TR MIPS32 Enhanced SIMD Architecture for Designers

Instruction Set Architecture Reference

Revision 2.1
April 25, 2019

Reed Ellison
015290260
Thomas Nguyen
016238935

CECS 440
April 23, 2019

Table of Contents

I. Purpose

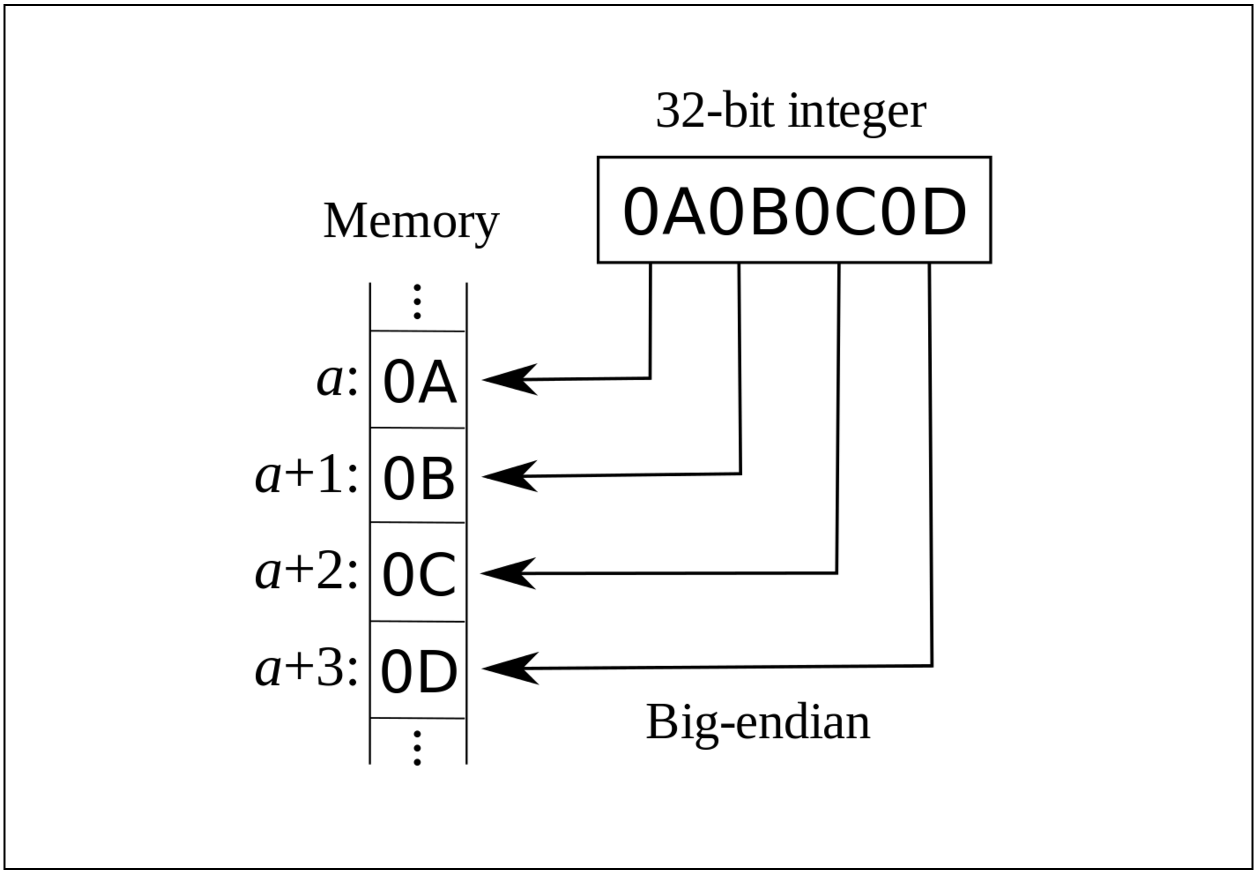
The purpose of this reference manual is to detail the operation and functions of the Instruction Set Architecture of the MIPS SIMD Processor. Within this document, the design of the processor, including its registers, functions, memory structure, instruction set, data types, and data paths. As a modified MIPS processor, most of the design choices are based on the general MIPS processor including most of the instructions, registers, and data/control paths and then the additional SIMD vector instructions and data. Within this processor are 32 general purpose 32-bit registers for storing and holding data. Two external memories can also be accessed through various Addressing Modes which are used to take in additional data. One external memory is for data registers and the other is for I/O memory. Through those various Addressing Modes, the MIPS SIMD processor will be able to handle input and output by using interrupt service routine (ISR) and loading and storing to the I/O memory. The MIPS32 SIMD will be able to handle vector based instructions and data. The types of vector it utilize are 4-byte and 2-word vectors. The MIPS SIMD will be able to use and switch between standard 32-bits and then the 8-bits and 16-bits modes at any time.

II. Instruction Set Architecture

A. Harvard Memory Architecture and Organization

The MIPS uses the Harvard memory architecture that has separate 32-bit address instructions and data memories. Each memory are “byte addressable” where each of the memory location holds one byte. The 32-bit address space holds a minimum of 1kB words from 0h to 3FFh for running code simulations. The 32-bit memory operands are stored in four consecutive memory locations in a “big endian” format. This format means that the most significant 8-bits are stored in a lower address and the least significant 8-bits are stored at a higher address.

Figure 1.1 Big Endian Format



B: Processor Register Set

These are the 32 available general purpose 32-bit registers. 6 of the 32 registers have restricted and special use and are not to be freely modified in a regular instruction. The remaining registers are free to be use in any manner.

Number	Name	Description
0	$zero	Constant 0 (Hardware)
1	$at	Reserved for assembler
2 - 3	$v0 - $v1	Expression evaluation and function results
4 - 7	$a0 - $a3	Arguments
8 - 15	$t0 - $t7	Temporary
16 - 23	$s0 - $s7	Saved on subroutines
24 - 25	$t8 - $t9	Temporary (continued)
26 - 27	$k0 - $k1	Reserved for OS kernel
28	$gp	Pointer to global area
29	$sp	Stack pointer
30	$fp	Frame pointer
31	$ra	Return address (Hardware)

1. Reserved Special Registers

$zero: This register has a constant 0 and cannot be modified.

$at: This is used by the assembler for expanding pseudocode instructions.

$gp: Points to a global area and should not be commonly used.

$sp: Points to the top of the stack

$fp: Points to the bottom of the stack frame.

$ra: Stores the address of the instruction after a CALL.

2. The Program Counter

The 32-bit program counter (PC) contains the current address of the next instruction to be fetched. Every time an instruction is executed, the PC will increment by 4. The reason for this increment is because the MIPS is 32-bits and all addresses are by a multiple of 4. The PC can be modified through I-Type and J-Type instructions.

3. Status Flags Register

The 5-bit status flag register holds the status of the current program. Instructions that have an affect on the status flags will modify this register and set or clear one or more of the bits inside the register.

Figure 1.2 Status Flag Register

5				0
IE	C	V	N	Z

The status flag register consists of the following: Interrupt Enable (IE), Carry (C), Overflow (V), Negative (N) and Zero (Z) flags.

- The IE flag can be set when the user wants there to be interrupts in their program
- The C flag will only be set when an instruction's result has a carry-out.
- The V flag will only be set when the results of an instruction exceeded the 32-bit and cannot be expressed correctly do to this overflow.
- The N flag will only be set when the results is negative and never during unsigned operations.
- The Z flag will be set when the results of an instruction is zero.

C. Data Types

The 32-bit MIPS processor has two data types. It uses signed and unsigned integers. The values are all 32-bit wide. No instruction bits exceed 32-bits. Results are all 32-bits with the exception of MULT and DIV instructions in which their 64-bit results are split where it has an upper and lower 32-bit value. The upper bits are stored in a 32-bit register HI while the lower bits are stored in a 32-bit register LO.

Every instruction uses signed integers except for ADDU, SLTU, SUBU and SLTIU. Unsigned instructions do not update the N status flag.

An additional vector type mode allows the 32-bit MIPS to utilize two extra data types of vector 4-bytes and vector 2-word. MULT and DIV instruction work in a similar way. For a vector 4-byte, MULT8 will result in a product two-word values. DIV8 will result in a vector of 4-byte quotient and remainder where the values are stored in the same respective HI and LO registers. MULT16 will result in a product of 32-bit or doubleword. DIV16 will work in the same way as DIV8 but being 2-word vectors.

D. Addressing Modes

The MIPS contains three addressing modes that affect the instructions that it contains. The three types of addressing modes are Immediate, Register and Register Indirect.

1. Immediate Address Mode (16-bit and 26-bit)

This addressing mode is used by I-Type and J-Type instructions. The immediate values for I-Types are 16-bit signed values located at the lower 16-bits of the instruction. The immediate values for J-Types are a 26-bit signed value located at the lower 26-bits of the instruction. The immediate offsets value can be used to add or use a number to modify a value inside a register or do jumps and branches.

Unlike other instructions, instruction types of the branch and jump variant have target addresses that are word aligned.

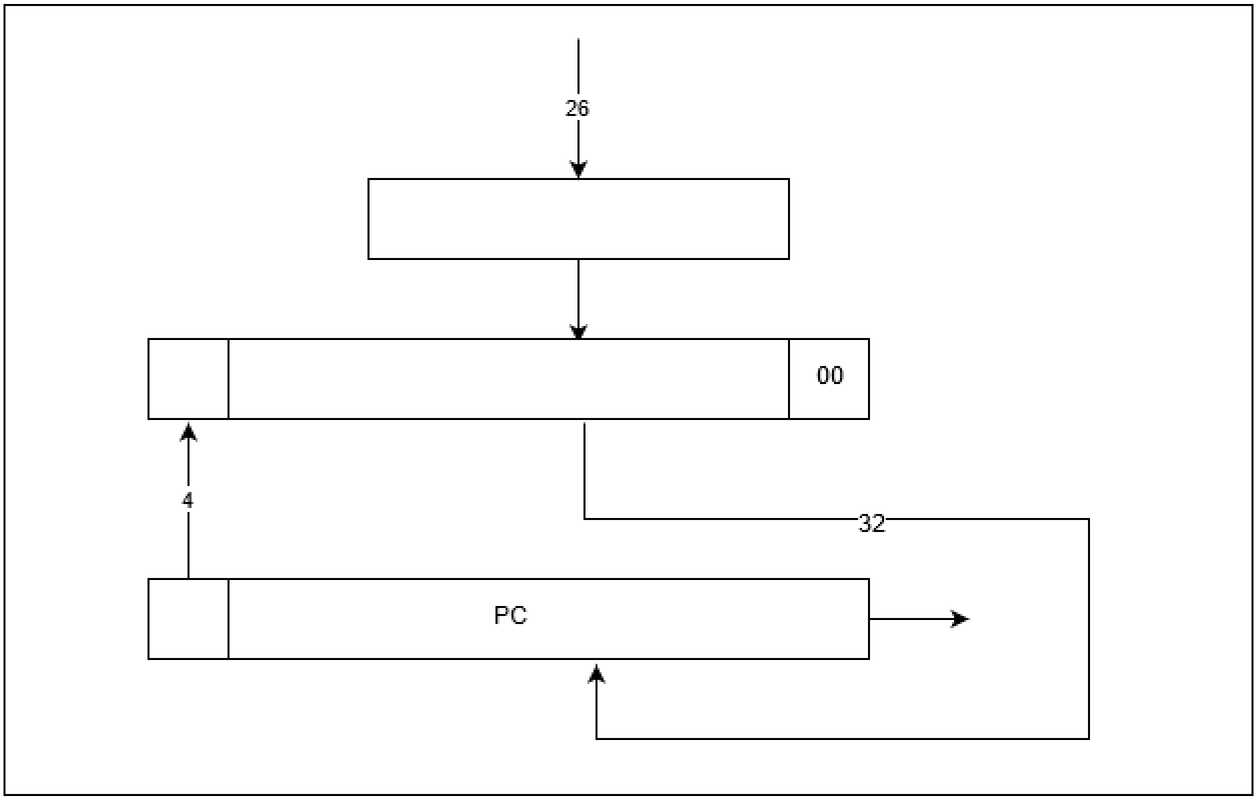
Branch instructions take their 16-bit immediate field and sign-extend the MSB to create a 32-bit while shifting to the left by 2.

The jump instructions’ 26-bit immediate field has an absolute address formed by concatenating the first two low-order bits to create a 28-bit address and concatenating the upper 4 bits of the current PC to create the 32-bit address. See Figure 2.1 Jump Concatenating

An example of an immediate address would be ADDI

$rt ← $rs + imm16

Figure 2.1 Jump Concatenating



2. Register Address Mode

The most commonly use addressing mode. The R-Type instruction use this addressing mode to do their tasks. Most of the arithmetic instructions use this mode as well. This takes two source registers or operands and modifies the register and then stores the result into a destination register rd. An example would be the ADD instruction.

$rd ← $rs + $rt

3. Register Indirect Address Mode

Register Indirect mode is only used by instructions load, store and jump. The instructions access memory from the location of an immediate offset value + address value at the source register rs.

- Load will use the memory at that location and load it into the destination register rt.
- Store will take the address value at the destination register rt and store it into the memory location.
- Jump will take the memory at that location and add it to the PC.

E. Instruction Set Abbreviations and Format Descriptions

Encoded constant and variable field for the R-Type are in order of the:
Op code, source register “rs”, source register “rt”, destination register “rd”, shift amount (shamt), and function code.

The Figure 2.2 shows an example of an instruction description for R-Type Instructions.

Figure 2.2 Instruction Description Example (R-Type Format)

Example Instruction Name

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

Format: EXAMPLE rd, rs, rt
This is the Assembler format for the instruction.

Purpose: Example Instruction Name
Short description of what the instruction does.

Description: rd ← rs Example_Op rt
Describes the operation of the instruction in detail by explaining what is happening in the table. It goes into further depth that would be impossible to detail on the operations page.

Restrictions:
Details what limitation the instructions has. This can range from values on the instruction fields, the registers it uses or can’t use, operand values, operand formats and access to memory addresses.

Example:
This section shows how the instruction is used in an example and what effects it has after it is executed.

Encoding fields that are compiled with hexadecimal values in order for the machine to read.

Abbreviation	**Meaning**
rs	Source Register/Operand 1
rt	Source Register/Operand 2/Destination for I-Type
rd	Destination Register
shamt	Shift Amount
fs	Function Select
opc	Opcode
Immediate	16-bit Offset Value
26-bit ADDRESS	PC + (20-bit Offset Value)

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

R-Type

31 26	25 21	20 16	15 0
opc	rs	rt	Immediate

I-Type

31 26	25 0
opc	26-bit ADDRESS

J-Type

31 26	25 21	20 16	15 11	10 6	5 0
0x2C	rs	rt	rd	shamt	fs

Vector-Type 8-bits

31 26	25 21	20 16	15 11	10 6	5 0
0x2C	rs	rt	rd	shamt	fs

Vector-Type 16-bits

1. Format Field Meaning

This is the assembler format for a specific instruction. This is for user use when they want to do a specific instruction. This makes it easier for the user to understand what they are trying to do as the machine sees only in binary.

Figure 2.2 Format Field Example (DIV)

Format: div rs, rt

2. Purpose Field Meaning

This field gives the full name of the instruction and a short description of what it does.

Figure 2.3 Purpose Field Example (DIV)

Purpose: Division
Divides two 32-bit registers and stores the results into two other 32-bit registers named *hi* and *lo*.

3. Description Field Meaning

This field gives a more in depth description of what the instruction does in case a one line description does not explain enough of what the instruction does. This also tells how the instruction is used and what it does with the registers and operands.

Figure 2.4 Description Field Example (DIV)

Description: rs ÷ rt
lo ← quotient
hi ← remainder
The 32-bit register rs is the numerator and the 32-bit register rt is the denominator. They are divided and results in a 64 bit value that is split into two parts: the quotient and the remainder. The quotient is loaded into the 32-bit register *lo* and the remainder is loaded into the 32-bit register *hi*. Status flags (C, V, N, Z) are checked and updated depending on results. The negative flag is dependent on the quotient or lower 32-bits.

The description starts by showing the RTL code and then is followed by a body of text. Inside the body of text explains what is going on in the instruction and at the end it will tell the changes to specific status flags are specified in this field if they are necessary.

4. Restriction Field Meaning

This field tells what limitations or restrictions an instruction has. Some instructions are incapable of doing specific things and others have no limitations.

Restrictions have specific categories:

- Valid values for the instruction fields
- Valid values for the operands
- Valid operand formats
- Valid memory access types

Figure 2.5 Restriction Field Example (DIV)

Restrictions:
If the register rt contains a value of 0, the results will be undefined.

5. Example Field Meaning

This field shows exactly how a specific instruction is used and the effects it has on the register it modifies.

Figure 2.6 Example Field Example (DIV)

Assembly	Machine Code
`div r1, r5`	`000000_00001_00101_00000_00000_011010`
`div r7, r4`	`000000_00111_00001_00000_00000_011010`

Operand Values	Result

`r1 = 0x0000020D`
`r5 = 0xFFFFFFE3`
	`hi = 0x00000003`
	`lo = 0xFFFFFFEE`
	`c=x, v=x, n=1, z=0`

`r7 = 0xFFFFFF9D`
`r4 = 0xFFFFFF9C`
	`hi = 0xFFFFFF9D`
	`lo = 0x00000000`
	`c=x, v=x, n=0, z=1`

6. Vector Type Operations

Vector-Type consists of two types: 8-bit and 16-bit. This type of operations makes use of unsigned integers. The 32-bit register represents the vector of either the 4-bytes or 2-words. When doing operations, each byte or word of one operand will only do the operation with the same corresponding index byte or word of the other operand. A different case happens when doing DIV/MULT instructions. Upper 16-bits operations are loaded into the HI register and lower 16-bits are loaded into the LO register. The process can be seen below.

4-Byte Vector Operation

Operand 1

31 25	21 16	15 8	7 0
A1	A2	A3	A4

Operand 2

31 25	21 16	15 8	7 0
B1	B2	B3	B4

Destination

31 25	21 16	15 8	7 0
A1 op B1	A2 op B2	A3 op B3	A4 op B4

2-Word Vector Operation

Operand 1

31 16	15 0
A1	A2

Operand 2

31 16	15 0
B1	B2

Destination

31 16	15 0
A1 op B1	A2 op B2

4-Byte Vector Operation For Multiply

Operand 1

31 25	21 16	15 8	7 0
A1	A2	A3	A4

Operand 2

31 25	21 16	15 8	7 0
B1	B2	B3	B4

HI Register

31 16	15 0
Product1	Product2

LO Register

31 16	15 0
Product3	Product4

2-Word Vector Operation For Multiply

Operand 1

31 16	15 0
A1	A2

Operand 2

31 16	15 0
B1	B2

HI Register

31 0
Product1

LO Register

31 0
Product2

4-Byte Vector Operation For Division

Operand 1

31 25	21 16	15 8	7 0
A1	A2	A3	A4

Operand 2

31 25	21 16	15 8	7 0
B1	B2	B3	B4

HI Register

31 25	21 16	15 8	7 0
Remainder1	Remainder2	Remaind3	Remainder4

LO Register

31 25	21 16	15 8	7 0
Quotient1	Quotient2	Quotient3	Quotient4

2-Word Vector Operation For Division

Operand 1

31 16	15 0
A1	A2

Operand 2

31 16	15 0
B1	B2

HI Register

31 16	15 0
Remainder1	Remainder2

LO Register

31 16	15 0
Quotient1	Quotient2

F. Instruction Set and Binary Instruction Formats

This section provides details on understanding the constructs of the instruction descriptions found for the instruction set format.

1. Instruction Set Summary List by Category

The next sections list MIPS instructions grouped by category type and provide additional information about how each instruction work and how they are used according to their type in alphabetical order.

The MIPS instruction set contains the following types of instructions: R-Type (Table 3.1), I-Type (Table 3.2), J-Type (Table 3.3), Enhanced (Table 3.4).

Table 3.1 MIPS R-Type Instructions

Name	Instruction Description
add	Addition (Signed)
addu	Addition (Unsigned)
and	Logical AND
break	Break
clr	Clear
div	Division
jr	Jump Register
mfhi	Move from Hi
mflo	Move from Lo
mov	Move
mult	Multiply
nop	No Operation
nor	Logical NOT OR
or	Logical OR
rotl	Rotate Left

Table 3.1 MIPS R-Type Instructions (Continued)

rotr	Rotate Right
setie	Set Interrupt Enable
sll	Shift Left Logical
slt	Set Less Than
sltu	Set Less Than (Unsigned)
sra	Shift Right Arithmetic
srl	Shift Right Logical
sub	Subtraction (Signed)
subu	Subtraction (Unsigned)
xor	Logical Exclusive OR

Table 3.2 MIPS I-Type Instructions

Name	Instruction Description
addi	ADD Immediate
andi	AND Immediate
beq	Branch if Equal
blez	Branch if Less than or Equal to Zero
bgtz	Branch if Greater than Zero
bne	Branch if Not Equal
lw	Load Word
lui	Load Upper Immediate
ori	OR Immediate
slti	Set Less than Immediate
sltiu	Set Less than Immediate (Unsigned)
sw	Store Word
xori	XOR Immediate

Table 3.3 MIPS J-Type Instructions

Name	Instruction Description
j	Jump
jal	Jump and Link

Table 3.4 MIPS Enhanced Instructions

Name	Instruction Description
input	Load Register from I/O
output	Store Register to I/O
pop	Load from stack pointer
push	Store to stack pointer
reti	Return from Interrupt
simd8	8-bit Vector Type
simd16	16-bit Vector Type
“e_key”	Reserved

Table 3.5 MIPS Vector-Type Instructions

Name	Instruction Description
add8	Vector Addition of 8-bit values
add16	Vector Addition of 16-bit values
and8	Vector Logical AND of 8-bit values
and16	Vector Logical AND of 16-bit values
div8	Vector Division of 8-bit values
div16	Vector Division of 16-bit values
mult8	Vector Multiply of 8-bit values
mult16	Vector Multiply of 16-bit values
nor8	Vector Logical Not OR of 8-bit values

Table 3.5 MIPS Vector-Type Instructions (Continued)

Name	Instruction Description
nor16	Vector Logical Not OR of 16-bit values
or8	Vector Logical OR of 8-bit values
or16	Vector Logical OR of 16-bit values
rotl8	Vector Rotate Left of 8-bit values
rotl16	Vector Rotate Left of 16-bit values
rotr8	Vector Rotate Right of 8-bit values
rotr16	Vector Rotate Right of 16-bit values
sll8	Vector Shift Left Logical of 8-bit values
sll16	Vector Shift Left Logical of 16-bit values
sra8	Vector Shift Right Arithmetic of 8-bit values
sra16	Vector Shift Right Arithmetic of 16-bit values
srl8	Vector Shift Right Logical of 8-bit values
srl16	Vector Shift Right Logical of 16-bit values
sub8	Vector Subtraction of 8-bit values
sub16	Vector Subtraction of 16-bit values
xor8	Vector Logical Exclusive OR of 8-bit values
xor16	Vector Logical Exclusive OR of 16-bit values

2. Instruction Set R-Type List

ADD

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

Format: add rd, rs, rt

Purpose: Signed Addition
Signed addition of two registers and storing result in the destination register

Description: rd ← rs + rt
The 32-bit source registers rs and rt are added together and their results are stored in the 32-bit destination register rd. Status flags (C, V, N, Z) are checked and updated depending on results.

Restrictions:
None

Example:

Assembly	Machine Code
`add r4, r1, r5`	`000000_00001_00101_00100_00000_100000`
`add r8, r7, r4`	`000000_00111_00001_01000_00000_100000`

Operand Values	Result

`r1 = 0x0000020D`
`r5 = 0xFFFFFFE3`
	`r4 = 0x000001F0`
	`c=1, v=0, n=0, z=0`

`r7 = 0xFFFFFF9D`
`r4 = 0xFFFFFF9C`
	`r8 = 0xFFFFFF39`
	`c=1, v=0, n=1, z=0`

ADDU

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

Format: addu rd, rs, rt

Purpose: Unsigned Addition
Unsigned addition of two registers and storing result in the destination register

Description: rd ← rs + rt
The 32-bit source registers rs and rt are added together and their results are stored in the 32-bit destination register rd. Status flags (C, V and Z) are checked and updated depending on results. Negative status flag is unaffected by this instruction.

