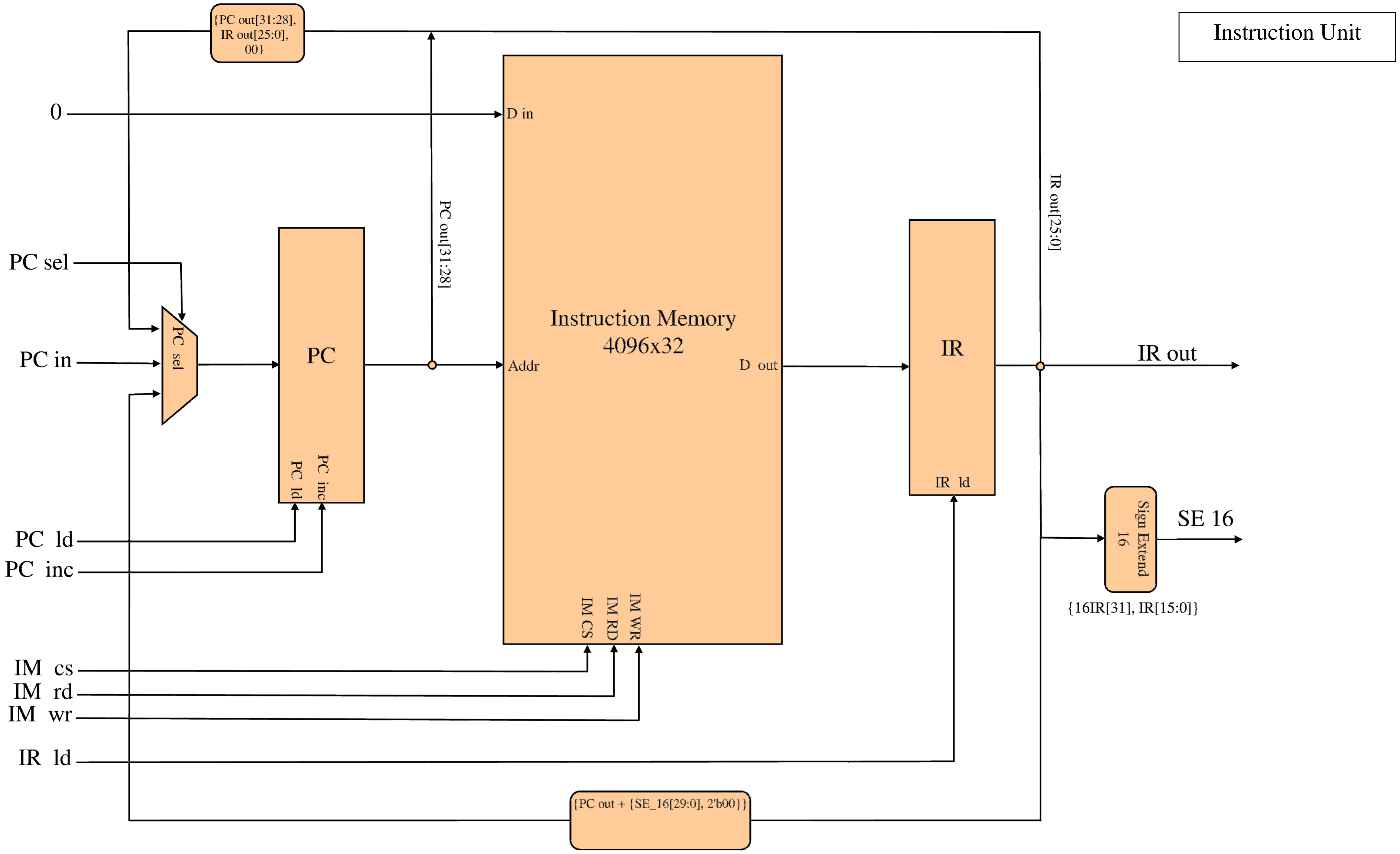
Pass flags for R14 and R15.