Restrictions:
rs and rt must be unsigned values

Example:

Assembly	Machine Code
`addu r4, r1, r5`	`000000_00001_00101_00100_00000_100001`
`addu r8, r7, r4`	`000000_00111_00001_01000_00000_100001`

Operand Values	Result

`r1 = 0x0000020D`
`r5 = 0xFFFFFFE3`
	`r4 = 0x000001F0`
	`c=1, v=1, n=0, z=0`

`r7 = 0xFFFFFF9D`
`r4 = 0xFFFFFF9C`
	`r8 = 0xFFFFFF39`
	`c=1, v=1, n=0, z=0`

AND

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

Format: and rd, rs, rt

Purpose: Logical AND
Do a bitwise AND to two 32-bit and stores result into the destination 32-bit register

Description: rd ← rs & rt
Doing a Logical AND, each bits of two 32-bit registers will either become 1s if both bits are 1s and 0s if they don't. The results of the rt and rs are then stored into the 32-bit register rd. Status flags (N and Z) are checked and updated depending on results.

Restrictions:
None

Example:

Assembly	Machine Code
`and r9, r1, r5`	`000000_00001_00101_01001_00000_100100`
`and r12, r7, r4`	`000000_00111_00001_01100_00000_100100`

Operand Values	Result

`r1 = 0xF0F03C3C`
`r5 = 0xBF0FF5F5`
	`r9 = 0xB0003434`
	`c=x, v=x, n=1, z=0`

`r7 = 0xA7F8F59D`
`r4 = 0x41049CCC`
	`r9 = 0x0100948C`
	`c=x, v=x, n=0, z=0`

BREAK

31 26	25 21	20 16	15 11	10 6	5 0
0x00	0	0	0	0	0x0D

Format: break

Purpose: Break
Terminates program and displays the contents of the registers and memory

Description:
This is used to terminate the current program and displays the contents of the registers and memory.

Restrictions:
None

Example:

Assembly	Machine Code
`Break`	`000000_00000_00000_00000_00000_001101`

CLR

31 26	25 21	20 16	15 11	10 6	5 0
0x00	0	rt	0	0	0x06

Format: clear

Purpose: Clear
Clears content of register

Description: rt ← 32’h0
This clears the content of the register by setting contents of the register to 0.

Restrictions:
None

Example:

Assembly	Machine Code
`clr r2`	`000000_00000_00010_00000_00000_000110`

Operand Values	Result

`r2 = 0xFFF5FFFF`	`r2 = 0x00000000`
	`c=x, v=x, n=x, z=x`

DIV

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

Format: div rs, rt

Purpose: Division
Divides two 32-bit registers and stores the results into two other 32-bit registers named *hi* and *lo*.

Description: rs ÷ rt
lo ← quotient
hi ← remainder
The 32-bit register rs is the numerator and the 32-bit register rt is the denominator. They are divided and results in a 64 bit value that is split into two parts: the quotient and the remainder. The quotient is loaded into the 32-bit register *lo* and the remainder is loaded into the 32-bit register *hi*. Status flags (N, Z) are checked and updated depending on results. The negative flag is dependent on the quotient or lower 32-bits.

Restrictions:
If the register rt contains a value of 0, the results will be undefined.

Example:

Assembly	Machine Code
`div r1, r5`	`000000_00001_00101_00000_00000_011010`
`div r7, r4`	`000000_00111_00001_00000_00000_011010`

Operand Values	Result

`r1 = 0x0000020D`
`r5 = 0xFFFFFFE3`
	`hi = 0x00000003`
	`lo = 0xFFFFFFEE`
	`c=x, v=x, n=1, z=0`

`r7 = 0xFFFFFF9D`
`r4 = 0xFFFFFF9C`
	`hi = 0xFFFFFF9D`
	`lo = 0x00000000`
	`c=x, v=x, n=0, z=1`

Operand 1: R1

31	0
0x0000020D	

Operand 2: R5

31	0
0xFFFFFFE3	

HI Register: Remainder

31	0
0x00000003	

LO Register: Quotient

31	0
0xFFFFFFEE	

JR

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

Format: div rs

Purpose: Jump Register
Unconditional jump to address stored in register rs

Description: PC ← rs
The value stored inside the 32-bit register rs is loaded into PC and then begins executing instructions at the new memory address. No status flags are updated.

Restrictions:
MIPS addresses increment/decrement by 4, so the lower bytes must be in multiples of 4 for the jump to work.

Example:

Assembly	Machine Code
`jr r9`	`000000_01001_00000_00000_00000_001000`
`jr r7`	`000000_00111_00000_00000_00000_001000`

Operand Values	Result
`r9 = 0xF0F03C3C`	`PC = 0xF0F03C3C`
`r7 = 0xA7F8F59D`	`PC = 0xA7f8F59C`

MFHI

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

Format: mfhi rd

Purpose: Move from Hi
Loads the stored value inside *hi* from after doing a MULT or DIV instruction into destination register rd.

Description: rd ← *hi*
The 32-bit register rd obtains the value stored inside the register *hi*. This must be done for any MULT and DIV instruction to store their results into a register address. No status flags are updated.

Restrictions:
None

Example:

Assembly	Machine Code
mfhi r5	000000_00000_00000_00101_00000_010000
mfhi r9	000000_00000_00000_01001_00000_010000

Operand Values	Result
hi = 0x0000020D	rd = 0x0000020D
hi = 0xFFFFFF9D	r7 = 0xFFFFFF9D

MFLO

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

Format: mflo rd

Purpose: Move from Lo
Load the stored value inside *lo* from after doing a MULT or DIV instruction into destination register rd.

Description: rd ← *lo*
The 32-bit register rd obtains the value stored inside the register *lo*. This must be done for any MULT and DIV instruction to store their results into a register address. No status flags are updated.

Restrictions:
None

Example:

Assembly	Machine Code
`mfhi r5`	`000000_00000_00000_00101_00000_010010`
`mfhi r9`	`000000_00000_00000_01001_00000_010010`

Operand Values	Result
`lo = 0xFFFFFFE3`	`r5 = 0xFFFFFFE3`
`lo = 0xFFFFFF9D`	`r9 = 0xFFFFFF9D`

MOV

31 26	25 21	20 16	15 11	10 6	5 0
0x00	rs	rt	0	0	0x09

Format: mov rd

Purpose: Move
Content of one operand register is moved to the other operand register

Description: rt ← rs
The 32-bit value in register rs is loaded into register rt, overwriting the previous contents.

Restrictions:
None

Example:

Assembly	Machine Code
`mov r6, r3`	`000000_00000_00000_00000_00000_001001`

Operand Values	Result

`r6 = 0xFFFFFFFF`
`r3 = 0xABCDEF01`
	`r6 = 0xABCDEF01`

MULT

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

Format: mult rs, rt

Purpose: Multiplication
To multiply 32-bit signed integers

Description: rs * rt
lo ← lower 32-bit result
hi ← upper 32-bit result
The 32-bit values in rt and rs are multiplied together and stored in the HiLo register. Status flags (N, Z) are checked and updated depending on results.

Restrictions:
rt and rs must be signed values.

Example:

Assembly	Machine Code
mult r8,r7	000000_01000_00111_00000_00000_011000
mult r1,r5	000000_00001_00101_00000_00000_011000

Operand Values	Result

r8 = 0x0000000B
r7 = 0xFFFFFFFB
	hi = 0xFFFFFFFF
	Lo = 0xFFFFFFC9
	c=x, v=x, n=1, z=0

r1 = 0x00000005
r5 = 0x00000010
	hi = 0x00000000
	Lo = 0x00000050
	c=x, v=x, n=0, z=0

Operand 1: R8

31	0
0x0000000B	

Operand 2: R7

31	0
0xFFFFFFFB	

HI Register: Remainder

31	0
0xFFFFFFFF	

LO Register: Quotient

31	0
0xFFFFFFC9	

NOP

31 26	25 21	20 16	15 11	10 6	5 0
0x00	0	0	0	0	0x07

Format: nop

Purpose: No Operation
Does a no operation instruction

Description:
This instruction does no operation. It is best used to stall or delay the CPU.

Restrictions:
None

Example:

Assembly	Machine Code
`nop`	`000000_00000_00000_00000_00000_000111`

NOR

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

Format: nor rd, rs, rt

Purpose: Logical Not OR
Do a bitwise NOR to two 32-bit registers

Description: rd ← ~(rs | rt)
Bitwise or operation done on rs and rt and then negated and returned to rd. Status flags (N and Z) are checked and updated depending on results.

Restrictions:
None

Example:

Assembly	Machine Code
`nor r4,r8,r7`	`000000_01000_00111_00100_00000_100111`
`nor r9,r1,r5`	`000000_00001_00101_01001_00000_100111`

Operand Values	Result

`r8 = 0xF0F0F0F0`
`r7 = 0x0F0F0F0F`
	`R4 = 0x00000000`
	`c=x, v=x, n=0, z=1`

`r1 = 0x00000000`
`r5 = 0x000FF000`
	`R9 = 0xFFF00FFF`
	`c=x, v=x, n=1, z=0`

OR

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

Format: or rd, rs, rt

Purpose: Logical OR
Do a bitwise OR to two 32-bit registers

Description: rd ← rs | rt
Bitwise or operation done on rs and rt and returned to rd. Status flags (N and Z) are checked and updated depending on results.

Restrictions:
None

Example:

Assembly	Machine Code
`or r4,r8,r7`	`000000_01000_00111_00100_00000_100101`
`or r9,r1,r5`	`000000_00001_00101_01001_00000_100101`

Operand Values	Result

`r8 = 0xF0F0F0F0`
`r7 = 0x0F0F0F0F`
	`R4 = 0xFFFFFFFF`
	`c=x, v=x, n=1, z=0`

`r1 = 0x00000000`
`r5 = 0x000FF000`
	`R9 = 0x000FF000`
	`c=x, v=x, n=0, z=0`

ROTL

31 26	25 21	20 16	15 11	10 6	5 0
0x00	0	rt	rd	shamt	0x04

Format: rotl rd, rt, shamt

Purpose: Rotate Left
Rotates the contents of rt by shamt and store it in rd

Description: rd ← rt << shamt
rt is rotated to the left shamt amount of times and is put into rd. The LSB bits are rotated to the MSB side. Status flags (N, Z) are checked and updated depending on results.

Restrictions:
Shift amount must be within 5 bits as that is the size of the shamt field.

Example:

Assembly	Machine Code
`rotl r4,r7,3`	`000000_00000_00111_00100_00011_000100`
`rotl r9,r5,1`	`000000_00000_00101_01001_00001_000100`

Operand Values	Result

`r7 = 0xF0F0F0F0`	`R4 = 0x87878787`
	`c=x, v=x, n=0, z=0`

`r5 = 0xABCDEF01`	`R9 = 0x579BDE03`
	`c=x, v=x, n=0, z=0`

ROTR

31 26	25 21	20 16	15 11	10 6	5 0
0x00	0	rt	rd	shamt	0x04

Format: rotr rd, rt, shamt

Purpose: Rotate right
Rotates the contents of rt by shamt and store it in rd

Description: rd ← rt << shamt
rt is rotated to the right shamt amount of times and is put into rd. The MSB bits are rotated to the LSB side. Status flags (N, Z) are checked and updated depending on results.

Restrictions:
Shift amount must be within 5 bits as that is the size of the shamt field.

Example:

Assembly	Machine Code
`rotl r4,r7,3`	`000000_00000_00111_00100_00011_000100`
`rotl r9,r5,1`	`000000_00000_00101_01001_00001_000100`

Operand Values	Result

`r7 = 0x0F0F0F0F`	`R4 = 0x87878787`
	`c=x, v=x, n=0, z=0`

`r5 = 0xABCDEF01`	`R9 = 0xD5E6F780`
	`c=x, v=x, n=0, z=0`

SETIE

31 26	25 21	20 16	15 11	10 6	5 0
0x00	0	0	0	0	0x1F

Format: setie

Purpose: Set Interrupt Enable

Description:
The interrupt enable status flag will be set to 1.

Restrictions:
None

Example:

Assembly	Machine Code
`setie`	`000000_00000_00000_00000_00000_011111`

Operand Values	Result
`None`	`IE = 1`

SLL

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

Format: sll rd, rt, shamt

Purpose: Shift Left Logical
Shift the contents of rt by shamt and store it in rd

Description: rd ← rt << shamt
rt is shifted to the left shamt amount of times and is put into rd. Status flags (C, N, Z) are checked and updated depending on results.

Restrictions:
Shift amount must be within 5 bits as that is the size of the shamt field.

Example:

Assembly	Machine Code
`sll r4,r7,3`	`000000_00000_00111_00100_00011_000000`
`sll r9,r5,1`	`000000_00000_00101_01001_00001_000000`

Operand Values	Result

`r7 = 0x0F0F0F0F`	`R4 = 0x78787878`
	`c=0, v=x, n=0, z=0`

`r5 = 0x000FF000`	`R9 = 0x007F8000`
	`c=0, v=x, n=0, z=0`

SLT

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

Format: sltu rd ,rs, rt

Purpose: Set Less Than
Boolean expression where rd is set to 1’b1 when rs < rt otherwise rd is 1’b0

Description: rd ← (rs < rt) ? 1’b1 : 1’b0
rs is compared to rt, should the 32-bit rs be less than rt, rd gets {31’b0,1’b1} otherwise 32’b0. Status flags (N and Z) are checked and updated depending on results.

Restrictions:
rs and rt must be unsigned

Example:

Assembly	Machine Code
`slt r4,r1,r7,`	`000000_00001_00111_00100_00011_101010`
`slt r9,r2,r5`	`000000_00010_00101_01001_00001_101010`

Operand Values	Result

`r1 = 0x0000F000`
`r7 = 0xFF0F0F0F`
	`R4 = 0x00000000`
	`c=x, v=x, n=0, z=1`

`r2 = 0x00002456`
`r5 = 0x000FF000`
	`R9 = 0x00000001`
	`c=x, v=x, n=0, z=0`

SLTU

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

Format: sltu rd ,rs, rt

Purpose: Set Less Than Unsigned
Boolean expression where rd is set to 1’b1 when rs < rt otherwise rd is 1’b0

Description: rd ← (rs < rt) ? 1’b1 : 1’b0
rs is compared to rt, should the 32-bit rs be less than rt, rd gets {31’b0,1’b1} otherwise 32’b0. Status flags (N and Z) are checked and updated depending on results.

Restrictions:
rs and rt must be unsigned

Example:

Assembly	Machine Code
`sltu r4,r1,r7,`	`000000_00001_00111_00100_00011_101011`
`sltu r9,r2,r5`	`000000_00010_00101_01001_00001_101011`

Operand Values	Result

`r1 = 0x0000F000`
`r7 = 0xFF0F0F0F`
	`R4 = 0x00000001`
	`c=x, v=x, n=0, z=0`

`r2 = 0xF0002456`
`r5 = 0x000FF000`
	`R9 = 0x00000000`
	`c=x, v=x, n=0, z=1`

SRA

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

Format: sra rd,rt,shamt

Purpose: Shift Right Arithmetic
Shift the contents of rt by shamt and store it in rd and keep bit 32 the same

Description: rd ← rt >> shamt
rd receives the value of rt shifted by shamt to the right. Bit 32 will keep the MSB of rt. Status flags (C, N, Z) are checked and updated depending on results.

Restrictions:
Shift amount must be within 5 bits

Example:

Assembly	Machine Code
`sra r4,r7,3`	`000000_00000_00111_00100_00011_000011`
`sra r9,r5,1`	`000000_00000_00101_01001_00001_000011`

Operand Values	Result

`r7 = 0x04001100`	`R4 = 0x00800220`
	`c=0, v=x, n=0, z=0`

`r5 = 0xF0F0F0F0`	`R9 = 0xF8787878`
	`c=0, v=x, n=1, z=0`

SRL

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

Format: srl rd,rt,shamt

Purpose: Shift Right Logical
Logical shift right of the contents of rt by shamt and store in rd

Description: rd ← rt >> shamt
rt is shifted to the right shamt amount of times and is put into rd. Status flags (C, N, Z) are checked and updated depending on results.

Restrictions:
shamt can only be 5 bits at maximum.

Example:

Assembly	Machine Code
`srl r4,r7,3`	`000000_00000_00111_00100_00011_000010`
`srl r9,r5,1`	`000000_00000_00101_01001_00001_000010`

Operand Values	Result

`r7 = 0x04001100`	`R4 = 0x08000220`
	`c=0, v=x, n=0, z=0`

`r5 = 0xF0F0F0F0`	`R9 = 0x78787878`
	`c=0, v=x, n=0, z=0`

SUB

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

Format: sub rd,rs,rt

Purpose: Subtraction
Subtract two 32-bit signed values

Description: rd ← rs - rt
The 32-bit source registers rs and rt are subtracting from each other and their results are stored in the 32-bit destination register rd. Status flags (C, V, N, Z) are checked and updated depending on results.

Restrictions:
rs and rt must be signed values.

Example:

Assembly	Machine Code
`sub r4,r8,r7`	`000000_01000_00111_00100_00000_100010`
`sub r9,r1,r5`	`000000_00001_00101_01001_00000_100010`

Operand Values	Result

`r8 = 0xFFFF0000`
`r7 = 0x0000FFFF`
	`R4 = 0xFFFE0001`
	`c=0, v=0, n=1, z=0`

`r1 = 0x11111111`
`r5 = 0x00000010`
	`R9 = 0x11111101`
	`c=0, v=0, n=0, z=0`

SUBU

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

Format: subu rd,rs,rt

Purpose: Subtraction Unsigned
Subtract 32-bit rs and rt as unsigned integers

Description: rd ← rs - rt
The 32-bit source registers rs and rt are subtracted from each and their results are stored in the 32-bit destination register rd. Status flags (C, V and Z) are checked and updated depending on results. Negative status flag is unaffected by this instruction.

Restrictions:
rs and rt must be unsigned values

Example:

Assembly	Machine Code
subu r4,r8,r7	000000_01000_00111_00100_00000_100100
subu r9,r1,r5	000000_00001_00101_01001_00000_100100

Operand Values	Result

r8 = 0xFFFF0000
r7 = 0x0000FFFF
	R4 = 0xFFFE0001
	c=0, v=0, n=0, z=0

r1 = 0x11111111
r5 = 0x00000010
	R9 = 0x11111101
	c=0, v=0, n=0, z=0

XOR

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

Format: xor rd,rs,rt

Purpose: Logical Exclusive OR
Do a bitwise exclusive or to two 32-bit registers

Description: rd ← rs^rt
rs is exclusive or’ed with rt and placed into rd. Status flags (N and Z) are checked and updated depending on results.

Restrictions:
None

Example:

Assembly	Machine Code
`xor r4,r8,r7`	`000000_01000_00111_00100_00000_100110`
`xor r9,r1,r5`	`000000_00001_00101_01001_00000_100110`

Operand Values	Result

`r8 = 0xFFFF0000`
`r7 = 0x0000FFFF`
	`R4 = 0xFFFFFFFF`
	`c=x, v=x, n=1, z=0`

`r1 = 0x11111111`
`r5 = 0x00000010`
	`R9 = 0x11111101`
	`c=x, v=x, n=0, z=0`

3. Instruction Set I-Type List

ADDI

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

Format: add rt, rs, immediate

Purpose: Add Immediate
To add an immediate or signed value to a register

Description: rt ← rs + immediate
The 16-bit immediate value offset is summed together with the 32-bit value inside the register rs and result is then stored inside the 32-bit destination register rt. Status flags (C, V, N, Z) are checked and updated depending on results.

Restrictions:
The immediate value is restricted to 16-bits of values (-32,768 to 32,767).

Example:

Assembly	Machine Code
`addi r9, r1, 12`	`001000_00001_01001_0000000000001100`
`addi r12, r7, -28`	`001000_00111_01100_1111111111100100`

Operand Values	Result

`r1 = 0xf0f03c3c`
`Immediate = 0xC`
	`r9 = 0xF0F03C48`
	`c=0, v=0, n=1, z=0`

`r7 = 0xFFFFFFFF`
`Immediate = 0xFFFFFFE4`
	`r12 = 0x0000001B`
	`c=0, v=0, n=0, z=0`

ANDI

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

Format: andi rt, rs, immediate

Purpose: Logical AND Immediate
Do a bitwise AND using rs with an immediate value

Description: rt ← rs & immediate
Extend immediate value to 32-bits with 16’b0. Perform and operation on rs and immediate and put into rd. Status flags (N, Z) are checked and updated depending on results.

Restrictions:
The immediate value is restricted to 16-bits of values (-32,768 to 32,767).

Example:

Assembly	Machine Code
`andi r9, r1, 12`	`001100_00001_01001_0000000000001100`
`andi r12, r7, 5128`	`001100_00111_01100_0001010000001000`

Operand Values	Result

`r1 = 0xF0F03C3C`
`Immediate = 0xC`
	`r9 = 0x0000000C`
	`c=x, v=x, n=0, z=0`

`r7 = 0xF312EBC7`
`Immediate = 0x1408`
	`r12 = 0x00000000`
	`c=x, v=x, n=0, z=1`

BEQ

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

Format: beq rt, rs, immediate

Purpose: Branch if Equal
Compare and rt and rs, and if they are equal branch with offset

Description:

```
if(rs == rt)
    pc ← pc + 4 + address
else
    pc ← pc + 4
```

If rt equals rs, pc will get pc + 4 + {14'b(address[15]),address(16-bits),2'b0}.

Restrictions:
The immediate value is restricted to 16-bits of values (-32,768 to 32,767).

Example:

Assembly	Machine Code
`beq r9, r1, Label`	`000100_00001_01001_0000000000010000`
`beq r12, r7, Label`	`000100_00111_01100_0001010000001000`

Operand Values	Result

`r1 = 0xF0F03C3C`
`r9 = 0xF0F03C3C`
	`PC = PC + 0x4 + 0x00000010`

`r7 = 0xF312EBC7`
`r12 = 0xFFFFFFFF`
	`PC = PC + 0x4`

BGTZ

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

Format: bgtz rt, immediate

Purpose: Branch if greater than to Zero
Compare if the operand is greater than zero and branch.

Description:
```
if(rs > 0)
    pc ← pc + 4 + address
else
    pc ← pc + 4
```
If rt is greater than zero, pc will get pc + 4 + {14’b(address[15]),address(16-bits),2’b0}.

Restrictions:
The immediate value is restricted to 16-bits of values (-32,768 to 32,767).

Example:

Assembly	Machine Code
`bgtz r9, Label`	`000111_00000_01001_0000000000010000`
`bgtz r12, Label`	`000111_00000_01100_0001010000001000`

Operand Values	Result
`r9 = 0x00F03C3C`	`PC = PC + 0x4 + 0x00000010`
`r12 = 0x8FFFFFFF`	`PC = PC + 0x4`

BLEZ

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

Format: blez rs, immediate

Purpose: Branch if Less than or Equal to Zero
Compare if rs is less than or equal to 0 and branch to address

Description:

```
if(rs <= 0)
    pc ← pc + 4 + address
else
    pc ← pc + 4
```

If rt is less than or equal to 0, pc will get pc + 4 + {14’b(address[15]),address(16-bits),2’b0}.

Restrictions:
The immediate value is restricted to 16-bits of values (-32,768 to 32,767).

Example:

Assembly	Machine Code
`blez r9, Label`	`000110_00000_01001_0000000000010000`
`blez r10, Label`	`000110_00000_01010_0000000000000100`
`blez r12, Label`	`000110_00000_01100_0001010000001000`

Operand Values	Result
`r9 = 0xF0F03C3C`	`PC = PC + 0x4 + 0x00000010`
`r10 = 0x00000000`	`PC = PC + 0x4 + 0x00000004`
`r12 = 0x7FFFFFFF`	`PC = PC + 0x4`

BNE

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

Format: bne rt, rs, immediate

Purpose: Branch if not equal
Compare rs and rt and if they are not equal, branch to address

Description:

```
if(rs != rt)
    pc ← pc + 4 + address
else
    pc ← pc + 4
```

If rt is not equal to rs, pc will get pc + 4 + {14’b(address[15]),address(16-bits),2’b0}.

Restrictions:
The immediate value is restricted to 16-bits of values (-32,768 to 32,767).

Example:

Assembly	Machine Code
`bne r9, r1, Label`	`000101_00001_01001_0000000000010000`
`bne r12, r7, Label`	`000101_00111_01100_0001010000001000`

Operand Values	Result

`r1 = 0xF0F03C3C`
`r9 = 0xF0F03C3C`
	`PC = PC + 0x4`

`r7 = 0xF312EBC7`
`r12 = 0xFFFFFFFF`
	`PC = PC + 0x4 + 0x1408`

LUI

31 26	25 21	20 16	15 0
0x0F	0	rt	Immediate

Format: lui rt, immediate

Purpose: Load Upper Immediate
Load the upper half of a word to rt

Description: rt[31:16] ← immediate
Place 16-bit immediate value into upper 16-bits.

Restrictions:
The immediate value is restricted to 16-bits of values (-32,768 to 32,767).

Example:

Assembly	Machine Code
`lui r9, 0xAAAA`	`001111_00001_01001_1010101010101010`
`lui r12, 0xFFFF`	`001111_00111_01100_1111111111111111`

Operand Values	Result
`r9 = 0x000000000`	`r9 = 0xAAAA0000`
`r12 = 0x3333AAAA`	`r12 = 0xFFFFAAAA`

LW

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

Format: lw rt, immediate(rs)

Purpose: Load Word
Load a word from Memory

Description: rt ← M[immediate + rs]
Load 32-bit value from memory at the offset + rs which gets placed into rt.

Restrictions:
Effective address must be word aligned.

Example:

Assembly	Machine Code
`lw r9, 0(r1)`	`100011_00001_01001_0000000000000000`
`lw r12, 8(r7)`	`100011_00111_01100_0000000000001000`

Operand Values	Result
`M[r1+0] = 0xBBBBBBBB`	`r9 = 0xBBBBBBBB`
`M[r7+8] = 0x1000004C`	`r12 = 0x1000004C`

ORI

31 26	25 21	20 16	15 0
0x0D	rs	rt	Immediate

Format: ori rt, rs, immediate

Purpose: Or Immediate
Logical or rs and immediate

Description: rt ← rs | zero-extended(address)
Zero fill upper 16 bits of the address. Bitwise OR the result with rs and place into rt. Status flags (N, Z) are checked and updated depending on results.

Restrictions:
The immediate value is restricted to 16-bits of values (-32,768 to 32,767).

Example:

Assembly	Machine Code
`ori r9, r1, 0xAAAA`	`001101_00001_01001_1010101010101010`
`ori r12,r7, 0xFFFF`	`001101_00111_01100_1111111111111111`

Operand Values	Result
`r1 = 0x55555555`	`r9 = 0x5555FFFF`
`r7 = 0x3333B111`	`r12 = 0x3333FFFF`

SLTI

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

Format: slti rt, rs, immediate

Purpose: Set less than immediate
Set or clear if register is less than the immediate.

Description: rt ← rs < sign extended(address)
Compare rs and sign extended address. If rs is less than the address, set rt = 32’b1, otherwise set rt = 32’b0.

Restrictions:
The immediate value is restricted to 16-bits of values (-32,768 to 32,767).

Example:

Assembly	Machine Code
`slti r9, r1, 0xAAAA`	`001010_00001_01001_1010101010101010`
`slti r12, r7, 0x1234`	`001010_00111_01100_0001001000110100`

Operand Values	Result

`r1 = 0x00001357`
`Sign extended imm. = 0xFFFFAAAA`
	`r9 = 0x00000000`

`r7 = 0xFA5A5A5A`
`Sign extended imm. = 0x00001234`
	`r9 = 0x00000001`

SLTIU

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

Format: sltiu rt, rs, immediate

Purpose: Set less than immediate unsigned
Set or clear if register is less than the immediate.

Description: rt ← rs < sign extended(immediate)
Compare rs with the sign extended immediate. If rs is less than the unsigned immediate value, rt gets a 32’b1 otherwise rt gets 32’b0.

Restrictions:
The immediate value is restricted to 16-bits of values (0 to 65,535).

Example:

Assembly	Machine Code
`sltiu r9, r1, 0xAAAA`	`001011_00001_01001_1010101010101010`
`sltiu r12, r7, 0x1234`	`001011_00111_01100_0001001000110100`

Operand Values	Result

`r1 = 0x00001357`
`Sign extended imm. = 0xFFFFAAAA`
	`r9 = 0x00000001`

`r7 = 0xFA5A5A5A`
`Sign extended imm. = 0x00001234`
	`r9 = 0x00000000`

SW

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

Format: sw rt, immediate(rs)

Purpose: Store Word
Store a word from memory

Description: M[immediate + rs] ← rt
Store a 32-bit value to a memory location determined by rs + immediate.

Restrictions:
The immediate value is restricted to 16-bits of values (-32,768 to 32,767).

Example:

Assembly	Machine Code
`sw r9, 0(r1)`	`101011_00001_01001_0000000000000000`
`sw r12, 8(r7)`	`101011_00111_01100_0000000000001000`

Operand Values	Result

`r9 = 0xFFFFFFFF`
`M[r1+0] = 0x9ABCDEF0`
	`M[r1+0] = 0xFFFFFFFF`

`r12 = 0xABCDEF01`
`M[r7+8] = 0x1000004C`
	`M[r7+8] = 0xABCDEF01`

XORI

31 26	25 21	20 16	15 0
0x0E	rs	rt	16-bit Immediate

Format: xori rt, rs, immediate

Purpose: Logical Exclusive OR immediate
Do a bitwise xor rs and the immediate value

Description: rt ← rs ^ zero-extended(immediate)
The zero extended immediate value is exclusive or'ed with rs and placed into rt. Status flags (N, Z) are checked and updated depending on results.

Restrictions:
The immediate value is restricted to 16-bits of values (-32,768 to 32,767).

Example:

Assembly	Machine Code
`xori r9, r1, 0xAAAA`	`001110_00001_01001_1010101010101010`
`xori r12, r7, 0x1234`	`001110_00111_01100_0001001000110100`

Operand Values	Result

`r1 = 0xFFFFFFFF`
`Sign extended imm. = 0xFFFFAAAA`
	`r9 = 0x00005555`

`r7 = 0xFA5A5A5A`
`Sign extended imm. = 0x00001234`
	`r9 = 0xFA5A486E`

4. Instruction Set J-Type List

J

31 26	25 0
0x02	26-bit ADDRESS

Format: j ADDRESS

Purpose: Jump
Loads the specified bit address into the PC.

Description: PC ← ADDRESS
Jumps to the effective address found by concatenated the 4 most significant bits of PC, offset, and two zeroes.

Restrictions:
MIPS addresses increment/decrement by 4, so the lower bytes must be in multiples of 4 for the jump to work.

Example:

Assembly	Machine Code
`j top`	`000010_00000000000000000011111010`
`j bot`	`000010_00000000000000FFFFFFFFFFFF`

Operand Values	Result
immediate = 0x000000FA PC Initial = 0x0000AB70 Effective Address = 0x000001E8	PC = 0x000001E8
immediate = 0x00000FFF PC Initial = 0x0000AB70 Effective Address = 0x00003FFC	PC = 0x00003FFC

Jal

31 26	25 0
0x03	26-bit ADDRESS

Format: jal ADDRESS

Purpose: Jump and Link
Does a procedure call by doing a jump after saving the current PC.

Description: \$ra ← PC + 4
PC ← ADDRESS
Store the return address in return address(ra) register and set the PC to the effective address found by concatenating the four most significant bits of current PC, offset, and two zeroes.

Restrictions:
MIPS addresses increment/decrement by 4, so the lower bytes must be in multiples of 4 for the jump to work.

Example:

Assembly	Machine Code
`j top`	`000011_00000000000000000011111010`
`j bot`	`000011_00000000000000FFFFFFFFFFFF`

Operand Values	Result

`immediate = 0x000000FA`
`PC Initial = 0x0000AB70`
`Effective Address = 0x000001E8`
	`ra = 0x0000AB74`
	`PC = 0x000001E8`

`immediate = 0x00000FFF`
`PC Initial = 0x0000AB70`
`Effective Address = 0x00003FFC`
	`ra = 0x0000AB74`
	`PC = 0x00003FFC`

5. Instruction Set Enhanced List

INPUT

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

Format: input rt immediate(rs)

Purpose: Load Register from I/O
Load register with a location in memory.

Description: rt ←io M[rs + address]
The 32-bit data at Memory Location of the rs register + the 16-bit signed extended address is loaded into register rt.

Restrictions:
None

Example:

Assembly	Machine Code
`input r9, 0(r1)`	`011100_00001_01001_0000000000000000`
`input r12, 8(r7)`	`011100_00111_01100_0000000000001000`

Operand Values	Result
`ioM[r1+0] = 0xBBBBBBBB`	`r9 = 0xBBBBBBBB`
`ioM[r7+8] = 0x1000004C`	`r12 = 0x1000004C`

OUTPUT

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

Format: output rt immediate(rs)

Purpose: Store Register from I/O
Store register to a location in memory

Description: ioM[rs + address] ← rt
The 32-bit register rt is stored into a memory location in I/O memory of the rt + 16-bit signed extended address.

Restrictions:
None

Example:

Assembly	Machine Code
`sw r9, 0(r1)`	`011101_00001_01001_0000000000000000`
`sw r12, 8(r7)`	`011101_00111_01100_0000000000001000`

Operand Values	Result

`r9 = 0xFFFFFFFF`
`M[r1+0] = 0x9ABCDEF0`
	`ioM[r1+0] = 0xFFFFFFFF`

`r12 = 0xABCDEF01`
`M[r7+8] = 0x1000004C`
	`ioM[r7+8] = 0xABCDEF01`

POP

31 26	25 21	20 16	15 11	10 6	5 0
0x00	rs	rt	0	0	0x14

Format: pop rt

Purpose: Pop
Load register with content at stack pointer.

Description: rt← M[$sp]
$sp← $sp+4
The 32-bit register rt gets the content at memory location of the stack pointer. This MIPS does a post-increment pop.

Restrictions:
None

Example:

Assembly	Machine Code
`pop r2`	`000000_00000_00010_00000_00000_010100`

Operand Values	Result

`r2 = 0x0000000`
`$sp = 0x008`
`M[00C] = 0x12345678`
`M[$sp] = 0xABCDEF01`
	`r2 = 0xABCDEF01`
	`$sp = 0x00C`
	`M[$sp] = 0x12345678`

PUSH

31 26	25 21	20 16	15 11	10 6	5 0
0x00	rs	rt	0	0	0x13

Format: pop rt

Purpose: Pop
Load register with content at stack pointer.

Description: $sp ← $sp-4
M[$sp] ← rt
The 32-bit register rt's content is stored at memory location of the stack pointer. This MIPS does a pre-decrement push.

Restrictions:
None

Example:

Assembly	Machine Code
`push r2`	`000000_00000_00010_00000_00000_010011`

Operand Values	Result

`r2 = 0xABCDEF01`
`$sp = 0x00C`
`M[$sp] = 0x12345678`
`M[008] = 0x00000000`
	`r2 = 0xABCDEF01`
	`$sp = 0x008`
	`M[$sp] = 0xABCDEF01`

RETI

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

Format: reti

Purpose: Return from Interrupt
Returns to where PC was before ISR was triggered.

Description: PC ← M[$sp]
Loads PC with the return address stored in the memory location of the stack pointer. This returns PC to the value it was at before the ISR was triggered so that the normal program will continue.

Restrictions:
An ISR must have triggered.

Example:

Assembly	Machine Code
`reti`	`011110_00000_00000_0000000000000000`

Operand Values	Result
`None`	`PC = M[$sp]`

SIMD8

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

Format: simd8 rd, rs, rt

Purpose: simd8
Toggles vector-type instructions for 8-bit values

Description:
This instruction allows the programmer to use vector type of 8-bit values. The 32-bit register is used to represent four 8-bit registers.

Restrictions:
None

Example:

31 25	21 16	15 8	7 0
byte	byte	byte	byte

SIMD16

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

Format: simd16 rd, rs, rt

Purpose: simd16
Toggles vector-type instructions for 16-bit values

Description:
This instruction allows the programmer to use vector type of 16-bit values. The 32-bit register is used to represent two 16-bit registers.

Restrictions:
None

Example:

31 16	15 0
Word	Word

“E_KEY”

31 26	25 21	20 16	15 0
0x1F	0	0	0

Format: None

Purpose:
Reserved MIPS opcode

Description:
Reserved MIPS opcode.

Restrictions:
None

Example:
None

6. Instruction Set Vector-Type List

ADD8

31 26	25 21	20 16	15 11	10 6	5 0
0x2C	rs	rt	rd	0	0x20

Format: add8 rd, rs, rt

Purpose: Vector Add 8-bits
Vector addition of two 32-bit registers that represent four vector byte values.

Description: rd ← rs + rt
The 32-bit source registers rs and rt are added together and their results are stored in the 32-bit destination register rd. Each 32-bit registers represent the four vector byte values. Status flag Z is checked and updated depending on results.

Restrictions:

Example:

Assembly	Machine Code
`add8 r3, r1, r2`	`101100_00001_00010_00011_00000_100000`
`add8 r8, r7, r4`	`101100_00111_00001_01000_00000_100000`

Operand Values	Result

`r1 = 0xF5F6F7F8`
`r2 = 0x206220E1`
	`r3 = 0x054707D9`
	`c=x, v=x, n=x, z=0`

`r7 = 0xFFFFFF9D`
`r4 = 0xFFFFFF9C`
	`r8 = 0xFEFEFE39`
	`c=x, v=x, n=x, z=0`

ADD16

31 26	25 21	20 16	15 11	10 6	5 0
0x2D	rs	rt	rd	0	0x20

Format: add8 rd, rs, rt

Purpose: Vector Add 16-bits
Vector addition of two 32-bit registers that represent two vector word values (DUAL16).

Description: rd ← rs + rt
The 32-bit source registers rs and rt are added together and their results are stored in the 32-bit destination register rd. Each 32-bit registers represent the two vector word values. Status flag Z is checked and updated depending on results.

Restrictions:

Example:

Assembly	Machine Code
add16 r3, r1, r2	101101_00001_00010_00011_00000_100000
add16 r8, r7, r4	101101_00111_00001_01000_00000_100000

Operand Values	Result

r1 = 0xF5F6F7F8
r2 = 0x206220E1
	r3 = 0x064808D9
	c=x, v=x, n=x, z=0

r7 = 0xFFFFFF9D
r4 = 0xFFFFFF9C
	r8 = 0xFFFEFF39
	c=x, v=x, n=x, z=0

AND8

31 26	25 21	20 16	15 11	10 6	5 0
0x2C	rs	rt	rd	0	0x24

Format: and8 rd, rs, rt

Purpose: Vector Logical AND 8-bit
Do a vector bitwise AND to two 32-bit that represent four vector byte values.

Description: rd ← rs & rt
Doing a Logical AND, each bits of two 32-bit registers will either become 1s if both bits are 1s and 0s if they don't. The results of the rt and rs are then stored into the 32-bit register rd. Each 32-bit registers represent the four vector byte values. Status flag Z is checked and updated depending on results.

Restrictions:
None

Example:

Assembly	Machine Code
`and8 r9, r1, r5`	`101100_00001_00101_01001_00000_100100`
`and8 r12, r7, r4`	`101100_00111_00001_01100_00000_100100`

Operand Values	Result

`r1 = 0xF0F03C3C`
`r5 = 0xBF0FF5F5`
	`r9 = 0xB0003434`
	`c=x, v=x, n=x, z=0`

`r7 = 0xA7F8F59D`
`r4 = 0x41049CCC`
	`r9 = 0x0100948C`
	`c=x, v=x, n=x, z=0`

AND16

31 26	25 21	20 16	15 11	10 6	5 0
0x2D	rs	rt	rd	0	0x24

Format: and8 rd, rs, rt

Purpose: Vector Logical AND 8-bit
Do a vector bitwise AND to two 32-bit that represent two vector word values (DUAL16)

Description: rd ← rs & rt
Doing a Logical AND, each bits of two 32-bit registers will either become 1s if both bits are 1s and 0s if they don't. The results of the rt and rs are then stored into the 32-bit register rd. Each 32-bit registers represent the two vector word values. Status flag Z is checked and updated depending on results.

Restrictions:
None

Example:

Assembly	Machine Code
`and16 r9, r1, r5`	`101101_00001_00101_01001_00000_100100`
`and16 r12, r7, r4`	`101101_00111_00001_01100_00000_100100`

Operand Values	Result

`r1 = 0xF0F03C3C`
`r5 = 0xBF0FF5F5`
	`r9 = 0xB0003434`
	`c=x, v=x, n=x, z=0`

`r7 = 0xA7F8F59D`
`r4 = 0x41049CCC`
	`r9 = 0x0100948C`
	`c=x, v=x, n=x, z=0`

DIV8

31 26	25 21	20 16	15 11	10 6	5 0
0x2C	rs	rt	0	0	0x1A

Format: div8 rs, rt

Purpose: Vector Division 8-bits
Divides two 32-bit registers and stores the results into two other 32-bit registers named *hi* and *lo*. Each 32-bit registers represent the four vector byte values.

Description: rs ÷ rt
lo ← quotient
hi ← remainder
The 32-bit register rs is the numerator and the 32-bit register rt is the denominator. They are divided and results in a 64 bit value that is split into two parts: the quotient and the remainder. The quotient is loaded into the 32-bit register *lo* and the remainder is loaded into the 32-bit register *hi*. Each 32-bit registers represent the four vector byte values. Status flag Z is checked and updated depending on results.

Restrictions:
If the register rt contains a value of 0, the results will be undefined.

Example:

Assembly	Machine Code
`div8 r1, r5`	`101100_00001_00101_00000_00000_011010`

Operand Values	Result

`r1 = 0xE956E003`
`r5 = 0x4BB4011E`
	`hi = 0x44433C78`
	`lo = 0x00E0005A`
	`c=x, v=x, n=x, z=0`

Operand 1: R1

31 25	21 16	15 8	7 0
0xE9	0x56	0xE0	0x03

Operand 2: R5

31 25	21 16	15 8	7 0
0x4B	0xB4	0x01	0x1E

HI Register: Remainders

31 25	21 16	15 8	7 0
0x44	0x43	0x3C	0x78

LO Register: Quotient

31 25	21 16	15 8	7 0
0x00	0xE0	0x00	0x5A

DIV16

31 26	25 21	20 16	15 11	10 6	5 0
0x2D	rs	rt	0	0	0x1A

Format: div8 rs, rt

Purpose: Vector Division 8-bits
Divides two 32-bit registers and stores the results into two other 32-bit registers named *hi* and *lo*. Each register represents two 16-bit values (DUAL16)

Description: rs ÷ rt
lo ← quotient
hi ← remainder
The 32-bit register rs is the numerator and the 32-bit register rt is the denominator. They are divided and results in a 64 bit value that is split into two parts: the quotient and the remainder. The quotient is loaded into the 32-bit register *lo* and the remainder is loaded into the 32-bit register *hi*. Each 32-bit registers represent the two vector word values. Status flag Z is checked and updated depending on results.

Restrictions:
If the register rt contains a value of 0, the results will be undefined.

Example:

Assembly	Machine Code
`div16 r1, r5`	`101101_00001_00101_00000_00000_011010`

Operand Values	Result

`r1 = 0xE956E003`
`r5 = 0x4BB4011E`
	`hi = 0x45004278`
	`lo = 0x00FA435A`
	`c=x, v=x, n=x, z=0`

Operand 1: R1

31 16	15 0
0xE956	0xE003

Operand 2: R5

31 16	15 0
0x4BB4	0x011E

HI Register: Remainder

31 16	15 0
0x4500	0x4278

LO Register: Quotient

31 16	15 0
0x00FA	0x435A

MULT8

31 26	25 21	20 16	15 11	10 6	5 0
0x2C	rs	rt	0	0	0x18

Format: mult8 rs, rt

Purpose: Vector Multiplication 8-bits
To multiply 32-bit signed integers. Each register represents four 8-bit values (QUAD8)

Description: rs * rt

lo ← lower 32-bit result

hi ← upper 32-bit result

The 32-bit values in rt and rs are multiplied together and stored in the HiLo register. Status flag Z is checked and updated depending on results. Each register represents four vector bytes and result in two vector word values.

Restrictions:
rt and rs must be signed values.

Example:

Assembly	Machine Code
`mult8 r8,r7`	`101100_01000_00111_00000_00000_011000`

Operand Values	Result

`r8 = 0xF5F6F7F8`
`r7 = 0x105110E1`
	`hi = 0x0F504DD6`
	`Lo = 0x0F70D9F8`
	`c=x, v=x, n=x, z=0`

Operand 1: R8

31 25	21 16	15 8	7 0
0xF5	0xF6	0xF7	0xF8

Operand 2: R7

31 25	21 16	15 8	7 0
0x10	0x51	0x11	0xE1

HI REGISTER: Bits[31:25] and Bits[21:16]

31 16	15 0
0x0F50	0x4DD6

LO REGISTER: Bits[15:8] and Bits[7:0]

31 16	15 0
0x0F70	0xD9F8

MULT16

31 26	25 21	20 16	15 11	10 6	5 0
0x2D	rs	rt	0	0	0x18

Format: mult16 rs, rt

Purpose: Vector Multiplication 16-bits
To multiply 32-bit signed integers. Each 32-bit registers represent the two vector word values.

Description: rs * rt
lo ← lower 32-bit result
hi ← upper 32-bit result
The 32-bit values in rt and rs are multiplied together and stored in the HiLo register. Each 32-bit registers represent the two vector word values and result in 32-bit values. The upper 16 bits are multiplied together and loaded into the hi register and the lower 16 bits are multiplied together and loaded into the lo register. Status flag Z is checked and updated depending on results.

Restrictions:
rt and rs must be signed values.

Example:

Assembly	Machine Code
`mult16 r8,r7`	`101101_01000_00111_00000_00000_011000`

Operand Values	Result

`r8 = 0xF5F6F7F8`
`r7 = 0x105110E1`
	`hi = 0x0FAD32D6`
	`Lo = 0x105970F8`
	`c=x, v=x, n=x, z=0`

Operand 1: R8

31 16	15 0
0xF5F6	0xF7F8

Operand 2: R7

31 16	15 0
0x1051	0x10E1

HI REGISTER: Bits[31:16]

31 0
0x0FAD32D6

LO REGISTER: Bits[15:0]

31 0
0x105970F8

NOR8

31 26	25 21	20 16	15 11	10 6	5 0
0x2C	rs	rt	rd	0	0x27

Format: nor8 rd, rs, rt

Purpose: Vector Logical Not OR 8-bit
Do a vector bitwise NOR to two 32-bit registers. Each 32-bit registers represent the four vector byte values.

Description: rd ← ~(rs | rt)
Bitwise or operation done on rs and rt and then negated and returned to rd. Each 32-bit registers represent the four vector byte values. Status flag Z is checked and updated depending on results.

Restrictions:
None

Example:

Assembly	Machine Code
`nor8 r4,r8,r7`	`101100_01000_00111_00100_00000_100111`
`nor8 r9,r1,r5`	`101100_00001_00101_01001_00000_100111`

Operand Values	Result

`r8 = 0xF0F0F0F0`
`r7 = 0x0F0F0F0F`
	`R4 = 0x00000000`
	`c=x, v=x, n=x, z=1`

`r1 = 0x00000000`
`r5 = 0x000FF000`
	`R9 = 0xFFF00FFF`
	`c=x, v=x, n=x, z=0`

NOR16

31 26	25 21	20 16	15 11	10 6	5 0
0x2D	rs	rt	rd	0	0x27

Format: nor16 rd, rs, rt

Purpose: Vector Logical Not OR16-bit
Do a vector bitwise NOR to two 32-bit registers. Each 32-bit registers represent the two vector word values.

Description: rd ← ~(rs | rt)
Bitwise or operation done on rs and rt and then negated and returned to rd. Each 32-bit registers represent the two vector word values. Status flag Z is checked and updated depending on results.

Restrictions:
None

Example:

Assembly	Machine Code
`nor16 r4,r8,r7`	`101101_01000_00111_00100_00000_100111`
`nor16 r9,r1,r5`	`101101_00001_00101_01001_00000_100111`

Operand Values	Result

`r8 = 0xF0F0F0F0`
`r7 = 0x0F0F0F0F`
	`R4 = 0x00000000`
	`c=x, v=x, n=x, z=1`

`r1 = 0x00000000`
`r5 = 0x000FF000`
	`R9 = 0xFFF00FFF`
	`c=x, v=x, n=x, z=0`

OR8

31 26	25 21	20 16	15 11	10 6	5 0
0x2C	rs	rt	rd	0	0x25

Format: o8r rd, rs, rt

Purpose: Vector Logical OR 8-bit
Do a vector bitwise OR to two 32-bit registers. Each 32-bit registers represent the four vector byte values.

Description: rd ← rs | rt
Vector bitwise or operation done on rs and rt and returned to rd. Each 32-bit registers represent the four vector byte values. Status flag Z is checked and updated depending on results.