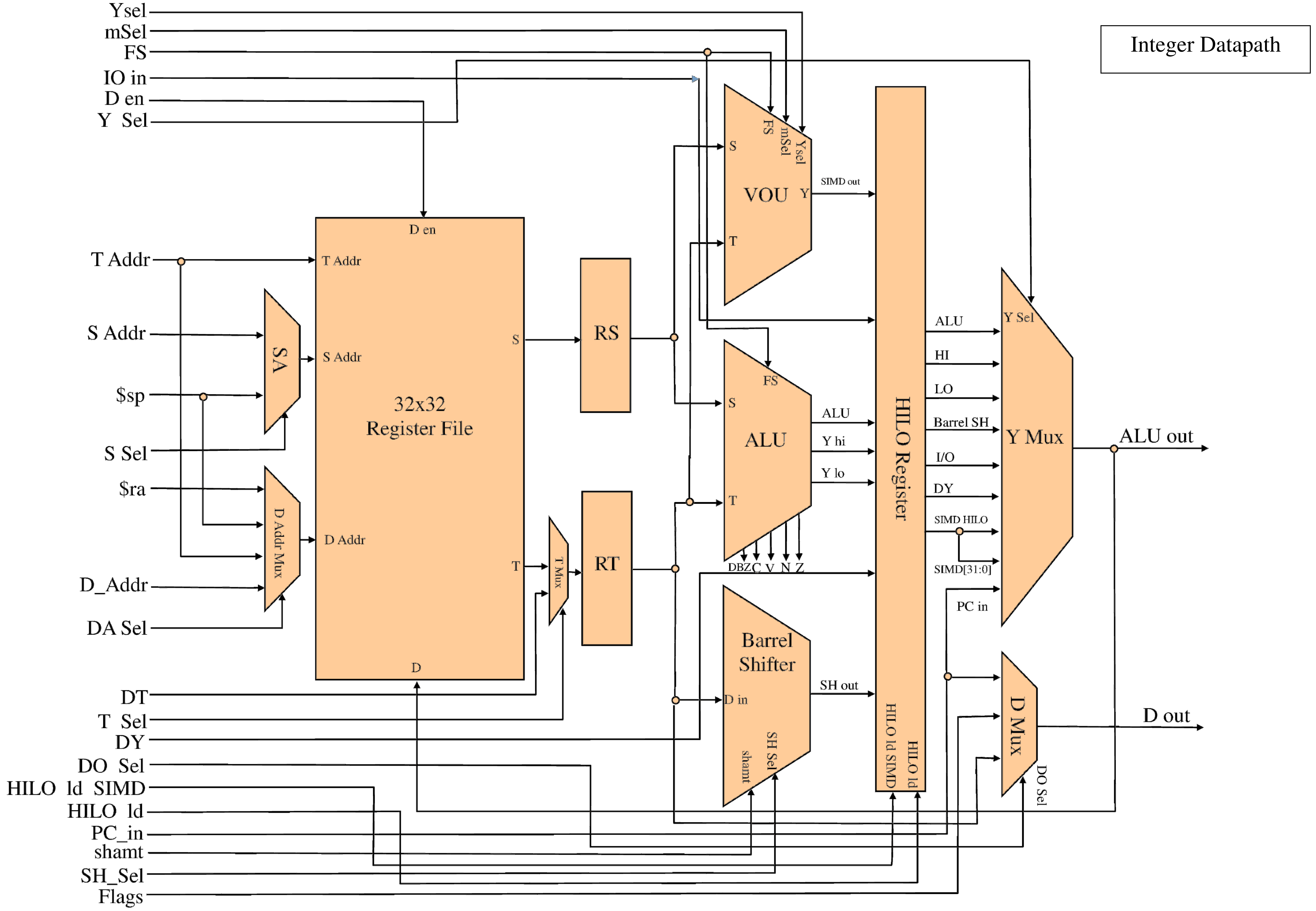
IV. Hardware - Implementation

Hardware Implementation

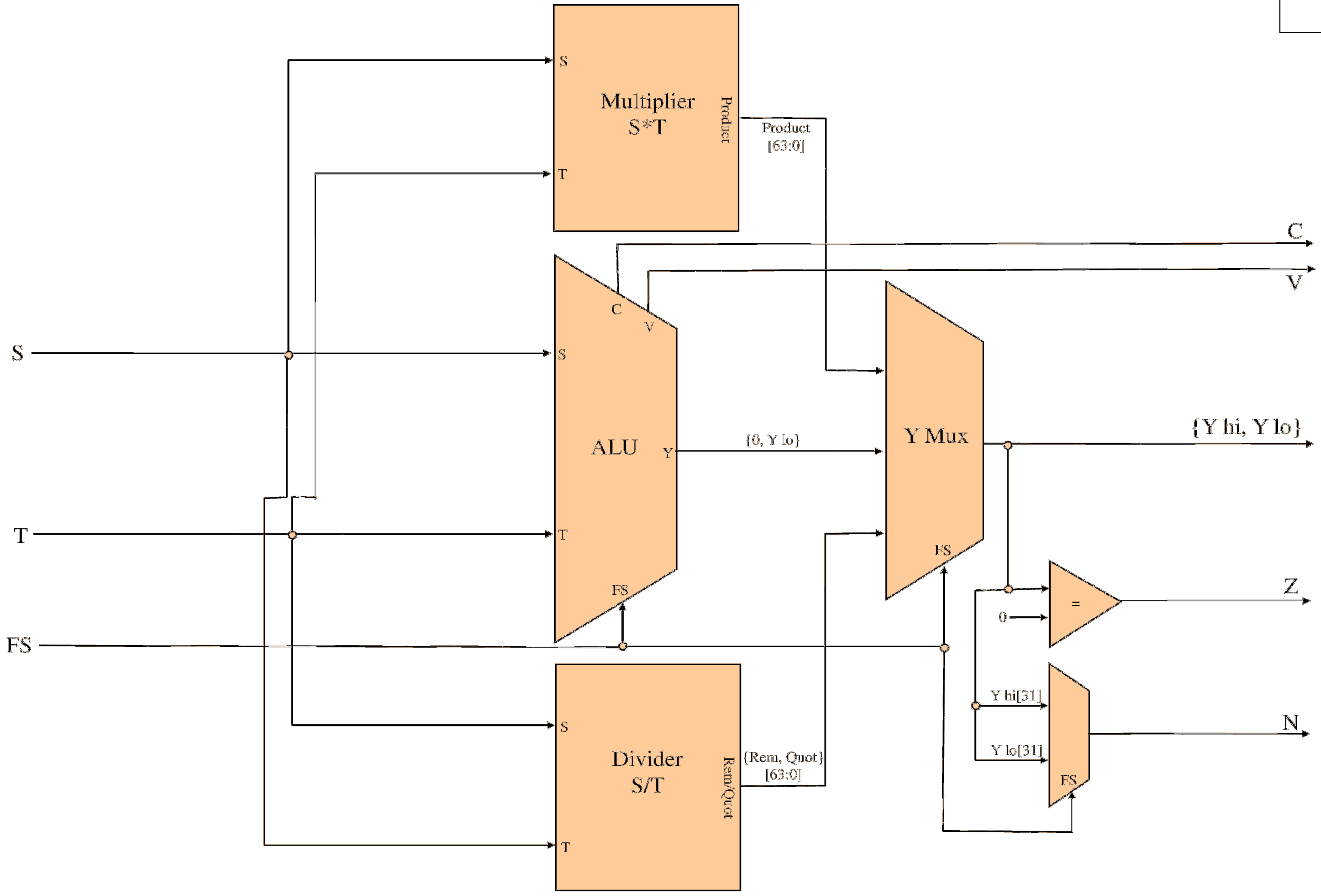


Control Unit/CPU Block


Instruction Unit
{PC out[31:28], IR out[25:0], 00}
0
D in
PC out[31:28]
IR out[25:0]
PC sel
PC in
PC sel
PC
Instruction Memory 4096x32
Addr
D out
IR
IR out
PC ld
PC inc
IR ld
Sign Extend 16
SE 16
{16IR[31], IR[15:0]}
PC ld
PC inc
IM CS
IM RD
IM WR
IM cs
IM rd
IM wr
IR ld
{PC out + {SE_16[29:0], 2'b00}}