Restrictions:
None

Example:

Assembly	Machine Code
`or8 r4,r8,r7`	`101100_01000_00111_00100_00000_100101`
`or8 r9,r1,r5`	`101100_00001_00101_01001_00000_100101`

Operand Values	Result

`r8 = 0xF0F0F0F0`
`r7 = 0x0F0F0F0F`
	`R4 = 0xFFFFFFFF`
	`c=x, v=x, n=x, z=0`

`r1 = 0x00000000`
`r5 = 0x000FF000`
	`R9 = 0x000FF000`
	`c=x, v=x, n=x, z=0`

OR16

31 26	25 21	20 16	15 11	10 6	5 0
0x2D	rs	rt	rd	0	0x25

Format: or16 rd, rs, rt

Purpose: Vector Logical OR 16-bit
Do a vector bitwise OR to two 32-bit registers. Each 32-bit registers represent the two vector word values.

Description: rd ← rs | rt
Vector bitwise or operation done on rs and rt and returned to rd. Each 32-bit registers represent the two vector word values. Status flag Z is checked and updated depending on results.

Restrictions:
None

Example:

Assembly	Machine Code
`or16 r4,r8,r7`	`101101_01000_00111_00100_00000_100101`
`or16 r9,r1,r5`	`101101_00001_00101_01001_00000_100101`

Operand Values	Result

r8 = 0xF0F0F0F0
r7 = 0x0F0F0F0F
	R4 = 0xFFFFFFFF
	c=x, v=x, n=x, z=0

r1 = 0x00000000
r5 = 0x000FF000
	R9 = 0x000FF000
	c=x, v=x, n=x, z=0

ROTL8

31 26	25 21	20 16	15 11	10 6	5 0
0x2C	0	rt	rd	shamt	0x04

Format: rotl8 rd, rt, shamt

Purpose: Vector Rotate Left 8-bit
Vector rotates the contents of rt by shamt and store it in rd. Each 32-bit registers represent the four vector byte values.

Description: rd ← rt << shamt
rt is rotated to the left shamt amount of times and is put into rd. The LSB bits are rotated to the MSB side. Each 32-bit registers represent the four vector byte values. Status flag Z is checked and updated depending on results.

Restrictions:
Shift amount must be within 5 bits as that is the size of the shamt field.

Example:

Assembly	Machine Code
`rotl8 r4,r7,1`	`101100_00000_00111_00100_00011_000100`

Operand Values	Result

`r7 = 0x055A0107`	`R4 = 0x4BB4011E`
	`c=x, v=x, n=x, z=0`

ROTL16

31 26	25 21	20 16	15 11	10 6	5 0
0x2D	0	rt	rd	shamt	0x04

Format: rotl16 rd, rt, shamt

Purpose: Vector Rotate Left 16-bit
Vector rotates the contents of rt by shamt and store it in rd. Each 32-bit registers represent the two vector word values.

Description: rd ← rt << shamt
rt is rotated to the left shamt amount of times and is put into rd. The LSB bits are rotated to the MSB side. Each 32-bit registers represent the two vector word values.. Status flag Z is checked and updated depending on results.

Restrictions:
Shift amount must be within 5 bits as that is the size of the shamt field.

Example:

Assembly	Machine Code
rotl16 r4,r7,1	101101_00000_00111_00100_00011_000100

Operand Values	Result

r7 = 0x055A0107	R4 = 0x4AB5001F
	c=x, v=x, n=x, z=0

ROTR8

31 26	25 21	20 16	15 11	10 6	5 0
0x2C	0	rt	rd	shamt	0x04

Format: rotr8 rd, rt, shamt

Purpose: Vector Rotate right 8-bit
Rotates the contents of rt by shamt and store it in rd. Each 32-bit registers represent the four vector byte values.

Description: rd ← rt << shamt
rt is rotated to the right shamt amount of times and is put into rd. The MSB bits are rotated to the LSB side. Each 32-bit registers represent the four vector byte values. Status flag Z is checked and updated depending on results.

Restrictions:
Shift amount must be within 5 bits as that is the size of the shamt field.

Example:

Assembly	Machine Code
rotl8 r4,r7,1	101100_00000_00111_00100_00011_000100

Operand Values	Result

r7 = 0x055A0107	R4 = 0xD22D4087
	c=x, v=x, n=x, z=0

ROTR16

31 26	25 21	20 16	15 11	10 6	5 0
0x2D	0	rt	rd	shamt	0x04

Format: rotr16 rd, rt, shamt

Purpose: Vector Rotate right 16-bit
Vector rotates the contents of rt by shamt and store it in rd. Each 32-bit registers represent the two vector word values.

Description: rd ← rt << shamt
rt is rotated to the right shamt amount of times and is put into rd. The MSB bits are rotated to the LSB side.Each 32-bit registers represent the two vector word values. Status flag Z is checked and updated depending on results.

Restrictions:
Shift amount must be within 5 bits as that is the size of the shamt field.

Example:

Assembly	Machine Code
`rotl16 r4,r7,1`	`101101_00000_00111_00100_00011_000100`

Operand Values	Result

`r7 = 0x055A0107`	`R4 = 0x52ADc007`
	`c=x, v=x, n=x, z=0`

SLL8

31 26	25 21	20 16	15 11	10 6	5 0
0x2C	0	rt	rd	shamt	0x00

Format: sll8 rd, rt, shamt

Purpose: Vector Shift Left Logical 8-bit
Vector shift the contents of rt by shamt and store it in rd. Each 32-bit registers represent the four vector byte values.

Description: rd ← rt << shamt
rt is shifted to the left shamt amount of times and is put into rd. Each 32-bit registers represent the four vector byte values. Status flag Z is checked and updated depending on results.

Restrictions:
Shift amount must be within 5 bits as that is the size of the shamt field.

Example:

Assembly	Machine Code
`sll r9,r5,1`	`1011000_00000_00101_01001_00001_000000`

Operand Values	Result

`r5 = 0xAFAFAFAF`	`R9 = 0x5E5E5E5E`
	`c=0, v=x, n=0, z=0`

SLL16

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

Format: sll16 rd, rt, shamt

Purpose: Vector Shift Left Logical 8-bit
Vector shift the contents of rt by shamt and store it in rd. Each 32-bit registers represent the two vector word values.

Description: rd ← rt << shamt
rt is shifted to the left shamt amount of times and is put into rd. Each 32-bit registers represent the two vector word values. Status flag Z is checked and updated depending on results.

Restrictions:
Shift amount must be within 5 bits as that is the size of the shamt field.

Example:

Assembly	Machine Code
`sll r9,r5,1`	`1011001_00000_00101_01001_00001_000000`

Operand Values	Result

`r5 = 0xAFAFAFAF`	`R9 = 0x5F5E5F5E`
	`c=0, v=x, n=0, z=0`

SRA8

31 26	25 21	20 16	15 11	10 6	5 0
0x2C	0	rt	rd	shamt	0x03

Format: sra8 rd,rt,shamt

Purpose: Vector Shift Right Arithmetic 8-bits
Shift the contents of rt by shamt and store it in rd for each of the four vector byte values.

Description: rd ← rt >> shamt
rd receives the value of rt shifted by shamt to the right. Each of the four vector byte values are shifted. Status flag Z is checked and updated depending on results.

Restrictions:
Shift amount must be within 5 bits

Example:

Assembly	Machine Code
`sra8 r4, r7, 2`	`101100_00000_00111_00100_00010_000011`

Operand Values	Result

`r7 = 0x055A0107`	`R4 = 0xE916E003`
	`c=x, v=x, n=x, z=0`

SRA16

31 26	25 21	20 16	15 11	10 6	5 0
0x2D	0	rt	rd	shamt	0x03

Format: sra16 rd,rt,shamt

Purpose: Vector Shift Right Arithmetic 16-bits
Shift the contents of rt by shamt and store it in rd for each of the two vector word values.

Description: rd ← rt >> shamt
rd receives the value of rt shifted by shamt to the right. Each of the two vector word values are shifted. Status flag Z is checked and updated depending on results.

Restrictions:
Shift amount must be within 5 bits

Example:

Assembly	Machine Code
`sra16 r4, r7, 2`	`101101_00000_00111_00100_00010_000011`

Operand Values	Result

`r7 = 0x055A0107`	`R4 = 0xE956E003`
	`c=x, v=x, n=x, z=0`

SRL8

31 26	25 21	20 16	15 11	10 6	5 0
0x2C	0	rt	rd	shamt	0x02

Format: srl8 rd,rt,shamt

Purpose: Vector Shift Right Logical 8-bits
Vector Logical shift right of the contents of rt by shamt and store in rd. Each 32-bit registers represent the four vector byte values.

Description: rd ← rt >> shamt
rt is shifted to the right shamt amount of times and is put into rd. Each 32-bit registers represent the four vector byte values. Status flag Z is checked and updated depending on results.

Restrictions:
shamt can only be 5 bits at maximum.

Example:

Assembly	Machine Code
`srl8 r9,r5,1`	`101100_00000_00101_01001_00001_000010`

Operand Values	Result

`r5 = 0x055A0107`	`R9 = 0x57575757`
	`c=x, v=x, n=x, z=0`

SRL16

31 26	25 21	20 16	15 11	10 6	5 0
0x2D	0	rt	rd	shamt	0x02

Format: srl16 rd,rt,shamt

Purpose: Vector Shift Right Logical 16-bits
Vector Logical shift right of the contents of rt by shamt and store in rd. Each 32-bit registers represent the two vector word values.

Description: rd ← rt >> shamt
rt is shifted to the right shamt amount of times and is put into rd. Each 32-bit registers represent the two vector word values. Status flag Z is checked and updated depending on results.

Restrictions:
shamt can only be 5 bits at maximum.

Example:

Assembly	Machine Code
`srl16 r9,r5,1`	`101101_00000_00101_01001_00001_000010`

Operand Values	Result

`r5 = 0x055A0107`	`R9 = 0x57D757D7`
	`c=x, v=x, n=x, z=0`

SUB8

31 26	25 21	20 16	15 11	10 6	5 0
0x2C	rs	rt	rd	0	0x22

Format: sub8 rd,rs,rt

Purpose: Vector Subtraction 8-bit
Vector subtract two 32-bit signed values. Each 32-bit registers represent the four vector byte values.

Description: rd ← rs - rt
The 32-bit source registers rs and rt are subtracting from each other and their results are stored in the 32-bit destination register rd. Each 32-bit registers represent the four vector byte values. Status flag Z is checked and updated depending on results.

Restrictions:
None

Example:

Assembly	Machine Code
sub8 r4,r8,r7	101100_01000_00111_00100_00000_100010
sub8 r9,r1,r5	101100_00001_00101_01001_00000_100010

Operand Values	Result

r8 = 0xFFFF0000
r7 = 0x0000FFFF
	R4 = 0xFFFFF1F1
	c=x, v=x, n=x, z=0

r1 = 0xE956E003
r5 = 0x4BB4011E
	R9 = 0x9EA2DFE5
	c=x, v=x, n=x, z=0

SUB16

31 26	25 21	20 16	15 11	10 6	5 0
0x2D	rs	rt	rd	0	0x22

Format: sub16 rd,rs,rt

Purpose: Vector Subtraction 16-bit
Vector subtract two 32-bit signed values. Each 32-bit registers represent the two vector word values.

Description: rd ← rs - rt
The 32-bit source registers rs and rt are subtracting from each other and their results are stored in the 32-bit destination register rd. Each 32-bit registers represent the two vector word values. Status flag Z is checked and updated depending on results.

Restrictions:
None

Example:

Assembly	Machine Code
`sub16 r4,r8,r7`	`101101_01000_00111_00100_00000_100010`
`sub16 r9,r1,r5`	`101101_00001_00101_01001_00000_100010`

Operand Values	Result

`r8 = 0xFF000000`
`r7 = 0x00FFFFFF`
	`R4 = 0xFE01FFF1`
	`c=x, v=x, n=x, z=0`

`r1 = 0xE956E003`
`r5 = 0x4BB4011E`
	`R9 = 0x9DA2DEE5`
	`c=x, v=x, n=x, z=0`

XOR8

31 26	25 21	20 16	15 11	10 6	5 0
0x2C	rs	rt	rd	0	0x26

Format: xor8 rd,rs,rt

Purpose: Vector Logical Exclusive OR 8-bit
Vector do a bitwise exclusive or to two 32-bit registers. Each 32-bit registers represent the four vector byte values.

Description: rd ← rs^rt
rs is exclusive or’ed with rt and placed into rd. Each 32-bit registers represent the four vector byte values. Status flag Zis checked and updated depending on results.

Restrictions:
None

Example:

Assembly	Machine Code
`xor8 r4,r8,r7`	`101100_01000_00111_00100_00000_100110`
`xor8 r9,r1,r5`	`101100_00001_00101_01001_00000_100110`

Operand Values	Result

`r8 = 0xFFFF00B0`
`r7 = 0x0E00FFFF`
	`R4 = 0xF1FFFF4F`
	`c=x, v=x, n=x, z=0`

`r1 = 0x11111111`
`r5 = 0x000F0010`
	`R9 = 0x111E1101`
	`c=x, v=x, n=x, z=0`

XOR16

31 26	25 21	20 16	15 11	10 6	5 0
0x2D	rs	rt	rd	0	0x26

Format: xor16 rd,rs,rt

Purpose: Vector Logical Exclusive OR 8-bit
Vector do a bitwise exclusive or to two 32-bit registers. Each 32-bit registers represent the two vector word values.

Description: rd ← rs^rt
rs is exclusive or'ed with rt and placed into rd. Each 32-bit registers represent the two vector word values. Status flag Z is checked and updated depending on results.

Restrictions:
None

Example:

Assembly	Machine Code
`xor16 r4,r8,r7`	`101101_01000_00111_00100_00000_100110`
`xor16 r9,r1,r5`	`101101_00001_00101_01001_00000_100110`

Operand Values	Result

`r8 = 0xFFFF00B0`
`r7 = 0x0E00FFFF`
	`R4 = 0xF1FFFF4F`
	`c=x, v=x, n=x, z=0`

`r1 = 0x11111111`
`r5 = 0x000F0010`
	`R9 = 0x111E1101`
	`c=x, v=x, n=x, z=0`

III. Verilog Implementation, Design and Verification

A. Top Level - Source Code

```
`timescale 1ns / 1ps
/****************************** C E C S  4 4 0 ******************************
 *
 * File Name:  Top_tb.v
 * Project:    Final_Project
 * Designer:   Thomas Nguyen and Reed Ellison
 * Email:      tholinngu@gmail.com and notwreed@gmail.com
 * Rev. No.:   Version 1.0
 * Rev. Date:  04/010/2019
 *
 * Purpose: Test Fixture to Implement our Top Level Design.
 * Instantiates CPU, IO Memory, and Data Memory Modules.
 * Interconnects the Memory Modules to the CPU to allow for
 * input from both memory modles. Data can also be transmitted
 * to both Memory Modules in order to access certain locations
 * of memory within the memory modules.
 *
 * Notes:
 ****************************************************************************/
module Top_tb;

       // Inputs
       reg clk;                //Clock
       reg rst;                //Reset

       // Outputs
   wire intr;              //Interrupt Request
   wire inta;              //Interrupt Acknowledge
   wire dm_cs;             //Data Memory Chip Select
   wire dm_rd;             //Data Memory Read Select
   wire dm_wr;             //Data Memory Write Select
   wire io_cs;             //IO Memory Chip Select
   wire io_rd;             //IO Memory Read Select
   wire io_wr;             //IO Memory Write Select
   wire [31:0] mem_to_DY;  //Wire from Memory Output to DY Input of CPU
   wire [31:0] madr;       //Memory Address
   wire [31:0] idp;        //IDP Output

   CPU          cpu(.clk(clk),       .rst(rst),             .intr(intr),
                    .inta(inta),     .dm_cs(dm_cs),         .dm_rd(dm_rd),
                    .dm_wr(dm_wr),   .io_cs(io_cs),         .io_rd(io_rd),
                    .io_wr(io_wr),   .dMemtoDY(mem_to_DY),
                    .madr(madr),     .idp(idp));

   Data_Memory dMEM(.clk(clk),       .Addr(madr[11:0]),     .D_in(idp),
                    .dm_cs(dm_cs),   .dm_wr(dm_wr),         .dm_rd(dm_rd),
                    .D_out(mem_to_DY));

   IO_MEM        io(.clk(clk),       .cs(io_cs),            .wr(io_wr),
                    .rd(io_rd),      .inta(inta),           .addr(madr[11:0]),
                    .in(idp),        .intr(intr),          .out(mem_to_DY));

   //Initialize Clock
   always #5 clk = ~clk;

       initial begin
              //Initialize Inputs
              {clk, rst} = 2'b0_0;

      //Format how time is displayed and sets it to display in nanoseconds.
      $timeformat(-9, 1, " ns", 9);

      //Load contents of a .dat file into the Instruction Unit Memory
      //New Enhancements
      $readmemh("iMem14_Sp19_w_isr_commented.dat", Top_tb.cpu.IU.IMEM.mem);

      //Load contents of a .dat file into the data memory
      $readmemh("dMem14_Sp19.dat", Top_tb.dMEM.mem);

      //Reset System
      @(negedge clk)
         rst = 1;
      @(negedge clk)
         rst = 0;

       end

endmodule
```

B. Verilog Modules

```
`timescale 1ns / 1ps
/****************************** C E C S  4 4 0 ******************************
 *
 * File Name:  CPU.v
 * Project:    Final_Project
 * Designer:   Thomas Nguyen and Reed Ellison
 * Email:      tholinngu@gmail.com and notwreed@gmail.com
 * Rev. No.:   Version 1.0
 * Rev. Date:  04/05/2019
 *
 * Purpose: Central Processing Unit Module that instantiates
 * the Instruction Unit, Integer Datapath, and Control Unit.
 * Input to the CPU comes from the Data Memory and Input/Output
 * Memory Modules.
 *
 * Notes:
 ****************************************************************************/
module CPU(clk, rst, intr, inta, dm_cs, dm_rd, dm_wr,
          io_cs, io_rd, io_wr, dMemtoDY, madr, idp);

   input        clk, rst, intr;     //Clock, Reset, and Interrupt Request

   input [31:0] dMemtoDY;           //Output of data memory module

   output       inta;               //Interrupt Acknowledge

   output dm_cs, dm_rd, dm_wr,      //Data Memory Chip Select, Read, and Write
          io_cs, io_rd, io_wr;      //IO Memory Chip Select, Read, and Write

   output [31:0] madr, idp;         //Memory Address and IDP Output

   wire        PC_Ld, PC_Inc;       //PC Load and Increment Signals
   wire  [1:0] PC_Sel;              //PC Select Signal
   wire        IR_Ld;               //IR Load Signal
   wire        sp_sel, s_sel;       //Stack Pointer and S Data Signals

   wire        IM_Cs,IM_Wr,IM_Rd;   //Instruction Memory and Data memory's

   wire        D_En, HILO_ld;       //Write Enable for Registers, HILO Load
   wire  [1:0] DA_Sel, T_Sel;       //MUX Select for IDP and for T Address
   wire  [2:0] Y_Sel;               //MUX Select for input to ALU_OUT Reg

   wire  [4:0] FS;                  //Function Select Value

   wire        C, V, N, Z;          //Status Flags
   wire [31:0] ALU_OUT, D_OUT;      //Outputs from IDP
   wire [31:0] PC_Out;              //Program Counter Out
   wire [31:0] IR_Out;              //Instruction Register Out
   wire [31:0] SE_16;               //Sign Extended 16-bits
   wire [4:0]  sp_outFlags, sp_inFlags; //Stack Pointer Flags
   wire [1:0]  simd;                //SIMD mode select

   assign madr = ALU_OUT;           //Memory Address = ALU_OUT

   assign idp = D_OUT;              //IDP Output = D_OUT

   //Instantiate Instruction Unit
   Instruction_Unit   IU(.clk(clk),           .rst(rst),            .PC_Ld(PC_Ld),
                         .PC_Inc(PC_Inc),     .IM_Cs(IM_Cs),        .IM_Wr(IM_Wr),
                         .IM_Rd(IM_Rd),       .IR_Ld(IR_Ld),        .PC_In(ALU_OUT),
                         .PC_Out(PC_Out),     .IR_Out(IR_Out),      .SE_16(SE_16),
                         .PC_Sel(PC_Sel));

   //Instantiate Integer Datapath
   Integer_Datapath  IDP(.clk(clk),             .reset(rst),   .sp_sel(sp_sel),
                         .s_sel(s_sel),         .D_En(D_En),   .HILO_ld(HILO_ld),
                         .T_Sel(T_Sel),         .Y_Sel(Y_Sel), .PC_in(PC_Out),
                         .DY(dMemtoDY),         .DT(SE_16),    .C(C),
                         .V(V),                 .N(N),
.sp_inFlags(sp_inFlags),
                         .ALU_OUT(ALU_OUT),     .D_OUT(D_OUT), .DA_Sel(DA_Sel),
                         .Z(Z),                 .sp_outFlags(sp_outFlags),
                         .FS(FS),               .shamt(IR_Out[10:6]),
                         .S_Addr(IR_Out[25:21]),.T_Addr(IR_Out[20:16]),
                         .D_Addr(IR_Out[15:11]),.simd_sel(simd));

   //Instantiate Control Unit
   Control_Unit       CU(.clk(clk),        .rst(rst),        .intr(intr),
                         .c(C),            .n(N),            .z(Z),     .v(V),
                         .IR(IR_Out),      .int_ack(inta),   .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),      .DA_sel(DA_Sel),  .T_sel(T_Sel),
                         .sp_sel(sp_sel),  .s_sel(s_sel),    .dm_cs(dm_cs),
                         .HILO_ld(HILO_ld),.Y_sel(Y_Sel),
                         .dm_rd(dm_rd),    .dm_wr(dm_wr),
                         .io_cs(io_cs),    .sp_inFlags(sp_inFlags),
                         .sp_outFlags(sp_outFlags),
                         .io_rd(io_rd),    .io_wr(io_wr), .FS(FS), .simd_sel(simd));


endmodule
```

```
`timescale 1ns / 1ps
/****************************** C E C S  4 4 0 ******************************
 *
 * File Name:  Data_Memory.v
 * Project:    Lab_Assignment_5
 * Designer:   Thomas Nguyen
 * Email:      tholinngu@gmail.com
 * Rev. No.:   Version 1.1
 * Rev. Date:  03/09/2019
 *
 * Purpose: A 4096 x 8 array register.
 * Memory module that recieves a 32 bit input for address and a 32 bit input for
 * its data input. This module has a 32 bit output for the data at the specificed
 * address. There are three status signals that indicate what the memory module
 * should do. DM_CS flag functions as a chip select that activates the Memory
 * Module. The DM_WR flag allows the memory module to be written to given the
 * input address and data. The DM_RD flag allows the module to read the data
 * at the given address and outputs the data at that memory location.
 *
 * Notes:
 ****************************************************************************/
module Data_Memory(clk, Addr, D_in, dm_cs, dm_wr, dm_rd, D_out);

   input             clk, dm_cs, dm_wr, dm_rd;
   input      [11:0] Addr;        //Starting Address for memory access
   input      [31:0] D_in;        //Data Input
   output     [31:0] D_out;       //Data Output

   reg         [7:0] mem [4095:0];//4096 by 8 Memory Register

   //Sequential Block. When Chip Select and Write flags are enable
   //Write to the Memory Register starting at the given address for 4 bytes
   always@(posedge clk)
      if(dm_cs == 1'b1 && dm_wr == 1'b1)
         {mem[Addr], mem[Addr + 1], mem[Addr + 2], mem[Addr + 3]} <= D_in;
      else
         {mem[Addr], mem[Addr + 1], mem[Addr + 2], mem[Addr + 3]} <=
         {mem[Addr], mem[Addr + 1], mem[Addr + 2], mem[Addr + 3]};

   //Continuous Assignment that assigns the 32 bit output based on cs and rd flags
   //The data at the four memory locations specified by the memory address is output
   //Else the output is High Impedence
   assign D_out = (dm_cs == 1'b1 && dm_rd == 1'b1) ?
                  {mem[Addr], mem[Addr + 1], mem[Addr + 2], mem[Addr + 3]} : 32'bZ;

endmodule
```

```
`timescale 1ns / 1ps
/****************************** C E C S  4 4 0 ******************************
 *
 * File Name:  IO_MEM.v
 * Project:    Final_Project
 * Designer:   Thomas Nguyen and Reed Ellison
 * Email:      tholinngu@gmail.com and notwreed@gmail.com
 * Rev. No.:   Version 1.0
 * Rev. Date:  04/10/2019
 *
 * Purpose: A 4096 x 8 array register.
 * Input/Output module that recieves a 32 bit input for address and a 32 bit input for
 * its data input. This module has a 32 bit output for the data at the specificed
 * address. There are three status signals that indicate what the memory module
 * should do. IO_CS flag functions as a chip select that activates the Memory
 * Module. The IO_WR flag allows the memory module to be written to given the
 * input address and data. The IO_RD flag allows the module to read the data
 * at the given address and outputs the data at that memory location.
 * The IO Memory Module has additional Interrupt Signals to allow for
 * the CPU to be interrupted and then execute IO Instructions
 *
 * Notes:
 *****************************************************************************/
module IO_MEM(clk,cs,wr,rd,inta,addr,in,intr,out);

   input        clk, inta; //Clock, Interrupt Acknowledge
   input        cs, wr, rd;//Chip Select, Write Select, Read Select

   input [11:0] addr;      //Address for Accessing Memory

   input [31:0] in;        //Input Data to Memory Module

   output intr;            //Interrupt Request
   reg intr;

   output [31:0] out;      //Output of Memory Module
   wire [31:0] out;

   reg [7:0] mem [0:4095]; //Array for Memory

   //Initial Block to Have the IO Subsystem
   //Output an Interrupt Request after
   //a Discrete Amount of Time to Check Interrupt
   //Functionality is accurate
   initial
   begin
      intr = 0;
      #200;
      intr = 1;
      @(posedge inta)
```

```
         intr = 0;
   end

   //Output of IO Mem Module is based on address input only when
   //Chip Select and RD Signals are active and WR is inactive
   //Otherwise, the output is High Impedance
   assign out = (cs & !wr & rd)? {mem[addr+0], mem[addr+1],
                                  mem[addr+2], mem[addr+3]}:
                                  32'hz;

   //Input Data to IO Memory when CS and WR are active
   //Otherwise, Data at the Addresses remain the same
   always@ (posedge clk)
      begin
         if(cs & wr & !rd)
            {mem[addr+0], mem[addr+1],
             mem[addr+2], mem[addr+3]} <= in;
         else
            {mem[addr+0], mem[addr+1],
             mem[addr+2], mem[addr+3]} <=
            {mem[addr+0], mem[addr+1],
             mem[addr+2], mem[addr+3]};
      end


endmodule
```

```
`timescale 1ns / 1ps
/****************************** C E C S  4 4 0 ******************************
 *
 * File Name:  Instruction_Unit.v
 * Project:    Lab_Assignment_6
 * Designer:   Reed  Ellison
 * Email:      Reed.Ellison@student.csulb.edu
 * Rev. No.:   Version 1.2
 * Rev. Date:  03/19/2019
 *
 * Purpose: Instantiate PC register which holds the current PC value.
 * The PC value can be incremented or loaded based on its control signals.
 * The PC Output access the Instruction Memory which transfers the selected
 * 32'bit instruction and transfers it to the IR Register. The IR Register
 * value is then output from the Instruction Unit in addition to a sign
 * extended value of the first 16 bits from the IR Output.
 *
 * Notes: 3/17/2019 - PC Mux was added to for PC+4, jumps, and branches
 *
 ***************************************************************************/
module Instruction_Unit(clk, rst, PC_Ld, PC_Inc, IM_Cs, IM_Wr, IM_Rd, IR_Ld,
                        PC_In, PC_Out, IR_Out, SE_16, PC_Sel);

   //Clock, reset, PC load, PC Incrementer, Instruction Memory Chip Select
   //Instruction Memory Write Enable, Instruction Memory Read Enable
   //Instruction Register Load
   input         clk, rst, PC_Ld, PC_Inc, IM_Cs, IM_Wr, IM_Rd, IR_Ld;

   input   [1:0] PC_Sel;

   //32'bit input to PC_Mux to do PC + 4.
   input  [31:0] PC_In;

   //Program Counter Output, Instruction Register Output
   //IR Output 16 bits extended to 32 bits
   output [31:0] PC_Out, IR_Out, SE_16;

   wire   [31:0] SE_16;
   wire   [31:0] IMEM_Out;

   //Output of PC Mux and into PC register
   wire   [31:0] PC_M_Out;

   //Program Counter
   PC            pc(.clk(clk),     .rst(rst),     .ld(PC_Ld),
                    .inc(PC_Inc),  .PC_In(PC_M_Out), .PC_Out(PC_Out));

   //Instruction Memory
   Data_Memory IMEM(.clk(clk),     .Addr(PC_Out[11:0]), .D_in(32'h0),
                    .dm_cs(IM_Cs), .dm_wr(IM_Wr),       .dm_rd(IM_Rd),
                    .D_out(IMEM_Out));

   //32'bit Loadable Register
   LD_reg32      IR(.clk(clk), .reset(rst), .ld(IR_Ld), .d(IMEM_Out),
                    .q(IR_Out));

   //Assign Sign Extended IR Output
   assign SE_16 = {{16{IR_Out[15]}}, IR_Out[15:0]};

   //Assign PC Mux output
   //PC+4:   PC_Sel = 0 (default)
   //Jump:   PC_Sel = 1
   //Branch: PC_sel = 2
   //JR:     PC_sel = 3
   assign PC_M_Out = (PC_Sel == 2'b01) ? {PC_Out[31:28], IR_Out[25:0], 2'b00}:
                     (PC_Sel == 2'b10) ? {PC_Out + {SE_16[29:0], 2'b00}}:
                     (PC_Sel == 2'b11) ? PC_In:
                                         PC_Out;


endmodule
```

```
`timescale 1ns / 1ps
/****************************** C E C S  4 4 0 ******************************
 *
 * File Name:  Integer_Datapath.v
 * Project:    Lab_Assignment_6
 * Designer:   Thomas Nguyen
 * Email:      tholinngu@gmail.com
 * Rev. No.:   Version 1.3
 * Rev. Date:  03/18/2019
 *
 * Purpose: This module is the hardware that does all the required operations
 * for a processor. It contains and connects the ALU to an array of registers
 * allowing operations to modify array registers and memory. Inside of
 * this are also 2 32-bit registers called HI and LO that holds the upper
 * and lower 32-bits of a 64-bit result of the MULT and DIV instruction.
 * When the two instructions are selected, the HILO_ld is set in order to
 * load the the registers. ALU_OUT contains the values inside an array register.
 * DY and DT are input from memory and PC_in is a value from the PC.
 * The IDP has 4 Pipelining registers: RS, RT, ALU_PIP and DIN.
 *
 * Notes: 3/01/2019 - This revision added the Pipeline Registers to the IDP
 *        3/07/2019 - Added a mux to select the destination address depending
 *                    on instruction type. When I-Type, DA_OUT = T_Addr
 *                    Else if R-Type, DA_OUT = D_Addr.
 *        3/17/2019 - Expanded the destination address mux to a 4 to 1
 *                    hex 2 and 3 have the value of 5'h1F and 5'h1D
 *        4/08/2019 - Added the Barrel shifter to do shifts
 *        4/17/2019 - Expanded the T_sel and added sp_outFlags and sp_inFlags,
 *                    sp_sel and s_sel to do stack related instructions.
 ****************************************************************************/
module Integer_Datapath(clk, reset, sp_sel, s_sel, D_En, HILO_ld, T_Sel, Y_Sel,
                        DA_Sel, D_Addr, S_Addr, T_Addr, FS, shamt, sp_inFlags,
                        PC_in, DY, DT, C, V, N, Z, sp_outFlags, ALU_OUT, D_OUT,
                        simd_sel);

   input         clk, reset;     //Clock, Reset
   input         sp_sel, s_sel;  //Stack Pointer Select and S Data Select
   input         D_En, HILO_ld;  //Write enable, HILO load

   //Selector for Destination address depending on instruction type
   input   [1:0] DA_Sel, T_Sel, simd_sel;

   //Address of Registers
   input   [4:0] D_Addr, S_Addr, T_Addr;

   input   [2:0] Y_Sel;          //Mux Select
   input   [4:0] FS, shamt;      //Function Select
   input   [4:0] sp_inFlags;     //Stack Pointer Input Flags
   input  [31:0] PC_in;          //Value from PC
   input  [31:0] DY;             //Value from dMem
   input  [31:0] DT;             //Value from iMem

   output        C, V, N, Z;     //Status Flag Output
   output  [4:0] sp_outFlags;    //Stack Pointer Out Flags
   output [31:0] ALU_OUT, D_OUT; //Output of IDP

   wire          c, v, n, z;     //Output status flag wires from ALU_32
   wire          simdz;
   wire    [4:0] DA_OUT;         //Output from Destination Address MUX
   wire   [31:0] RS_Out;         //Output S from regfile32
   wire   [31:0] DY_Out;         //Output from D_in Pipeline register
   wire   [31:0] ALU_wire;       //Output from ALU_PIP Pipeline register
   wire   [31:0] HI_out, LO_out; //Output from HI and LO register
   wire   [31:0] Y_hi, Y_lo;     //Output from ALU. Upper and Lower 32 bit
   wire   [31:0] Y_hi_std, Y_lo_std;
   wire   [31:0] Y_hi_simd;      //Output from V_ALU. Upper and Lower 32 bit
   wire   [31:0] Y_lo_simd;
   wire   [31:0] Reg_S, Reg_T;   //Output from Register File
   wire   [31:0] T_MUX;          //Output from multiplexor

   wire [4:0] sp_muxOut;         //Stack Pointer Mux Output Wire
   wire [31:0] s_muxOut;         //S Mux Output Wire

   //Arithmetic Logical Unit
   ALU_32       uut0(.S(s_muxOut),  .T(D_OUT),   .FS(FS), .shamt(shamt),
                     .y_hi(Y_hi_std), .y_lo(Y_lo_std),
                     .c(c), .v(v), .n(n), .z(z));

   V_ALU        SIMD(.simd_sel(simd_sel), .S(s_muxOut), .T(D_OUT), .FS(FS),
                     .shamt(shamt), .y_hi(Y_hi_simd), .y_lo(Y_lo_simd), .z(simdz));

   //32x32 Register
   regfile32    uut1(.clk(clk),       .reset(reset),   .D_En(D_En), .D_Addr(DA_OUT),
                     .S_Addr(sp_muxOut), .T_Addr(T_Addr), .D(ALU_OUT), .S(Reg_S),
                     .T(Reg_T));

   //HI and LO load registers that are used when MULTI or DIV operations are used
   LD_reg32       HI(.clk(clk), .reset(reset), .ld(HILO_ld), .d(Y_hi), .q(HI_out));
   LD_reg32       LO(.clk(clk), .reset(reset), .ld(HILO_ld), .d(Y_lo), .q(LO_out));

   //32-bit pipeline registers that take output operands of the regfile32's S and T
   REG32          RS(.clk(clk), .reset(reset), .D(Reg_S), .Q(RS_Out));
   REG32          RT(.clk(clk), .reset(reset), .D(T_MUX), .Q(D_OUT));

   //32-bit pipeline register that loads the y_lo value from the ALU32
   REG32     ALU_PIP(.clk(clk), .reset(reset), .D(Y_lo), .Q(ALU_wire));

   //32-bit pipeline register that loads the value from memory (DY)
   REG32         DIN(.clk(clk), .reset(reset), .D(DY),   .Q(DY_Out));
```

```
   //A 2 to 1 multiplexor that sets T_MUX to DT from iMem or T from the regfile
   assign T_MUX   = (T_Sel == 2'b01) ? DT    :
                    (T_Sel == 2'b10) ? PC_in :
                    (T_Sel == 2'b11) ? {27'b0, sp_inFlags} :
                                       Reg_T;

   //A 5 to 1 multiplexor that sets ALU_OUT to HI LO registers, ALU, DY
   //or PC_in. Defaults is Y_lo of the ALU
   assign ALU_OUT = (Y_Sel == 3'h1) ? HI_out:
                    (Y_Sel == 3'h2) ? LO_out:
                    (Y_Sel == 3'h3) ? DY_Out:
                    (Y_Sel == 3'h4) ? PC_in:
                                      ALU_wire;

   //MUX Output allows for the S_Addr to Reg File to either be the
   //passed in S_Addr or the Stack Pointer Register Location(5'h1D)
   assign sp_muxOut = (sp_sel) ? 5'h1D : S_Addr;

   //Select the Input to the S input of the ALU
   //If s_sel is active, input the ALU_OUT Value
   //Otherwise, input the value from the RS_Out Register
   assign s_muxOut = (s_sel) ? ALU_OUT : RS_Out;

   //Captures Stack Pointer Output Flags from DY_Out[4:0]
   assign sp_outFlags = DY_Out[4:0];

   //When I-Type instruction is being used, DA_Sel = 0 so that the destination
   //address will use the T_Addr. If R-type then D_Addr will be use (DA_Sel = 1)
   //When in RESET state of the MCU, DA_Sel == 3 and when in INTR_1 state it is 2
   assign DA_OUT  = (DA_Sel == 2'h1) ? D_Addr:
                    (DA_Sel == 2'h2) ? 5'h1F:
                    (DA_Sel == 2'h3) ? 5'h1D:
                                       T_Addr;

   //Select Status flags and Y_hi/Y_lo data based on simd mode. 0 is standard
   assign {C, V, N, Z} = (simd_sel == 2'b0) ? {c, v, n, z} :
                                              {1'bx, 1'bx, 1'bx, simdz};

   assign {Y_hi, Y_lo} = (simd_sel == 2'b0) ? {Y_hi_std, Y_lo_std} :
                                              {Y_hi_simd, Y_lo_simd};

endmodule
```

```
`timescale 1ns / 1ps
/****************************** C E C S  4 4 0 ******************************
 *
 * File Name:  Instruction_Unit.v
 * Project:    Lab_Assignment_6
 * Designer:   Reed  Ellison
 * Email:      Reed.Ellison@student.csulb.edu
 * Rev. No.:   Version 1.5
 * Rev. Date:  04/17/2019
 *
 * Purpose: State machine implementing the MIPS Control Unit (MCU) for the
 * major cycles of fetch, decode and various execute instructions and write
 * back states.
 *
 * Notes: 4/08/2019 - Added lui, ori, sw, addi and shift instructions.
 *        4/13/2019 - Added bne, beq, lw, ori, mflo, mfhi, mult, div instructions
 *        4/14/2019 - Added xor, xori, slt, slti, jr instructions
 *        4/15/2019 - Added blez, bgtz, setie instructions
 *        4/17/2019 - Expanded the T_sel and added sp_outFlags and sp_inFlags,
 *                    sp_sel and s_sel to do stack related instructions.
 *                    Added reti and expanded interrupt states.
 *************************************************************************/
module Control_Unit(clk, rst, intr, c, n, z, v, IR, int_ack, pc_sel, pc_ld,
                    pc_inc, ir_ld, im_cs, im_rd, im_wr, D_En, DA_sel, T_sel,
                    sp_sel, s_sel, HILO_ld, Y_sel, sp_inFlags, sp_outFlags,
                    dm_cs, dm_rd, dm_wr, io_cs, io_rd, io_wr, FS, simd_sel);

   input             clk, rst, intr;//Clock, Reset, and Interrupt Request
   input             c, n, z, v;    //Flags
   input      [4:0]  sp_inFlags;    //Stack Pointer In Flags
   input      [31:0] IR;            //Instruction Register

   //Outputs
   output reg        pc_ld, pc_inc; //Program Counter load and increment
   output reg  [1:0] pc_sel, DA_sel;//Program Counter Input select
                                    //and Dest. Addr Select
   output reg        ir_ld, im_cs,  //Instruction Reg. load and
                     im_rd, im_wr;  //Instruction Memory Chip Select and
                                    //Read and Write signals
   output reg        D_En, HILO_ld; //Write Enable, T Select and Hi Lo Load
   output reg  [1:0] T_sel;         //T Value Select
   output reg  [2:0] Y_sel;         //ALU Out Select
   output reg        dm_cs, dm_rd, dm_wr;//Data Memory chip select, read and write
   output reg        io_cs, io_rd, io_wr;//IO Memory chip select, read and write
   output reg  [4:0] FS;            //Function Select
   output reg  [4:0] sp_outFlags;   //Stack Pointer Output Flags
   output reg        s_sel, sp_sel; //S_Sel and Stack Pointer Select
   output reg  [1:0] simd_sel;      //SIMD mode select. 0=std, 1=8bit, 2=16bit
   output       int_ack;            //interrupt acknowledge
   reg          int_ack;
```

```
integer i;

//States of control unit
parameter
   RESET  = 00,  FETCH      = 01,  DECODE   = 02,  ADD    = 10,
   ADDU   = 11,  AND        = 12,  ORI      = 20,  JR     = 8,
   JR2    = 408, BEQ        = 24,  BEQ2     = 424,
   BNE    = 25,  BNE2       = 425, SRL      = 50,
   ADDI   = 51,  SRA        = 52,  SLT      = 53,  SLL    = 54,
   SLTI   = 55,  LW_2       = 57,  MULT     = 58,  MFLO   = 59,
   MFHI   = 60,  SLTU       = 61,  OR       = 62,  NOR    = 63,
   XOR    = 64,  DIV        = 65,  XORI     = 66,  SLTIU  = 67,
   ANDI   = 68,  SUB        = 69,  BLEZ     = 70,  BLEZ_2 = 71,
   BGTZ   = 72,  BGTZ_2     = 73,  SETIE    = 74,  INPUT  = 75,
   INPUT_2 = 76, OUTPUT     = 77,  OUTPUT_2 = 78,
   RETI    = 79, RETI_2     = 80,  RETI_3   = 81,
   RETI_4  = 82, RETI_5     = 83,  RETI_6   = 84,
   ROTL    = 88, ROTR       = 89,  CLR      = 90,
   CLR_2   = 91, NOP        = 92,  MOV      = 93,
   MOV_2   = 94, PUSH       = 95,  PUSH_2   = 96,
   PUSH_3  = 97, PUSH_4     = 98,  POP      = 99,
   POP_2   = 100, POP_3     = 101, POP_4    = 102,
   POP_5   = 103, SUBU      = 104,
   LUI    = 21,  LW         = 22,  SW     = 23,
   WB_ALU = 30,  WB_IMM     = 31,  WB_DIN = 32,
   WB_HI  = 33,  WB_LO      = 34,  WB_MEM = 35,
   J      = 37,  JAL        = 38,

   SIMD8  = 300, SIMD16     = 301, WB_ALU_SIMD = 302,
   ADD_SIMD = 303, SUB_SIMD = 304, MULT_SIMD = 305,
   DIV_SIMD = 305, AND_SIMD = 306, OR_SIMD = 307,
   XOR_SIMD = 308, NOR_SIMD = 309, SLL_SIMD = 310,
   SRL_SIMD = 311, SRA_SIMD = 312, ROTL_SIMD = 313,
   ROTR_SIMD = 314,

   INTR_1 = 501, INTR_2     = 502, INTR_3 = 503,
   INTR_4 = 504, INTR_5     = 505, INTR_6 = 506,
   BREAK  = 510, ILLEGAL_OP = 511;

//ALU Opcodes for Function Select
parameter
   pass_s_  = 5'h00, pass_t_ = 5'h01, add_  = 5'h02,
   addu_    = 5'h03, sub_    = 5'h04, subu_ = 5'h05,
   slt_     = 5'h06, sltu_   = 5'h07, and_  = 5'h08,
   or_      = 5'h09, xor_    = 5'h0A, nor_  = 5'h0B,
   srl_     = 5'h0C, sra_    = 5'h0D, sll_  = 5'h0E,
   inc_     = 5'h0F, inc4_   = 5'h10, dec_  = 5'h11,
   dec4_    = 5'h12, zeros_  = 5'h13, ones_ = 5'h14,
   sp_init_ = 5'h15, andi_   = 5'h16, ori_  = 5'h17,
   lui_     = 5'h18, xori_   = 5'h19, rotl_ = 5'h1A,
   rotr_    = 5'h1B, clr_    = 5'h1C, mul_   = 5'h1E,
   div_ = 5'h1F;

reg [8:0] state;

//Flags
reg   psi, psc, psv, psn, psz;
reg   nsi, nsc, nsv, nsn, nsz;

always@(posedge clk, posedge rst)
   if(rst)
      {psi, psc, psv, psn, psz} <= 5'b0;
   else begin
      {psi, psc, psv, psn, psz} <= {nsi, nsc, nsv, nsn, nsz};
      end

always @(posedge clk, posedge rst)
   if(rst)
      begin
         //control word assignments for "deasserting" 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_00_0_000;
         {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
         {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
         {sp_sel, s_sel}                       = 2'b0_0;
         int_ack = 1'b0;                    FS = sp_init_; //Gets value 0x3FC
         simd_sel = 2'b00;
         #1 {nsi, nsc, nsv, nsn, nsz} = 5'b0;
         state   = RESET;
      end
   else
      case (state)
         FETCH:
            if(int_ack == 0 && (intr == 1 && psi == 1))
               begin /*new interrupt pending, prepare for ISR*/
                  //control word assignments for "deasserting" 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_00_0_000;
                  {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
                  {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
                  {sp_sel, s_sel}                       = 2'b0_0;
                  int_ack = 1'b0;                    FS = 5'h0;
                  simd_sel = 2'b00;
                  #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, psn, psz};
                  state   = INTR_1;
               end
            else
               begin /*no new interrupt pending, fetch and instruction*/
                  if((int_ack == 1 && intr == 0) || (psi == 1 && intr == 0))
                     int_ack = 1'b0;
                  // control word assignemtns: IR <-- iM[PC], PC <-- PC + 4
                  @(negedge clk)
                  {pc_sel, pc_ld, pc_inc, ir_ld}        = 5'b00_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_00_00_0_000;
                  {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
                  {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
                  {sp_sel, s_sel}                       = 2'b0_0;
                  int_ack = 1'b0;                    FS = 5'h0;
                  simd_sel = 2'b00;
                  #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, psn, psz};
                  state   = DECODE;
               end
         RESET:
            begin
               // control word assignments: $sp <-- ALU_OUT(32'h3FC)
               {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_00_0_000;
               {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
               {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
               {sp_sel, s_sel}                       = 2'b0_0;
               int_ack = 1'b0;                    FS = 5'h0;
               simd_sel = 2'b00;
               #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, psn, psz};
               state = FETCH;
            end
         DECODE:
            begin
               @(negedge clk)
               if(IR[31:26] == 6'h0) // check for MIPS format
                  begin // it is an R-type format
                        // control word assignements: RS <-- 4rs,
                        //RT <-- $rt (default)
                     {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_00_0_000;
                     {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
                     {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
                     {sp_sel, s_sel}                       = 2'b0_0;
                     int_ack = 1'b0;                    FS = 5'h0;
                     simd_sel = 2'b00;
                     #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, psn, psz};
                     case (IR[5:0])
                        6'h00  : state = SLL;
                  6'h02  : state = SRL;
                  6'h03  : state = SRA;
                  6'h04  : state = ROTL;
                  6'h05  : state = ROTR;
                  6'h06  : state = CLR;
                  6'h07  : state = NOP;
                  6'h09  : state = MOV;
                  6'h08  : state = JR;
                  6'h10  : state = MFHI;
                  6'h12  : state = MFLO;
                  6'h13  : state = PUSH;
                  6'h14  : state = POP;
                  6'h0D  : state = BREAK;
                  6'h18  : state = MULT;
                  6'h1A  : state = DIV;
                  6'h1F  : state = SETIE;
                  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;
                  default: state = ILLEGAL_OP;
               endcase
            end  // end of if statement for R-type format
         else
            begin // it is an I-type or J-type format
                  // control word assignments: RS <-- $rs,
                  //RT <-- DT(se_16)
               {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_01_0_000;
               {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
               {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
               {sp_sel, s_sel}                       = 2'b0_0;
               int_ack = 1'b0;                    FS = 5'h0;
               simd_sel = 2'b00;
               #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, psn, psz};
               case (IR[31:26])
                  6'h02  : state = J;