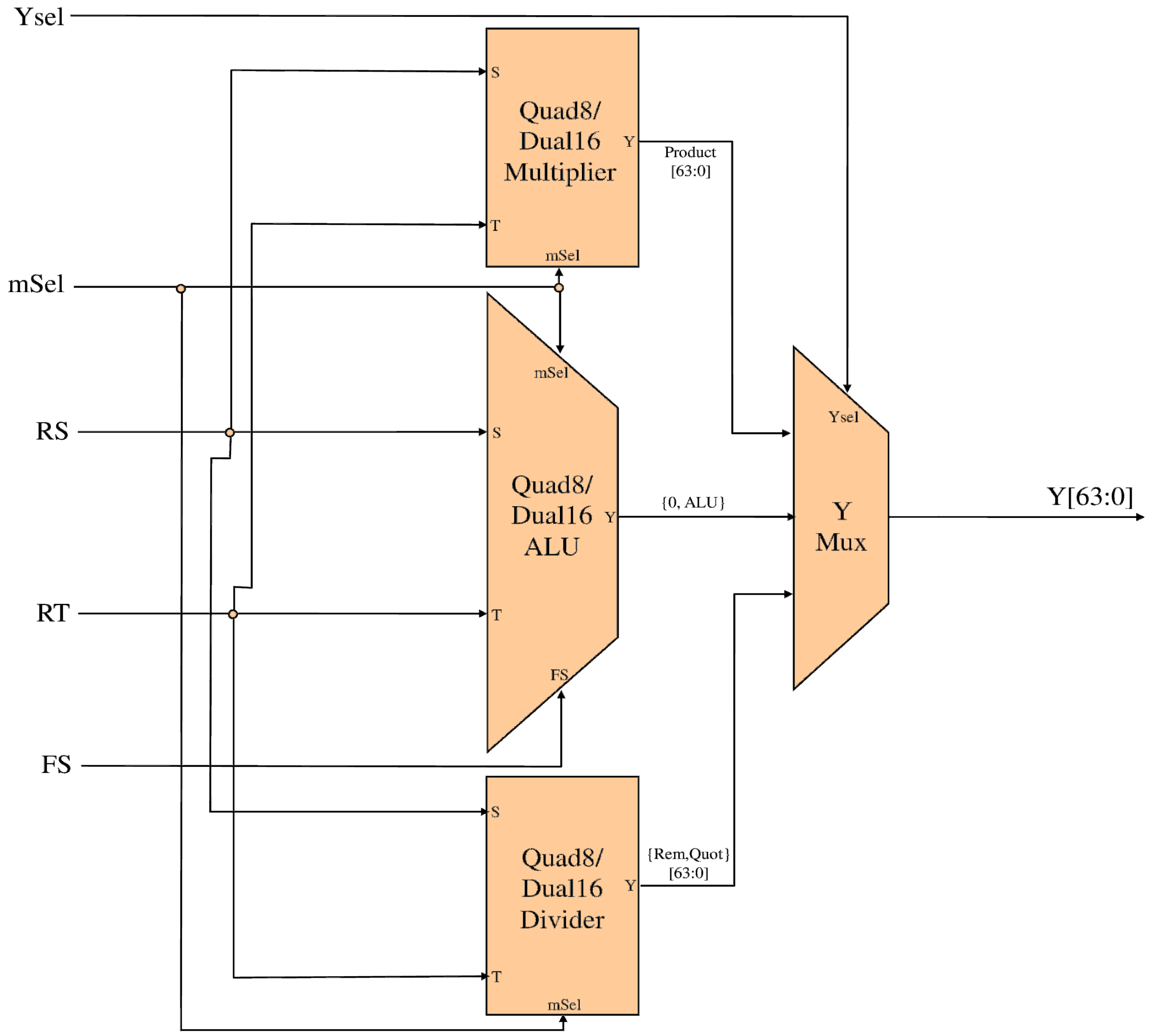

Integer Datapath
Ysel
mSel
FS
IO in
D en
Y Sel
T Addr
S Addr
$sp
S Sel
$ra
D_Addr
DA Sel
DT
T Sel
DY
DO Sel
HILO ld SIMD
HILO ld
PC_in
shamt
SH_Sel
Flags
SA
D Addr Mux
D en
T Addr
S Addr
D Addr
32x32
Register File
S
T
D
T Mux
RS
RT
VOU
FS
mSel
Ysel
S
T
Y
SIMD out
ALU
FS
S
T
ALU
Y hi
Y lo
DBZ C V N Z
Barrel
Shifter
D in
SH out
SH Sel
shamt
HILO Register
HILO ld SIMD
HILO ld
Y Mux
Y Sel
ALU
HI
LO
Barrel SH
I/O
DY
SIMD HILO
SIMD[31:0]
PC in
ALU out
D Mux
DO Sel
D out

Main ALU



Multiplier
S*T
S
T
Product
Product
[63:0]
C
V
S
T
FS
ALU
Y
C
V
{0, Y lo}
Y Mux
FS
{Y hi, Y lo}
Z
=
0
Y hi[31]
Y lo[31]
FS
N
Divider
S/T
S
T
Rem/Quot
{Rem, Quot}
[63:0]

Vector ALU


Ysel
mSel
RS
RT
FS
S
Quad8/
Dual16
Multiplier
Y
T
mSel
Product
[63:0]
mSel
S
Quad8/
Dual16
ALU
Y
T
FS
{0, ALU}
Ysel
Y
Mux
Y[63:0]
S
Quad8/
Dual16
Divider
Y
T
mSel
{Rem,Quot}
[63:0]