                  6'h03  : state = JAL;
                  6'h04  : state = BEQ;
                  6'h05  : state = BNE;
                  6'h06  : state = BLEZ;
                  6'h07  : state = BGTZ;
                  6'h08  : state = ADDI;
               6'h0A  : state = SLTI;
               6'h0B  : state = SLTIU;
               6'h0C  : state = ANDI;
               6'h0D  : state = ORI;
               6'h0E  : state = XORI;
               6'h0F  : state = LUI;
               6'h1C  : state = INPUT;
               6'h1D  : state = OUTPUT;
               6'h1E  : state = RETI;
               6'h23  : state = LW;
               6'h2B  : state = SW;
               6'h2C  : state = SIMD8;
               6'h2D  : state = SIMD16;
               default: state = ILLEGAL_OP;
            endcase
            // Case of Branches
            //    if T_sel = 0, RT <- $rt
            //    IR[15:0] will be used
            if(state == BEQ || state == BNE ||
               state == BGTZ || state == BLEZ)
               T_sel = 2'b0;
            else
               T_sel = 2'b1;

            // Case when SIMD8 or SIMD16. Checks again for IR[5:0]
            if(state == SIMD8 || state == SIMD16) begin
               if(state == SIMD8)
                  simd_sel = 2'b01;
               else
                  simd_sel = 2'b10;
               T_sel = 2'b0;
               case(IR[5:0])
                  6'h00  : state = SLL_SIMD;
                  6'h02  : state = SRL_SIMD;
                  6'h03  : state = SRA_SIMD;
                  6'h04  : state = ROTL_SIMD;
                  6'h05  : state = ROTR_SIMD;
                  6'h18  : state = MULT_SIMD;
                  6'h1A  : state = DIV_SIMD;
                  6'h20  : state = ADD_SIMD;
                  6'h22  : state = SUB_SIMD;
                  6'h24  : state = AND_SIMD;
                  6'h25  : state = OR_SIMD;
                  6'h26  : state = XOR_SIMD;
                  6'h27  : state = NOR_SIMD;
                  default: state = ILLEGAL_OP;
               endcase
            end

         end   // end of else statement for I-type or J-type formats
```

```
   end   // end of DECODE
ADD:
   begin
      // control word assignments: ALU_OUT <-- RS($rs) + RT($rt)
      @(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_00_0_000;
      {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
      {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
      int_ack = 1'b0;                    FS = add_;
      {sp_sel, s_sel}                       = 2'b0_0;
      #1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};
      state   = WB_ALU;
   end
 SUB:
   begin
      // control word assignments: ALU_OUT <-- RS($rs) - RT($rt)
      @(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_00_0_000;
      {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
      {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
      int_ack = 1'b0;                    FS = sub_;
      {sp_sel, s_sel}                       = 2'b0_0;
      #1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};
      state   = WB_ALU;
   end
ADDU:
   begin
      // control word assignments: ALU_OUT <-- RS($rs) + RT($rt)
      @(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_00_0_000;
      {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
      {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
      int_ack = 1'b0;                    FS = addu_;
      {sp_sel, s_sel}                       = 2'b0_0;
      #1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};
      state   = WB_ALU;
   end
 SUBU:
   begin
      // control word assignments: ALU_OUT <-- RS($rs) - RT($rt)
      @(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_00_0_000;
      {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
      {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
      int_ack = 1'b0;                    FS = subu_;
      {sp_sel, s_sel}                       = 2'b0_0;
      #1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};
      state   = WB_ALU;
   end
CLR:
   begin
      // control word assignments: ALU_OUT <-- 0x0000
      @(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_01_00_0_000;
      {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
      {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
      int_ack = 1'b0;                    FS = zeros_;
      {sp_sel, s_sel}                       = 2'b0_0;
      #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, psn, psz};
      state   = CLR_2;
   end
CLR_2:
   begin
      // control word assignments: RT($rt) <-- ALU_OUT
      @(negedge clk)
      {pc_sel, pc_ld, pc_inc, ir_ld}        = 5'b00_0_0_0;
      {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
      {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b1_00_00_0_000;
      {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
      {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
      int_ack = 1'b0;                    FS = 0;
      {sp_sel, s_sel}                       = 2'b0_0;
      #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, psn, psz};
      state   = FETCH;
   end
MOV:
   begin
      // control word assignments: ALU_OUT <-- RS($rs)
      @(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_00_0_000;
      {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
      {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
      int_ack = 1'b0;                    FS = 0;
      {sp_sel, s_sel}                       = 2'b0_0;
      #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, psn, psz};
      state   = MOV_2;
   end
MOV_2:
```

```
               begin
                  // control word assignments: RT($rt) <-- ALU_OUT
                  @(negedge clk)
                  {pc_sel, pc_ld, pc_inc, ir_ld}        = 5'b00_0_0_0;
                  {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
                  {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b1_00_00_0_000;
                  {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
                  {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
                  int_ack = 1'b0;                    FS = 0;
                  {sp_sel, s_sel}                       = 2'b0_0;
                  #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, psn, psz};
                  state   = FETCH;
               end
            XOR:
               begin
                  // control word assignments: ALU_OUT <-- RS($rs) ^ {16'h0,
RT($rt[15:0])}
                  @(negedge clk)
                  {pc_sel, pc_ld, pc_inc, ir_ld}        = 5'b00_0_0_0;
                  {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
                  {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_00_00_0_000;
                  {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
                  {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
                  int_ack = 1'b0;                    FS = xor_;
                  {sp_sel, s_sel}                       = 2'b0_0;
                  #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, n, z};
                  state   = WB_ALU;
               end
            XORI:
               begin
                  // control word assignments: ALU_OUT <-- RS($rs) ^ RT($rt)
                  @(negedge clk)
                  {pc_sel, pc_ld, pc_inc, ir_ld}        = 5'b00_0_0_0;
                  {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
                  {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_00_00_0_000;
                  {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
                  {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
                  int_ack = 1'b0;                    FS = xori_;
                  {sp_sel, s_sel}                       = 2'b0_0;
                  #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, n, z};
                  state   = WB_IMM;
               end
            AND:
               begin
                  // control word assignments: ALU_OUT <-- RS($rs) & RT($rt)
                  @(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_00_0_000;
                  {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
```

```
      {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
      int_ack = 1'b0;                    FS = and_;
      {sp_sel, s_sel}                       = 2'b0_0;
      #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, n, z};
      state   = WB_ALU;
   end
PUSH:
   begin
      // control word assignments: DM[--$SP] <-- $RT
      @(negedge clk)
      {pc_sel, pc_ld, pc_inc, ir_ld}        = 5'b00_0_0_0;
      {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
      {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_01_00_0_000;
      {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
      {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
      int_ack = 1'b0;                    FS = 0;
      {sp_sel, s_sel}                       = 2'b1_0;
      #1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};
      state   = PUSH_2;
   end
PUSH_2:
   begin
      // control word assignments: ALU_OUT <-- RS($sp)-4; RT <-- R[$RT]
      @(negedge clk)
      {pc_sel, pc_ld, pc_inc, ir_ld}        = 5'b00_0_0_0;
      {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
      {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_01_00_0_000;
      {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
      {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
      int_ack = 1'b0;                    FS = dec4_;
      {sp_sel, s_sel}                       = 2'b0_0;
      #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, psn, psz};
      state   = PUSH_3;
   end
PUSH_3:
   begin
      // control word assignments: DM[ALU_OUT] <-_ RT($rt)
      //                           ALU_OUT <-- ALU_OUT($SP-4)
      @(negedge clk)
      {pc_sel, pc_ld, pc_inc, ir_ld}        = 5'b00_0_0_0;
      {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
      {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_01_00_0_000;
      {dm_cs, dm_rd, dm_wr}                 = 3'b1_0_1;
      {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
      int_ack = 1'b0;                    FS = 0;
      {sp_sel, s_sel}                       = 2'b0_1;
      #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, psn, psz};
      state   = PUSH_4;
   end
PUSH_4:
   begin
      // control word assignments: R[$SP] <-- ALU_OUT($sp-4)
      @(negedge clk)
      {pc_sel, pc_ld, pc_inc, ir_ld}        = 5'b00_0_0_0;
      {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
      {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b1_11_00_0_000;
      {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
      {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
      int_ack = 1'b0;                   FS = 0;
      {sp_sel, s_sel}                       = 2'b0_0;
      #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, psn, psz};
      state   = FETCH;
   end
POP:
   begin
      // control word assignments:  RT[$rt] <-- DM[$SP++]
      //                            RS <-- R[$SP]
      @(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_00_0_000;
      {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
      {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
      int_ack = 1'b0;                   FS = 0;
      {sp_sel, s_sel}                       = 2'b1_0;
      #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, psn, psz};
      state   = POP_2;
   end
POP_2:
   begin
      // control word assignments:  ALU_OUT <-- RS($SP)
      @(negedge clk)
      {pc_sel, pc_ld, pc_inc, ir_ld}        = 5'b00_0_0_0;
      {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
      {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_00_00_0_000;
      {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
      {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
      int_ack = 1'b0;                   FS = 0;
      {sp_sel, s_sel}                       = 2'b0_0;
      #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, psn, psz};
      state   = POP_3;
   end
POP_3:
   begin
      // control word assignments:  D_IN <-- DM[ALU_OUT]
      //                            ALU_OUT <-- ALU_OUT($SP)
      @(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_00_0_000;
```

```
            {dm_cs, dm_rd, dm_wr}                 = 3'b1_1_0;
            {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
            int_ack = 1'b0;                    FS = 0;
            {sp_sel, s_sel}                       = 2'b0_1;
            #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, psn, psz};
            state   = POP_4;
         end
      POP_4:
         begin
            // control word assignments:  RT[$rt] <- D_IN
            //                            ALU_OUT <-- ALU_OUT($sp) + 4
            @(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_00_0_011;
            {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
            {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
            int_ack = 1'b0;                    FS = inc4_;
            {sp_sel, s_sel}                       = 2'b0_1;
            #1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};
            state   = POP_5;
         end
      POP_5:
         begin
            // control word assignments:  R(sp) <-- ALU_OUT((sp+4)
            @(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_00_0_000;
            {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
            {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
            int_ack = 1'b0;                    FS = 0;
            {sp_sel, s_sel}                       = 2'b0_0;
            #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, psn, psz};
            state   = FETCH;
         end
      NOP:
         begin
            // control word assignments: Do Nothing - ALU_OUT <- ALU_OUT
            @(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_00_0_000;
            {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
            {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
            int_ack = 1'b0;                    FS = 0;
            {sp_sel, s_sel}                       = 2'b0_1;
            #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, psn, psz};
            state   = FETCH;
         end
```

```
ANDI:
   begin
      // control word assignments: ALU_OUT <-- RS($rs) & RT(se_16)
      @(negedge clk)
      {pc_sel, pc_ld, pc_inc, ir_ld}        = 5'b00_0_0_0;
      {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
      {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_00_00_0_000;
      {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
      {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
      int_ack = 1'b0;                    FS = andi_;
      {sp_sel, s_sel}                       = 2'b0_0;
      #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, n, z};
      state   = WB_IMM;
   end
OR:
   begin
      // control word assignments: ALU_OUT <-- RS($rs) | RT($rt)
      @(negedge clk)
      {pc_sel, pc_ld, pc_inc, ir_ld}        = 5'b00_0_0_0;
      {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
      {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_00_00_0_000;
      {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
      {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
      int_ack = 1'b0;                    FS = or_;
      {sp_sel, s_sel}                       = 2'b0_0;
      #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, n, z};
      state   = WB_ALU;
   end
NOR:
   begin
      // control word assignments: ALU_OUT <-- ~(RS($rs) | RT($rt))
      @(negedge clk)
      {pc_sel, pc_ld, pc_inc, ir_ld}        = 5'b00_0_0_0;
      {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
      {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_00_00_0_000;
      {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
      {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
      int_ack = 1'b0;                    FS = nor_;
      {sp_sel, s_sel}                       = 2'b0_0;
      #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, n, z};
      state   = WB_ALU;
   end
ROTL:
   begin
      // control word assignments: ALU_OUT <--
      @(negedge clk)
      {pc_sel, pc_ld, pc_inc, ir_ld}        = 5'b00_0_0_0;
      {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
      {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_00_00_0_000;
      {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
      {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
      int_ack = 1'b0;                    FS = rotl_;
      {sp_sel, s_sel}                       = 2'b0_0;
      #1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, psv, n, z};
      state   = WB_ALU;
   end
ROTR:
   begin
      // control word assignments: ALU_OUT <--
      @(negedge clk)
      {pc_sel, pc_ld, pc_inc, ir_ld}        = 5'b00_0_0_0;
      {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
      {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_00_00_0_000;
      {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
      {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
      int_ack = 1'b0;                    FS = rotr_;
      {sp_sel, s_sel}                       = 2'b0_0;
      #1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, psv, n, z};
      state   = WB_ALU;
   end
MULT:
   begin
      // control word assignments: {Hi,Lo} <-- RS($rs) * RT($rt)
      @(negedge clk)
      {pc_sel, pc_ld, pc_inc, ir_ld}        = 5'b00_0_0_0;
      {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
      {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_00_00_1_000;
      {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
      {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
      int_ack = 1'b0;                    FS = mul_;
      {sp_sel, s_sel}                       = 2'b0_0;
      #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, n, z};
      state   = FETCH;
   end
DIV:
   begin
      // control word assignments: {Hi} <-- RS($rs) / RT($rt)
      // {Lo} <-- RS($rs) % RT($rt)
      @(negedge clk)
      {pc_sel, pc_ld, pc_inc, ir_ld}        = 5'b00_0_0_0;
      {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
      {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_00_00_1_000;
      {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
      {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
      int_ack = 1'b0;                    FS = div_;
      {sp_sel, s_sel}                       = 2'b0_0;
      #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, n, z};
      state   = FETCH;
   end
MFLO:
   begin
      // control word assignments: RD($rd) <-- Lo
      @(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_01_00_0_010;
      {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
      {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
      int_ack = 1'b0;                    FS = 0;
      {sp_sel, s_sel}                       = 2'b0_0;
      #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, psn, psz};
      state   = FETCH;
   end
 MFHI:
   begin
      // control word assignments: RD($rd) <-- Hi
      @(negedge clk)
      {pc_sel, pc_ld, pc_inc, ir_ld}        = 5'b00_0_0_0;
      {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
      {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b1_01_00_0_001;
      {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
      {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
      {sp_sel, s_sel}                       = 2'b0_0;
      int_ack = 1'b0;                    FS = 0;
      #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, psn, psz};
      state   = FETCH;
   end
SETIE:
   begin
      // control word assignments: psi <- 1'b1
      @(negedge clk)
      {pc_sel, pc_ld, pc_inc, ir_ld}        = 5'b00_0_0_0;
      {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
      {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_00_00_0_000;
      {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
      {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
      {sp_sel, s_sel}                       = 2'b0_0;
      int_ack = 1'b0;                    FS = 0;
      #1 {nsi, nsc, nsv, nsn, nsz} = {1'b1, psc, psv, psn, psz};
      state   = FETCH;
   end
BEQ:
   begin
      // control word assignments: ALU_OUT <--
      //RS($rs) - RT($rt)
      @(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_00_0_000;
      {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
```

```
            {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
            {sp_sel, s_sel}                       = 2'b0_0;
            int_ack = 1'b0;                    FS = sub_;
            #1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};
            state   = BEQ2;
         end
      BEQ2:
         begin
            // control word assignments: PC <--
            //PC + signext(IR[15:0])<<2
            @(negedge clk)
            if(psz == 1'b1)
               {pc_sel, pc_ld, pc_inc, ir_ld}     = 5'b10_1_0_0;
            else
               {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_00_0_000;
            {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
            {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
            {sp_sel, s_sel}                       = 2'b0_0;
            int_ack = 1'b0;                    FS = 5'h00;
            #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, psn, psz};
            state   = FETCH;
         end
      BNE:
         begin
         //Dump_Registers;
            // control word assignments: ALU_OUT <--
            //RS($rs) - RT($rt)
            @(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_00_0_000;
            {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
            {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
            {sp_sel, s_sel}                       = 2'b0_0;
            int_ack = 1'b0;                    FS = sub_;
            #1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};
            state   = BNE2;
         end
      BNE2:
         begin
            // control word assignments: PC <--
            //PC + signext(IR[15:0])<<2
            @(negedge clk)
            if(psz == 1'b0)
               {pc_sel, pc_ld, pc_inc, ir_ld}        = 5'b10_1_0_0;
            else
               {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_00_0_000;
            {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
            {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
            {sp_sel, s_sel}                       = 2'b0_0;
            int_ack = 1'b0;                    FS = 5'h00;
            #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, psn, psz};
            state   = FETCH;
         end
      BLEZ:
         begin
            // control word assignments: ALU_OUT <--
            //RS($rs) - RT($rt)
            @(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_00_0_000;
            {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
            {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
            {sp_sel, s_sel}                       = 2'b0_0;
            int_ack = 1'b0;                    FS = sub_;
            #1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};
            state   = BLEZ_2;
         end
      BLEZ_2:
         begin
            // control word assignments: RS($rs) <= 0 ? PC <- PC+s_ext(IR[15:0]
<<2

            @(negedge clk)
            if(psn == 1 || psz == 1)
               {pc_sel, pc_ld, pc_inc, ir_ld}        = 5'b10_1_0_0;
            else
               {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_00_0_000;
            {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
            {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
            {sp_sel, s_sel}                       = 2'b0_0;
            int_ack = 1'b0;                    FS = 0;
            #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, psn, psz};
            state   = FETCH;
         end
      BGTZ:
         begin
            // control word assignments: ALU_OUT <--
            //RS($rs) - RT($rt)
            @(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_00_0_000;
            {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
```

```
                  {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
                  {sp_sel, s_sel}                       = 2'b0_0;
                  int_ack = 1'b0;                    FS = sub_;
                  #1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};
                  state   = BGTZ_2;
               end
            BGTZ_2:
               begin
                  // control word assignments: RS($rs) >= 0 ? PC <- PC+s_ext(IR[15:0]
<<2
                  @(negedge clk)
                  if(psn == 0 || psz == 1)
                     {pc_sel, pc_ld, pc_inc, ir_ld}        = 5'b10_1_0_0;
                  else
                     {pc_sel, pc_ld, pc_inc, ir_ld}        = 5'b00_0_0_0;
                  {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
                  {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_00_00_0_000;
                  {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
                  {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
                  {sp_sel, s_sel}                       = 2'b0_0;
                  int_ack = 1'b0;                    FS = 0;
                  #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, psn, psz};
                  state   = FETCH;
               end
            ORI:
               begin
                  // control word assignments: ALU_OUT <--
                  //RS($rs) | {16'h0, RT[15:0]}
                  @(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_01_0_000;
                  {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
                  {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
                  {sp_sel, s_sel}                       = 2'b0_0;
                  int_ack = 1'b0;                    FS = ori_;
                  #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, psn, z};
                  state   = WB_IMM;
               end
            LUI:
               begin
                  // control word assignments: ALU_OUT <-- {RT[15:0], 16'h0}
                  @(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_01_0_000;
                  {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
                  {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
                  {sp_sel, s_sel}                       = 2'b0_0;
                  int_ack = 1'b0;                    FS = lui_;
```

```
      #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, n, z};
      state   = WB_IMM;
   end
ADDI:
   begin
      // control word assignments: ALU_OUT <-- RS($rs) + RT(se_16)
      @(negedge clk)
      {pc_sel, pc_ld, pc_inc, ir_ld}        = 5'b00_0_0_0;
      {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
      {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_00_00_0_000;
      {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
      {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
      {sp_sel, s_sel}                       = 2'b0_0;
      int_ack = 1'b0;                    FS = add_;
      #1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};
      state   = WB_IMM;
   end
SRL:
   begin
      // control word assignments: ALU_OUT <-- RT($rt) >> shamt
      @(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_00_0_000;
      {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
      {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
      {sp_sel, s_sel}                       = 2'b0_0;
      int_ack = 1'b0;                    FS = srl_;
      #1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, psv, n, z};
      state   = WB_ALU;
   end
 SRA:
   begin
      // control word assignments: ALU_OUT <-- RT($rt) >> shamt
      @(negedge clk)
      {pc_sel, pc_ld, pc_inc, ir_ld}        = 5'b00_0_0_0;
      {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
      {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_00_00_0_000;
      {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
      {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
      {sp_sel, s_sel}                       = 2'b0_0;
      int_ack = 1'b0;                    FS = sra_;
      #1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, psv, n, z};
      state   = WB_ALU;
   end
SLL:
   begin
      // control word assignments: ALU_OUT <-- RT($rt) << shamt
      @(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_00_0_000;
                  {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
                  {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
                  {sp_sel, s_sel}                       = 2'b0_0;
                  int_ack = 1'b0;                    FS = sll_;
                  #1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, psv, n, z};
                  state   = WB_ALU;
               end
            SLT:
               begin
                  // control word assignments: ALU_OUT <-- RS($rs) < RT($rt) ? 1'b1:
1'b0
                  @(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_00_0_000;
                  {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
                  {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
                  {sp_sel, s_sel}                       = 2'b0_0;
                  int_ack = 1'b0;                    FS = slt_;
                  #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, n, z};
                  state   = WB_ALU;
               end
            SLTI:
               begin
                  // control word assignments: ALU_OUT <-- RS($rs) < RT($rt_se_16) ?
                  //                                                      1'b1: 1'b0
                  @(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_00_0_000;
                  {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
                  {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
                  {sp_sel, s_sel}                       = 2'b0_0;
                  int_ack = 1'b0;                    FS = slt_;
                  #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, n, z};
                  state   = WB_IMM;
               end
            SLTIU:
               begin
                  // control word assignments: ALU_OUT <-- RS($rs) < RT($rt_se_16) ?
                  //                                                      1'b1: 1'b0
                  @(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_00_0_000;
                  {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
                  {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
                  {sp_sel, s_sel}                       = 2'b0_0;
                  int_ack = 1'b0;                    FS = sltu_;
                  #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, n, z};
                  state   = WB_IMM;
               end
            SLTU:
               begin
                  // control word assignments: ALU_OUT <-- RS($rs) < RT($rt) ? 1'b1:
1'b0
                  @(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_00_0_000;
                  {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
                  {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
                  {sp_sel, s_sel}                       = 2'b0_0;
                  int_ack = 1'b0;                    FS = sltu_;
                  #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, n, z};
                  state   = WB_ALU;
               end
            SW:
               begin
                  // control word assignments: ALU_OUT <--
                  // RS($rs) + RT(se_16), RT <-- $rt
                  @(negedge clk)
                  {pc_sel, pc_ld, pc_inc, ir_ld}        = 5'b00_0_0_0;
                  {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
                  {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_00_00_0_000;
                  {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
                  {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
                  {sp_sel, s_sel}                       = 2'b0_0;
                  int_ack = 1'b0;                    FS = add_;
                  #1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};
                  state   = WB_MEM;
               end
            LW:
               begin
                  // control word assignments: ALU_OUT <--
                  // RS($rs) + RT(se_16), RT <-- $rt
                  @(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_00_0_000;
                  {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
                  {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
                  {sp_sel, s_sel}                       = 2'b0_0;
                  int_ack = 1'b0;                    FS = add_;
                  #1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};
                  state   = LW_2;
               end
            LW_2:
```

```
            begin
               // control word assignments: D_in <--
               // M[ALU_Out]
               @(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_00_0_000;
               {dm_cs, dm_rd, dm_wr}                 = 3'b1_1_0;
               {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
               {sp_sel, s_sel}                       = 2'b0_0;
               int_ack = 1'b0;                    FS = add_;
               #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, psn, psz};
               state   = WB_DIN;
            end
         J:
            begin
               // control word assignments: PC <- PC + signext(IR[25:0]) << 2
               @(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_00_00_0_100;
               {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
               {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
               {sp_sel, s_sel}                       = 2'b0_0;
               int_ack = 1'b0;                    FS = 5'h0;
               #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, psn, psz};
               state   = FETCH;
            end
         JAL:
            begin
               // control word assignments: PC <- PC + signext(IR[25:0]) << 2
               // R[$31(ra)] <- PC
               @(negedge clk)
               {pc_sel, pc_ld, pc_inc, ir_ld}        = 5'b01_1_0_0;
               {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
               {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b1_10_00_0_100;
               {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
               {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
               {sp_sel, s_sel}                       = 2'b0_0;
               int_ack = 1'b0;                    FS = 5'h0;
               #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, psn, psz};
               state   = FETCH;
            end
         JR:
            begin
               // control word assignments: ALU_OUT <-- RS($rs)
               @(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_00_0_000;
```

```
      {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
      {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
      {sp_sel, s_sel}                       = 2'b0_0;
      int_ack = 1'b0;                    FS = 5'h0;
      #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, psn, psz};
      state   = JR2;
   end
JR2:
   begin
      // control word assignments: PC_Out <-- ALU_Out($rs)
      @(negedge clk)
      {pc_sel, pc_ld, pc_inc, ir_ld}        = 5'b11_1_0_0;
      {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
      {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_00_00_0_000;
      {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
      {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
      {sp_sel, s_sel}                       = 2'b0_0;
      int_ack = 1'b0;                    FS = 5'h0;
      #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, psn, psz};
      state   = FETCH;
   end
INPUT:
   begin
      // control word assignments: ALU_OUT <-- RS($rs) + sign_ext(RT($rt))
      @(negedge clk)
      {pc_sel, pc_ld, pc_inc, ir_ld}        = 5'b00_0_0_0;
      {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
      {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_00_00_0_000;
      {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
      {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
      {sp_sel, s_sel}                       = 2'b0_0;
      int_ack = 1'b0;                    FS = add_;
      #1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};
      state   = INPUT_2;
   end
INPUT_2:
   begin
      // control word assignments: D_IN <-- IOMem[ALU_OUT]
      @(negedge clk)
      {pc_sel, pc_ld, pc_inc, ir_ld}        = 5'b00_0_0_0;
      {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
      {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_00_00_0_000;
      {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
      {io_cs, io_rd, io_wr}                 = 3'b1_1_0;
      {sp_sel, s_sel}                       = 2'b0_0;
      int_ack = 1'b0;                    FS = 0;
      #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, psn, psz};
      state   = WB_DIN;
   end
OUTPUT:
   begin
      // control word assignments: ALU_OUT <-- RS($rs) + sign_ext(RT($rt))
      @(negedge clk)
      {pc_sel, pc_ld, pc_inc, ir_ld}        = 5'b00_0_0_0;
      {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
      {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_00_00_0_000;
      {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
      {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
      {sp_sel, s_sel}                       = 2'b0_0;
      int_ack = 1'b0;                    FS = add_;
      #1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};
      state   = OUTPUT_2;
   end
OUTPUT_2:
   begin
      // control word assignments: IOMem[ALU_OUT] <-- D_IN
      @(negedge clk)
      {pc_sel, pc_ld, pc_inc, ir_ld}        = 5'b00_0_0_0;
      {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
      {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_00_00_0_000;
      {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
      {io_cs, io_rd, io_wr}                 = 3'b1_0_1;
      int_ack = 1'b0;                    FS = 0;
      {sp_sel, s_sel}                       = 2'b0_0;
      #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, psn, psz};
      state   = FETCH;
   end
//SIMD Instruction Execution states
SLL_SIMD:
   begin
      // control word assignments: ALU_OUT <-- RT($rt) << shamt
      @(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_00_0_000;
      {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
      {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
      {sp_sel, s_sel}                       = 2'b0_0;
      int_ack = 1'b0;                    FS = sll_;
      #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, psn, z};
      state   = WB_ALU;
   end
SRL_SIMD:
   begin
      // control word assignments: ALU_OUT <-- RT($rt) >> shamt
      @(negedge clk)
      {pc_sel, pc_ld, pc_inc, ir_ld}        = 5'b00_0_0_0;
      {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
      {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_00_00_0_000;
      {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
      {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
      {sp_sel, s_sel}                       = 2'b0_0;
      int_ack = 1'b0;                    FS = srl_;
      #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, psn, z};
      state   = WB_ALU;
   end
 SRA_SIMD:
   begin
      // control word assignments: ALU_OUT <-- RT($rt) >> shamt
      @(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_00_0_000;
      {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
      {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
      {sp_sel, s_sel}                       = 2'b0_0;
      int_ack = 1'b0;                    FS = sra_;
      #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, psn, z};
      state   = WB_ALU;
   end
ROTL_SIMD:
   begin
      // control word assignments: ALU_OUT <-- RT($rt) << shamt
      @(negedge clk)
      {pc_sel, pc_ld, pc_inc, ir_ld}        = 5'b00_0_0_0;
      {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
      {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_00_00_0_000;
      {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
      {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
      int_ack = 1'b0;                    FS = rotl_;
      {sp_sel, s_sel}                       = 2'b0_0;
      #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, psn, z};
      state   = WB_ALU;
   end
ROTR_SIMD:
   begin
      // control word assignments: ALU_OUT <-- RT($rt) >> shamt
      @(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_00_0_000;
      {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
      {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
      int_ack = 1'b0;                    FS = rotr_;
      {sp_sel, s_sel}                       = 2'b0_0;
      #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, psn, z};
      state   = WB_ALU;
   end
ADD_SIMD:
   begin
```

```
      // control word assignments: ALU_OUT <-- RS($rs) + RT($rt)
      @(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_00_0_000;
      {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
      {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
      int_ack = 1'b0;                    FS = add_;
      {sp_sel, s_sel}                       = 2'b0_0;
      #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, psn, z};
      state   = WB_ALU_SIMD;
   end
 SUB_SIMD:
   begin
      // control word assignments: ALU_OUT <-- RS($rs) - RT($rt)
      @(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_00_0_000;
      {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
      {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
      int_ack = 1'b0;                    FS = sub_;
      {sp_sel, s_sel}                       = 2'b0_0;
      #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, psn, z};
      state   = WB_ALU_SIMD;
   end
XOR_SIMD:
   begin
      // control word assignments: ALU_OUT <-- RS($rs) ^ {16'h0,
      //                                                RT($rt[15:0])}
      @(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_00_0_000;
      {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
      {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
      int_ack = 1'b0;                    FS = xor_;
      {sp_sel, s_sel}                       = 2'b0_0;
      #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, psn, z};
      state   = WB_ALU_SIMD;
   end
AND_SIMD:
   begin
      // control word assignments: ALU_OUT <-- RS($rs) & RT($rt)
      @(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_00_0_000;
      {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
      {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
      int_ack = 1'b0;                    FS = and_;
      {sp_sel, s_sel}                       = 2'b0_0;
      #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, psn, z};
      state   = WB_ALU_SIMD;
   end
OR_SIMD:
   begin
      // control word assignments: ALU_OUT <-- RS($rs) | RT($rt)
      @(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_00_0_000;
      {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
      {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
      int_ack = 1'b0;                    FS = or_;
      {sp_sel, s_sel}                       = 2'b0_0;
      #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, psn, z};
      state   = WB_ALU_SIMD;
   end
NOR_SIMD:
   begin
      // control word assignments: ALU_OUT <-- ~(RS($rs) | RT($rt))
      @(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_00_0_000;
      {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
      {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
      int_ack = 1'b0;                    FS = nor_;
      {sp_sel, s_sel}                       = 2'b0_0;
      #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, psn, z};
      state   = WB_ALU_SIMD;
   end
MULT_SIMD:
   begin
      // control word assignments: {Hi,Lo} <-- RS($rs) * RT($rt)
      @(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_00_1_000;
      {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
      {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
      int_ack = 1'b0;                    FS = mul_;
      {sp_sel, s_sel}                       = 2'b0_0;
      #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, psn, z};
      state   = FETCH;
   end
DIV_SIMD:
   begin
      // control word assignments: {Hi} <-- RS($rs) / RT($rt)
      // {Lo} <-- RS($rs) % RT($rt)
      @(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_00_1_000;
      {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
      {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
      int_ack = 1'b0;                    FS = div_;
      {sp_sel, s_sel}                       = 2'b0_0;
      #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, psn, z};
      state   = FETCH;
   end
//END of SIMD Instructions
WB_ALU_SIMD:
   begin
      // control word assignments: R[rd] <-- ALU_OUT
      @(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_01_00_0_000;
      {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
      {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
      int_ack = 1'b0;                    FS = 5'h0;
      {sp_sel, s_sel}                       = 2'b0_0;
      #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, psn, psz};
      state   = FETCH;
   end
WB_DIN:
   begin
      // control word assignments: RT[$rt] <-- D_IN[M[ALU_Out]]
      @(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_00_0_011;
      {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
      {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
      int_ack = 1'b0;                    FS = 5'h0;
      {sp_sel, s_sel}                       = 2'b0_0;
      #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, psn, psz};
      state   = FETCH;
   end
WB_ALU:
   begin
      // control word assignments: R[rd] <-- ALU_OUT
      @(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_01_00_0_000;
      {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
      {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
```

```
         int_ack = 1'b0;                    FS = 5'h0;
         {sp_sel, s_sel}                       = 2'b0_0;
         #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, psn, psz};
         state   = FETCH;
      end
   WB_IMM:
      begin
         // control word assignments: R[rt] <-- ALU_OUT
         @(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_01_0_000;
         {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
         {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
         int_ack = 1'b0;                    FS = 5'h0;
         {sp_sel, s_sel}                       = 2'b0_0;
         #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, psn, psz};
         state   = FETCH;
      end
   WB_MEM:
      begin
         // control word assignments for
         // M[ALU_OUT($rs + se_16)] <-- RT($rt)
         @(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_00_0_000;
         {dm_cs, dm_rd, dm_wr}                 = 3'b1_0_1;
         {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
         int_ack = 1'b0;                    FS = 5'h0;
         {sp_sel, s_sel}                       = 2'b0_0;
         #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, psn, psz};
         state   = FETCH;
      end
   BREAK:
      begin
         $display("BREAK INSTRUCTION FETCHED %t", $time);
         //control word assignments for "deasserting" 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_00_0_000;
         {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
         int_ack = 1'b0;                    FS = 5'h0;
         {sp_sel, s_sel}                       = 2'b0_0;
         #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, psn, psz};

         $display(" REGISTERS AFTER BREAK");
         $display("  ");
         Dump_Registers; // task to output MIPs RegFile
```

```
            $display("  ");
            $display(" MEMORY AFTER BREAK");
            Dump_Mems;
            $finish;
         end
      ILLEGAL_OP:
         begin
            $display("ILLEGAL OPCODE FETCHED %t", $time);
            //control word assignments for "deasserting" 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_00_0_000;
            {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
            int_ack = 1'b0;                    FS = 5'h0;
            {sp_sel, s_sel}                       = 2'b0_0;
            #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, psn, psz};

            $display(" REGISTERS AFTER ILLEGAL OPCODE");
            $display("  ");
            Dump_Registers;
            $display("  ");
            $display(" PC AND IR AFTER ILLEGAL OPCODE");
            $display("  ");
            Dump_PC_and_IR;
            $finish;
         end
      INTR_1:
         begin
            // PC gets address of interrupt vector; Save PC in $ra
            // control word assignments: RS <-- R[$sp], R[$ra] <-- PC
            @(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_10_00_0_100;
            {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
            {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
            {sp_sel, s_sel}                       = 2'b1_0;
            int_ack = 1'b0;                    FS = 5'h0;
            #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, psn, psz};
            state   = INTR_2;
         end
      INTR_2:
         begin
            // Pass $sp from RS to ALU_OUT;
            // control word assignments: ALU_OUT <-- RS($sp)
            @(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_00_0_000;
```

```
               {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
               {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
               {sp_sel, s_sel}                       = 2'b1_0;
               int_ack = 1'b0;                    FS = 5'h0;
               #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, psn, psz};
               state   = INTR_3;
            end
         INTR_3:
            begin
               // Read address of ISR into D_in;
               // control word assignments: D_in <-- dM[ALU_OUT(0x3FC)
               @(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_00_0_000;
               {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
               {dm_cs, dm_rd, dm_wr}                 = 3'b1_1_0;
               {sp_sel, s_sel}                       = 2'b0_0;
               int_ack = 1'b0;                    FS = 5'h0;
               #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, psn, psz};
               state   = INTR_4;
            end
         INTR_4:
            begin
               // Load PC with M[$sp], pre-decrement $sp. prepare PUSH
               // control word assignments: PC <-- D_in(dM[$sp]),
               //                            ALU_OUT <-- ALU_OUT - 4,
               //                            RT <-- PC
               @(negedge clk)
               {pc_sel, pc_ld, pc_inc, ir_ld}        = 5'b11_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_00_10_0_011;
               {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
               {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
               {sp_sel, s_sel}                       = 2'b0_1;
               int_ack = 1'b0;                    FS = dec4_;
               #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, psn, psz};
               state   = INTR_5;
            end
         INTR_5:
            begin
               // DM[$SP-4] <-- RT(PC)
               // ALU_OUT <-- ALU_OUT($SP-4)-4;
               // RT <-- flags
               @(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_11_0_000;
               {dm_cs, dm_rd, dm_wr}                 = 3'b1_0_1;
               {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
```

```
      {sp_sel, s_sel}                       = 2'b0_1;
      int_ack = 1'b0;                    FS = dec4_;
      #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, psn, psz};
      state   = INTR_6;
   end
INTR_6:
   begin
      // DM[ALU_OUT($SP-8)] <-- RT(flags)
      // R[$SP] <-- ALU_OUT($SP-8)
      // int_ack <-- 1'b1;
      @(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_00_0_000;
      {dm_cs, dm_rd, dm_wr}                 = 3'b1_0_1;
      {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
      {sp_sel, s_sel}                       = 2'b0_0;
      int_ack = 1'b1;                    FS = 0;
      #1 {nsi, nsc, nsv, nsn, nsz} = {1'b0, psc, psv, psn, psz};
      state   = FETCH;
   end

RETI:
   begin
      // ALU_OUT <- RS($SP)
      @(negedge clk)
      {pc_sel, pc_ld, pc_inc, ir_ld}        = 5'b00_0_0_0;
      {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
      {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_00_00_0_000;
      {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
      {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
      int_ack = 1'b0;                    FS = 0;
      #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, psn, psz};
      state   = RETI_2;
   end
RETI_2:
   begin
      // D_IN <- DM[ALU_OUT[$SP]]
      @(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_00_0_000;
      {dm_cs, dm_rd, dm_wr}                 = 3'b1_1_0;
      {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
      int_ack = 1'b0;                    FS = 0;
      #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, psn, psz};
      state   = RETI_3;
   end
RETI_3:
   begin
```

```
      // ALU_OUT <-- ALU_OUT($sp) + 4
      @(negedge clk)
      {pc_sel, pc_ld, pc_inc, ir_ld}        = 5'b00_0_0_0;
      {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
      {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_00_00_0_000;
      {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
      {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
      int_ack = 1'b0;                    FS = inc4_;
      {sp_sel, s_sel}                       = 2'b0_1;
      #1 {nsi, nsc, nsv, nsn, nsz} = {sp_inFlags};
      state   = RETI_4;
   end
RETI_4:
   begin
      // D_IN <-- DM[ALU_OUT($sp+4)]
      // ALU_OUT <-- ALU_OUT($sp+4)+4
      @(negedge clk)
      {pc_sel, pc_ld, pc_inc, ir_ld}        = 5'b00_0_0_0;
      {im_cs, im_rd, im_wr}                 = 3'b0_0_0;
      {D_En, DA_sel, T_sel, HILO_ld, Y_sel} = 9'b0_00_00_0_000;
      {dm_cs, dm_rd, dm_wr}                 = 3'b1_1_0;
      {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
      int_ack = 1'b0;                    FS = inc4_;
      {sp_sel, s_sel}                       = 2'b0_1;
      #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, psn, psz};
      state   = RETI_5;
   end
RETI_5:
   begin
      // PC <-- D_IN(PC)
      // ALU_OUT <-- ALU_OUT($sp+8)
      @(negedge clk)
      {pc_sel, pc_ld, pc_inc, ir_ld}        = 5'b11_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_00_00_0_011;
      {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
      {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
      int_ack = 1'b0;                    FS = 0;
      {sp_sel, s_sel}                       = 2'b0_1;
      #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, psn, psz};
      state   = RETI_6;
   end
RETI_6:
   begin
      // R[$SP] <-- ALU_OUT($sp+8)
      @(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_00_0_000;
      {dm_cs, dm_rd, dm_wr}                 = 3'b0_0_0;
```

```
               {io_cs, io_rd, io_wr}                 = 3'b0_0_0;
               int_ack = 1'b0;                    FS = 0;
               {sp_sel, s_sel}                       = 2'b0_0;
               #1 {nsi, nsc, nsv, nsn, nsz} = {psi, psc, psv, psn, psz};
               state   = FETCH;
            end
      endcase // end of FSM logic

//Implements a reusable code to display contents of the 32x32 array registers.
task Dump_Registers;
   for(i = 0; i<16; i = i + 1) begin
         $display("Time =%t: REG_ADDR[%h] = %h || REG_ADDR[%h] = %h",
         $time, i[4:0],       Top_tb.cpu.IDP.uut1.RegFile[i],
                i[4:0]+5'd16, Top_tb.cpu.IDP.uut1.RegFile[i+16]);
      end
endtask

//Implements a reusable code to display contents of the PC and IR registers.
task Dump_PC_and_IR;
   begin
      $display("PC = %h --- IR = %h",
                Top_tb.cpu.IU.pc.PC_Out, Top_tb.cpu.IU.IR.q);
   end
endtask

//Implements a reusable code to display contents in Memory.
task Dump_Mem;
   for(i = 9'h0C0; i < 9'h100; i = i + 4) begin
      $display("Time =%t: DM[%h] = %h",
               $time, i[11:0], {Top_tb.dMEM.mem[i],
                                 Top_tb.dMEM.mem[i+1],
                                 Top_tb.dMEM.mem[i+2],
                                 Top_tb.dMEM.mem[i+3]});
   end
endtask

//Implements a reusable code to display contents
//in IO and Data Memories.
task Dump_Mems;
begin
   for(i = 9'h0C0; i < 9'h100; i = i + 4) begin
      $display("Time =%t: DM[%h] = %h || IOM[%h] = %h",
               $time, i[11:0], {Top_tb.dMEM.mem[i],
                                 Top_tb.dMEM.mem[i+1],
                                 Top_tb.dMEM.mem[i+2],
                                 Top_tb.dMEM.mem[i+3]},
                      i[11:0], {Top_tb.io.mem[i],
                                 Top_tb.io.mem[i+1],
                                 Top_tb.io.mem[i+2],
                                 Top_tb.io.mem[i+3]},);
      end
      $display("DM[3f0] = %h",    {Top_tb.dMEM.mem[12'h3F0],
                                   Top_tb.dMEM.mem[12'h3F1],
                                   Top_tb.dMEM.mem[12'h3F2],
                                   Top_tb.dMEM.mem[12'h3F3]});

      $display("DM[3f8] = %h",    {Top_tb.dMEM.mem[12'h3F8],
                                   Top_tb.dMEM.mem[12'h3F9],
                                   Top_tb.dMEM.mem[12'h3FA],
                                   Top_tb.dMEM.mem[12'h3FB]});

   end
   endtask


endmodule
```

```
`timescale 1ns / 1ps
/****************************** C E C S  4 4 0 ******************************
 *
 * File Name:  PC.v
 * Project:    Lab_Assignment_5
 * Designer:   Reed Ellison
 * Email:      Reed.Ellison@student.csulb.edu
 * Rev. No.:   Version 1.0
 * Rev. Date:  03/09/2019
 *
 * Purpose: 32'bit Register which holds the value of the Program Counter
 * This register can be incremented by four or can be set to a 32'bit input.
 *
 * Notes:
 *
 ****************************************************************************/
module PC(clk,rst,ld,inc,PC_In,PC_Out);

   //Clock, reset, load signal to load PC_In to Register value
   //Increment PC which is by 4 as memory word is 4 bytes
   input clk,rst,ld,inc;

   //32'bit input value that can be set as PC value
   input [31:0] PC_In;

   //Output of PC Register
   output reg [31:0] PC_Out;

   //Procedural Block
   always @(posedge clk, posedge rst)
      begin
         if(rst) //Output is all zeroes
            PC_Out <= 32'b0;
         //When load is asserted, set input PC value as register value
         else if({ld,inc} == 2'b10)
            PC_Out <= PC_In;
         //When increment is asserted, Increment current PC value by 4
         else if({ld,inc} == 2'b01)
            PC_Out <= PC_Out + 4'h4;
         else
         //Keep PC value as the same
            PC_Out <= PC_Out;
      end

endmodule
```

```
`timescale 1ns / 1ps
/****************************** C E C S  4 4 0 ******************************
 *
 * File Name:  LD_reg32.v
 * Project:    Lab_Assignment_5
 * Designer:   Thomas Nguyen
 * Email:      tholinngu@gmail.com
 * Rev. No.:   Version 1.1
 * Rev. Date:  03/09/2019
 *
 * Purpose: A 32 bit load register that changes output when load or "ld" is
 * asserted, otherwise the output remains the same.
 *
 * Notes:
 ****************************************************************************/
module LD_reg32(clk, reset, ld, d, q);

   input             clk, reset, ld;
   input      [31:0] d;
   output reg [31:0] q;

   //Sequential Block. If load is set, output gets input else stays the same
   always@(posedge clk, posedge reset)
      if(reset)    q <= 32'b0;
      else if(ld)  q <= d;
      else         q <= q;

endmodule
```

```
`timescale 1ns / 1ps
/****************************** C E C S  4 4 0 ******************************
 *
 * File Name:  ALU_32.v
 * Project:    Lab_Assignment_3
 * Designer:   Thomas Nguyen
 * Email:      tholinngu@gmail.com
 * Rev. No.:   Version 1.0
 * Rev. Date:  01/24/2019
 *
 * Purpose: An arithmetic logical unit that does behavioral operations based on
 * a 5 bit function select (FS) opcode. It takes in two 32 bit inputs of (S) data
 * and (T) data and uses those in a specified operation to output a 64 bit value
 * in form of two 32 bit outputs called Y_hi for the upper 32 bit and Y_lo for
 * lower 32 bit. There are status flags of carry (c), overflow (v), negative (n),
 * and zero (z) that are checked.
 *
 * Notes:
 * ALU OPERATION CODES
 * 0 - Pass_S   7 - SLTU   E  - SLL     15 - SP_INIT
 * 1 - Pass_T   8 - AND    F  - INC     16 - ANDI
 * 2 - ADD      9 - OR     10 - INC4    17 - ORI
 * 3 - ADDU     A - XOR    11 - DEC     18 - LUI
 * 4 - SUB      B - NOR    12 - DEC4    19 - XORI
 * 5 - SUBU     C - SRL    13 - ZEROS   1E - MUL
 * 6 - SLT      D - SRA    14 - ONES    1F - DIV
 ****************************************************************************/
module ALU_32(S, T, FS, shamt, y_hi, y_lo, c, v, n, z);

   input  [31:0] S, T;                                          //Inputs
   input   [4:0] FS, shamt;                                     //Function Select
   output [31:0] y_hi, y_lo;                                    //Outputs

   //Status Flag bits carry, overflow, negative and zero
   output        c, v, n, z;
   wire          c_w, v_w, n_w;

   wire   [63:0] prdct;
   wire   [31:0] quote, rem, Y;

   //Barrell Shifter
   wire [31:0] T_shift;
   wire        C_shift;

   //Instantiate MPY_32, DIV_32 and MIPS_32 modules
   MPY_32  uut0(.S(S), .T(T), .prdct(prdct));
   DIV_32  uut1(.S(S), .T(T), .quote(quote), .rem(rem));
   MIPS_32 uut2(.S(S), .T(T), .FS(FS),       .Y(Y),     .c(c_w),   .v(v_w));
   BarrellShifter bs(.D(T), .type(FS), .shamt(shamt), .Out(T_shift), .C(C_shift));
   //Multiplexor to determine which Operation Module is being used
   assign {y_hi, y_lo} = (FS == 5'h1e)         ? {prdct[63:32], //MULT upper 32 bit
                                                  prdct[31:0]}: //MULT lower 32 bit
                         (FS == 5'h1f)         ? {rem, quote} : //DIV
                         (FS == 5'h0C || FS == 5'h0E ||
                          FS == 5'h0D || FS == 5'h1A ||
                          FS == 5'h1B)         ? {31'h0, T_shift}://Shift

                                                 {31'h0, Y}   ; //MIPS ops

   //Set carry flag to x if FS is MULT or DIV else take from MIPS opcodes
   assign c = (FS == 5'h1e || FS == 5'h1f)     ? 1'bX    :
              (FS == 5'h0C || FS == 5'h0E ||
               FS == 5'h0D || FS == 5'h1A ||
               FS == 5'h1B)                    ? C_shift :   c_w;

   //Set overflow flag to x if FS is MULT or DIV else take from MIPS opcodes
   assign v = (FS == 5'h1e || FS == 5'h1f)     ? 1'bX : v_w;

   //Set zero flag. Check DIV op's y_lo for zeroes else check all 64 bits
   assign z = (FS == 5'h1f && y_lo == 32'h0)   ? 1'b1:          //DIV quotient
              (y_hi == 32'h0 && y_lo == 32'h0) ? 1'b1:          //Check 64 bits
                                                 1'b0;

   //Set negative flag using MSB depending on FS. y_hi for mult else y_lo
   assign n = ((FS == 5'h1e))                  ? y_hi[31]:      //MULT
              ((FS == 5'h1f && y_lo[31]))      ? y_lo[31]:      //DIV quotient
              ((FS == 5'h03 || FS == 5'h05))   ? 1'b0    :      //ADDU & SUBU
                                                 y_lo[31];      //Other opcodes

endmodule
```

```
`timescale 1ns / 1ps
/****************************** C E C S  4 4 0 ******************************
 *
 * File Name:  V_ALU.v
 * Project:    SIMD MIPS32
 * Designer:   Thomas Nguyen
 * Email:      tholinngu@gmail.com
 * Rev. No.:   Version 1.0
 * Rev. Date:  04/18/2019
 *
 * Purpose: An SIMD arithmetic logical unit that does behavioral operations based
 * on a 5 bit function select (FS) opcode. It takes in two 32 bit inputs of (S)
 * data and (T) data and uses those in a specified operation to output different
 * types of data. It can output 16-bits or 8-bits. This runs in parallel with
 * the standard ALU_32. This runs in parallel with the normal ALU. Status flags
 * are not used except for zero flag.
 *
 * Notes:
 *
 ***************************************************************************/
module V_ALU(simd_sel, S, T, FS, shamt, z, y_hi, y_lo);
   input  [31:0] S, T;        //Inputs
   input   [1:0] simd_sel;    //SIMD mode sel
   input   [4:0] FS, shamt;   //Function Select
   output [31:0] y_hi, y_lo;  //Outputs

   //Status Flag bits carry, overflow, negative and zero
   output        z;

   wire   [63:0] prdct;
   wire   [31:0] quote, rem, Y;
   wire   [31:0] T_shift;

   //Instantiate SIMD versions of the MPY_32, DIV_32 and MIPS_32 modules for
   //Vector ALU
   MPY_SIMD  uut0(.simd_sel(simd_sel), .S(S), .T(T), .prdct(prdct));
   DIV_SIMD  uut1(.simd_sel(simd_sel), .S(S), .T(T), .quote(quote), .rem(rem));
   MIPS_SIMD uut2(.SIMD_Sel(simd_sel), .S(S), .T(T), .FS(FS),       .Y(Y));

   BarrelShift_SIMD BS(.simd_sel(simd_sel), .D(T), .type(FS),
                       .shamt(shamt),       .Out(T_shift));

   assign {y_hi, y_lo} = (FS == 5'h1e)         ? {prdct[63:32], //MULT upper 32 bit
                                                  prdct[31:0]}: //MULT lower 32 bit
                         (FS == 5'h1f)         ? {rem, quote} : //DIV
                         (FS == 5'h0C || FS == 5'h0E ||
                          FS == 5'h0D || FS == 5'h1A ||
                          FS == 5'h1B)         ? {31'h0, T_shift}://Shift
                                                 {31'h0, Y}   ; //MIPS ops
   //Set zero flag. Check DIV op's y_lo for zeroes else check all 64 bits
   assign z = (y_hi == 32'h0 && y_lo == 32'h0) ? 1'b1: //Check 64 bits
                                                 1'b0;

endmodule
```

```verilog
`timescale 1ns / 1ps
/****************************** C E C S  4 4 0 ******************************
 *
 * File Name:  regfile32.v
 * Project:    Lab_Assignment_4
 * Designer:   Thomas Nguyen
 * Email:      tholinngu@gmail.com
 * Rev. No.:   Version 1.0
 * Rev. Date:  02/07/2019
 *
 * Purpose: This is a 32 x 32 dual port register. It contains general registers
 * and address registers used to support addressing modes. These registers are
 * for the MIPS processor use. This uses a write enable called "D_En" that allows
 * for the registers to be written to and sets the register to the value on the
 * 32-bit D data input. It contains outputs S and T that take their -bit hex
 * values from specific registers depending on their correlating S_Addr and T_Addr
 * values. They are asynchronous outputs from the register.
 * $r0 is always to have the value 0 and cannot be written to because it is
 * a "Read Only Register".
 *
 * Notes: Registers are referred to as $rX where X is a number between 0 - 31 in
 * this document.
 ****************************************************************************/
module regfile32(clk, reset, D_En, D_Addr, S_Addr, T_Addr, D, S, T);

   input         clk, reset, D_En;       //clk, reset and write enable
   input   [4:0] D_Addr, S_Addr, T_Addr; //read address and write addresses
   input  [31:0] D;                      //value to write to
   output [31:0] S, T;                   //outputs to ALU_32

   //32x32 Register from $r0 to $r31
   reg    [31:0] RegFile [31:0];

   //Assign Outputs to equal register at specific addresses
   assign S = RegFile[S_Addr];
   assign T = RegFile[T_Addr];

   always@(posedge clk, posedge reset)
      if(reset)
         RegFile[0] <= 32'h0;          //$r0 should always be 0
      else if(D_En && D_Addr != 5'h0) //Checks to make write enable is on and not $r0
         RegFile[D_Addr] <= D;        //Writes to register
      else
         RegFile[D_Addr] <= RegFile[D_Addr];

endmodule
```

```
`timescale 1ns / 1ps
/****************************** C E C S  4 4 0 ******************************
 *
 * File Name:  REG32.v
 * Project:    Lab_Assignment_4
 * Designer:   Thomas Nguyen
 * Email:      tholinngu@gmail.com
 * Rev. No.:   Version 1.0
 * Rev. Date:  03/01/2019
 *
 * Purpose: A 32 bit register that passes input of D to Q on every active
 * edge of the clock. Used as a pipeline register.
 *
 * Notes:
 ****************************************************************************/
module REG32(clk, reset, D, Q);

   input             clk, reset;
   input      [31:0] D;
   output reg [31:0] Q;

   //Sequential Block. Output gets input else gets 0 on reset
   always@(posedge clk, posedge reset)
      if(reset)
         Q <= 32'b0;
      else
         Q <= D;

endmodule
```

```verilog
`timescale 1ns / 1ps
/****************************** C E C S  4 4 0 ******************************
 *
 * File Name:  MPY_32.v
 * Project:    Lab_Assignment_1
 * Designer:   Thomas Nguyen
 * Email:      tholinngu@gmail.com
 * Rev. No.:   Version 1.0
 * Rev. Date:  01/24/2019
 *
 * Purpose: The multiplication module is part of the function select opcode
 * that uses two 32 bit inputs of S and T data and converts those registers
 * to integer and does the calculation. The output is then 64 bits and then
 * goes back to the ALU_32 and split into half of Y_hi and Y_lo.
 * This uses signed integer.
 *
 * Notes:
 *
 ****************************************************************************/
module MPY_32(S, T, prdct);

   input      [31:0] S, T;          //Inputs
   output reg [63:0] prdct;         //Output Product

   integer           int_S, int_T;

   always@(S, T) begin
      int_S = S;                    //Convert to integer
      int_T = T;

      prdct = int_S * int_T;        //Use integers to fill register

   end

endmodule
```

```
`timescale 1ns / 1ps
/****************************** C E C S  4 4 0 ******************************
 *
 * File Name:  DIV_32.v
 * Project:    Lab_Assignment_1
 * Designer:   Thomas Nguyen
 * Email:      tholinngu@gmail.com
 * Rev. No.:   Version 1.0
 * Rev. Date:  01/24/2019
 *
 * Purpose: The division module is part of the function select opcode that
 * uses two 32 bit inputs S and T and two 32 bit output. S and T are converted
 * to integer and then the quotient and remainder are calculated using S in
 * the numerator and T in the denominator. Modulus is used to find the remainder.
 * This uses signed integer.
 *
 * Notes:
 *
 ****************************************************************************/
module DIV_32(S, T, quote, rem);

   input      [31:0] S, T;          //Inputs
   output reg [31:0] quote, rem;    //Outputs Quotient and Remainder

   integer           int_S, int_T;

   always@(S, T) begin
      int_S = S;                    //Convert to integer
      int_T = T;

      quote = int_S / int_T;        //Lower 32 bit output

      rem   = int_S % int_T;        //Upper 32 bit output

   end

endmodule
```

```
`timescale 1ns / 1ps
/****************************** C E C S  4 4 0 ******************************
 *
 * File Name:  MIPS_32.v
 * Project:    Lab_Assignment_1
 * Designer:   Thomas Nguyen
 * Email:      tholinngu@gmail.com
 * Rev. No.:   Version 1.0
 * Rev. Date:  01/29/2019
 *
 * Purpose: The Microprocessor without Interlocked Pipelined Stages is a
 * 32 bit RISC ISA that uses a 5 bit function select opcode to do ALU
 * (Arithmetic Logical Unit) instructions. This uses two 32 bit input S and T
 * and does certain operations on it depending on the opcode. Flags carry and
 * overflow determined by the FS opcode. This module contains Arithmetic,
 * Logical and Other operations.
 *
 * Notes: Logical operations do not care for carry or overflow.
 * All arithmetic operations use signed integers except ADDU and SUBU.
 * Signed add/sub operation overflow is determined by bit sign of the
 * two addend and the sum or the minuend, subtrahend and difference. If
 * result bit is different than the bit of first number than it overflows.
 * Unsigned add/sub operation's carry is also their overflow.
 *
 ***************************************************************************/
module MIPS_32(S, T, FS, Y, c, v);

   input      [31:0] S, T;                                           //Inputs
   input       [4:0] FS;                                             //FS opcode
   output reg [31:0] Y;                                              //Output

   //Integers to store Inputs
   integer           int_S, int_T;

   //Status Flag bits carry and overflow
   output reg        c, v;

   //Arithmetic Operations
   parameter PASS_S = 5'h00, PASS_T = 5'h01, ADD = 5'h02,     ADDU = 5'h03,
             SUB = 5'h04,    SUBU = 5'h05,   SLT = 5'h06,     SLTU = 5'h07,

   //Logical Operations
             AND = 5'h08,    OR = 5'h09,     XOR = 5'h0a,     NOR = 5'h0b,
             SRL = 5'h0c,    SRA = 5'h0d,    SLL = 5'h0e,     ANDI = 5'h16,
             ORI = 5'h17,    LUI = 5'h18,    XORI = 5'h19,

   //Other Operations
             INC = 5'h0f,    INC4 = 5'h10,   DEC = 5'h11,     DEC4= 5'h12,
             ZEROS = 5'h13,  ONES = 5'h14,   SP_INIT = 5'h15;
```

```
always@(S or T or FS) begin
   int_S = S;
   int_T = T;

   case(FS)
      PASS_S : {c, v, Y} = {1'bX, 1'bX, S};                       //Pass S

      PASS_T : {c, v, Y} = {1'bX, 1'bX, T};                       //Pass T

      ADD    : begin                                              //Add Signed
                  {c, Y} = S + T;
                  if((int_S < 0 && int_T < 0 && !Y[31]) ||        //Overflow logic
                     (int_S > 0 && int_T > 0 &&  Y[31]))          //addition
                     v = 1'b1;
                  else
                     v = 1'b0;
               end

      ADDU   : begin                                              //Add Unsigned
                  {c, v, Y} = S + T;                              //Carry is
                  c = v;                                          //overflow in
               end                                                //unsigned.

      SUB    : begin                                              //Subtract
                  {c, Y} = S - T;                                 //Signed
                  if((int_S < 0 && int_T > 0 && !Y[31]) ||        //Overflow logic
                     (int_S > 0 && int_T < 0 &&  Y[31]))          //subtraction.
                     v = 1'b1;
                  else
                     v = 1'b0;
               end

      SUBU   : begin                                              //Subtract
                  {c, v, Y} = S - T;                              //Unsigned
                  c = v;
               end

      SLT    : {c, v, Y} = (int_S < int_T) ? {1'bX, 1'bX, 32'b1}: //Set on less
                                             {1'bX, 1'bX, 32'b0}; //than signed

      SLTU   : {c, v, Y} = (S < T) ? {1'bX, 1'bX, 32'b1}:         //Set on less
                                     {1'bX, 1'bX, 32'b0};         //than unsigned

      AND    : {c, v, Y} = {1'bX, 1'bX, S & T};                   //Logic AND

      OR     : {c, v, Y} = {1'bX, 1'bX, S | T};                   //Logic OR

      XOR    : {c, v, Y} = {1'bX, 1'bX, S ^ T};                   //Logic XOR

      NOR    : {c, v, Y} = {1'bX, 1'bX, ~(S | T)};                //Logic NOR
```

```
SRL    : {c, v, Y} = {T[0], 1'bX, 1'b0, T[31:1]};           //Shift Right
                                                             //Logical

SRA    : {c, v, Y} = {T[0], 1'bX, T[31], T[31:1]};          //Shift Right
                                                             //Arithmetic

SLL    : {c, v, Y} = {T[31], 1'bX, T[30:0], 1'b0};          //Shift Left
                                                             //Logical

ANDI   : {c, v, Y} = {1'bX, 1'bX, S & {16'h0, T[15:0]}};    //AND Immediate

ORI    : {c, v, Y} = {1'bX, 1'bX, S | {16'h0, T[15:0]}};    //OR Immediate

LUI    : {c, v, Y} = {1'bX, 1'bX, {T[15:0], 16'h0}};        //Load upper
                                                             //Immediate

XORI   : {c, v, Y} = {1'bX, 1'bX, S ^ {16'h0, T[15:0]}};    //XOR Immediate

INC    : begin                                               //Increment by 1
            {c, Y} = S + 1;
            if((int_S < 0 && int_T < 0 && !Y[31]) ||        //Overflow logic
               (int_S > 0 && int_T > 0 &&  Y[31]))          //addition.
               v = 1'b1;
            else
               v = 1'b0;
         end

INC4   : begin                                               //Increment by 4
            {c, Y} = S + 4;
            if((int_S < 0 && int_T < 0 && !Y[31]) ||
               (int_S > 0 && int_T > 0 &&  Y[31]))
               v = 1'b1;
            else
               v = 1'b0;
         end

DEC    : begin                                               //Decrement by 1
            {c, Y} = S - 1;
            if((int_S < 0 && int_T > 0 && !Y[31]) ||        //Overflow logic
               (int_S > 0 && int_T < 0 &&  Y[31]))          //subtraction.
               v = 1'b1;
            else
               v = 1'b0;
         end

DEC4   : begin                                               //Decrement by 4
            {c, Y} = S - 4;
            if((int_S < 0 && int_T > 0 && !Y[31]) ||
               (int_S > 0 && int_T < 0 &&  Y[31]))
```

```
                        v = 1'b1;
                     else
                        v = 1'b0;
                  end

         ZEROS  : {c, v, Y} = {1'bX, 1'bX, 32'h0};                   //Fill all zeros

         ONES   : {c, v, Y} = {1'bX, 1'bX, {32{1'hf}}};              //Fill all ones

         SP_INIT: {c, v, Y} = {1'bX, 1'bX, 32'h3fc};                 //Set 0x3fc

         default: {c, v, Y} = 1'bX;

      endcase
      end //begin end

endmodule
```

```
`timescale 1ns / 1ps
/****************************** C E C S  4 4 0 ******************************
 *
 * File Name:  BarrellShifter.v
 * Project:    Final_Project
 * Designer:   Thomas Nguyen and Reed Ellison
 * Email:      Tholinngu@gmail.com and notwreed@gmail.com
 * Rev. No.:   Version 1.0
 * Rev. Date:  04/10/2019
 *
 * Purpose: Barrell Shifter Module Do Implement Dhe Shift Instructions
 * for the ALU
 *
 * NoDes:
 ****************************************************************************/
module BarrellShifter(D, type, shamt, Out, C);

   input       [4:0]   type;           // Function Select from MCU
   input       [4:0]   shamt;          // Shifting amount from IR[10:6]
   input      [31:0]   D;              // Data input for shifiDing
   output reg          C;              // Carry flag
   output reg [31:0]   Out;            // Shifted Output

   always@(*)
      case(type)
         5'h0E: // Shift Left Logical
            case(shamt)
               5'd0:  {C,Out} = {1'b0, D};
               5'd1:  {C,Out} = {D[31], D[30:0],  1'b0};
               5'd2:  {C,Out} = {D[30], D[29:0],  2'b0};
               5'd3:  {C,Out} = {D[29], D[28:0],  3'b0};
               5'd4:  {C,Out} = {D[28], D[27:0],  4'b0};
               5'd5:  {C,Out} = {D[27], D[26:0],  5'b0};
               5'd6:  {C,Out} = {D[26], D[25:0],  6'b0};
               5'd7:  {C,Out} = {D[25], D[24:0],  7'b0};
               5'd8:  {C,Out} = {D[24], D[23:0],  8'b0};
               5'd9:  {C,Out} = {D[23], D[22:0],  9'b0};
               5'd10: {C,Out} = {D[22], D[21:0], 10'b0};
               5'd11: {C,Out} = {D[21], D[20:0], 11'b0};
               5'd12: {C,Out} = {D[20], D[19:0], 12'b0};
               5'd13: {C,Out} = {D[19], D[18:0], 13'b0};
               5'd14: {C,Out} = {D[18], D[17:0], 14'b0};
               5'd15: {C,Out} = {D[17], D[16:0], 15'b0};
               5'd16: {C,Out} = {D[16], D[15:0], 16'b0};
               5'd17: {C,Out} = {D[15], D[14:0], 17'b0};
               5'd18: {C,Out} = {D[14], D[13:0], 18'b0};
               5'd19: {C,Out} = {D[13], D[12:0], 19'b0};
               5'd20: {C,Out} = {D[12], D[11:0], 20'b0};
               5'd21: {C,Out} = {D[11], D[10:0], 21'b0};
               5'd22: {C,Out} = {D[10], D[ 9:0], 22'b0};
               5'd23: {C,Out} = {D[ 9], D[ 8:0], 23'b0};
               5'd24: {C,Out} = {D[ 8], D[ 7:0], 24'b0};
               5'd25: {C,Out} = {D[ 7], D[ 6:0], 25'b0};
               5'd26: {C,Out} = {D[ 6], D[ 5:0], 26'b0};
               5'd27: {C,Out} = {D[ 5], D[ 4:0], 27'b0};
               5'd28: {C,Out} = {D[ 4], D[ 3:0], 28'b0};
               5'd29: {C,Out} = {D[ 3], D[ 2:0], 29'b0};
               5'd30: {C,Out} = {D[ 2], D[ 1:0], 30'b0};
               5'd31: {C,Out} = {D[ 1], D[0],    31'b0};
            endcase
         5'h0C: // Shift Right Logical
            case(shamt)
               5'd0:  {C,Out} = {1'b0, D};
               5'd1:  {C,Out} = {D[ 0],  1'b0, D[31: 1]};
               5'd2:  {C,Out} = {D[ 1],  2'b0, D[31: 2]};
               5'd3:  {C,Out} = {D[ 2],  3'b0, D[31: 3]};
               5'd4:  {C,Out} = {D[ 3],  4'b0, D[31: 4]};
               5'd5:  {C,Out} = {D[ 4],  5'b0, D[31: 5]};
               5'd6:  {C,Out} = {D[ 5],  6'b0, D[31: 6]};
               5'd7:  {C,Out} = {D[ 6],  7'b0, D[31: 7]};
               5'd8:  {C,Out} = {D[ 7],  8'b0, D[31: 8]};
               5'd9:  {C,Out} = {D[ 8],  9'b0, D[31: 9]};
               5'd10: {C,Out} = {D[ 9], 10'b0, D[31:10]};
               5'd11: {C,Out} = {D[10], 11'b0, D[31:11]};
               5'd12: {C,Out} = {D[11], 12'b0, D[31:12]};
               5'd13: {C,Out} = {D[12], 13'b0, D[31:13]};
               5'd14: {C,Out} = {D[13], 14'b0, D[31:14]};
               5'd15: {C,Out} = {D[14], 15'b0, D[31:15]};
               5'd16: {C,Out} = {D[15], 16'b0, D[31:16]};
               5'd17: {C,Out} = {D[16], 17'b0, D[31:17]};
               5'd18: {C,Out} = {D[17], 18'b0, D[31:18]};
               5'd19: {C,Out} = {D[18], 19'b0, D[31:19]};
               5'd20: {C,Out} = {D[19], 20'b0, D[31:20]};
               5'd21: {C,Out} = {D[20], 21'b0, D[31:21]};
               5'd22: {C,Out} = {D[21], 22'b0, D[31:22]};
               5'd23: {C,Out} = {D[22], 23'b0, D[31:23]};
               5'd24: {C,Out} = {D[23], 24'b0, D[31:24]};
               5'd25: {C,Out} = {D[24], 25'b0, D[31:25]};
               5'd26: {C,Out} = {D[25], 26'b0, D[31:26]};
               5'd27: {C,Out} = {D[26], 27'b0, D[31:27]};
               5'd28: {C,Out} = {D[27], 28'b0, D[31:28]};
               5'd29: {C,Out} = {D[28], 29'b0, D[31:29]};
               5'd30: {C,Out} = {D[29], 30'b0, D[31:30]};
               5'd31: {C,Out} = {D[30], 31'b0, D[31]   };
            endcase
         5'h0D: // Shift Right Arithmetic
            case(shamt)
               5'd0:  {C,Out} = {1'b0, D};
               5'd1:  {C,Out} = {D[ 0], D[31], { 1{D[31]}}, D[30: 1]};
               5'd2:  {C,Out} = {D[ 1], D[31], { 2{D[31]}}, D[30: 2]};
               5'd3:  {C,Out} = {D[ 2], D[31], { 3{D[31]}}, D[30: 3]};
               5'd4:  {C,Out} = {D[ 3], D[31], { 4{D[31]}}, D[30: 4]};
               5'd5:  {C,Out} = {D[ 4], D[31], { 5{D[31]}}, D[30: 5]};
               5'd6:  {C,Out} = {D[ 5], D[31], { 6{D[31]}}, D[30: 6]};
               5'd7:  {C,Out} = {D[ 6], D[31], { 7{D[31]}}, D[30: 7]};
               5'd8:  {C,Out} = {D[ 7], D[31], { 8{D[31]}}, D[30: 8]};
               5'd9:  {C,Out} = {D[ 8], D[31], { 9{D[31]}}, D[30: 9]};
               5'd10: {C,Out} = {D[ 9], D[31], {10{D[31]}}, D[30:10]};
               5'd11: {C,Out} = {D[10], D[31], {11{D[31]}}, D[30:11]};
               5'd12: {C,Out} = {D[11], D[31], {12{D[31]}}, D[30:12]};
               5'd13: {C,Out} = {D[12], D[31], {13{D[31]}}, D[30:13]};
               5'd14: {C,Out} = {D[13], D[31], {14{D[31]}}, D[30:14]};
               5'd15: {C,Out} = {D[14], D[31], {15{D[31]}}, D[30:15]};
               5'd16: {C,Out} = {D[15], D[31], {16{D[31]}}, D[30:16]};
               5'd17: {C,Out} = {D[16], D[31], {17{D[31]}}, D[30:17]};
               5'd18: {C,Out} = {D[17], D[31], {18{D[31]}}, D[30:18]};
               5'd19: {C,Out} = {D[18], D[31], {19{D[31]}}, D[30:19]};
               5'd20: {C,Out} = {D[19], D[31], {20{D[31]}}, D[30:20]};
               5'd21: {C,Out} = {D[20], D[31], {21{D[31]}}, D[30:21]};
               5'd22: {C,Out} = {D[21], D[31], {22{D[31]}}, D[30:22]};
               5'd23: {C,Out} = {D[22], D[31], {23{D[31]}}, D[30:23]};
               5'd24: {C,Out} = {D[23], D[31], {24{D[31]}}, D[30:24]};
               5'd25: {C,Out} = {D[24], D[31], {25{D[31]}}, D[30:25]};
               5'd26: {C,Out} = {D[25], D[31], {26{D[31]}}, D[30:26]};
               5'd27: {C,Out} = {D[26], D[31], {27{D[31]}}, D[30:27]};
               5'd28: {C,Out} = {D[27], D[31], {28{D[31]}}, D[30:28]};
               5'd29: {C,Out} = {D[28], D[31], {29{D[31]}}, D[30:29]};
               5'd30: {C,Out} = {D[29], D[31], {30{D[31]}}, D[30]   };
               5'd31: {C,Out} = {D[30], D[31], {31{D[31]}}          };
            endcase
         5'h1A: // Rotate Left
            case(shamt)
               5'd0:  {C,Out} = D;
               5'd1:  {C,Out} = {D[30:0], D[31]};
               5'd2:  {C,Out} = {D[29:0], D[31:30]};
               5'd3:  {C,Out} = {D[28:0], D[31:29]};
               5'd4:  {C,Out} = {D[27:0], D[31:28]};
               5'd5:  {C,Out} = {D[26:0], D[31:27]};
               5'd6:  {C,Out} = {D[25:0], D[31:26]};
               5'd7:  {C,Out} = {D[24:0], D[31:25]};
               5'd8:  {C,Out} = {D[23:0], D[31:24]};
               5'd9:  {C,Out} = {D[22:0], D[31:23]};
               5'd10: {C,Out} = {D[21:0], D[31:22]};
               5'd11: {C,Out} = {D[20:0], D[31:21]};
               5'd12: {C,Out} = {D[19:0], D[31:20]};
               5'd13: {C,Out} = {D[18:0], D[31:19]};
               5'd14: {C,Out} = {D[17:0], D[31:18]};
               5'd15: {C,Out} = {D[16:0], D[31:17]};
               5'd16: {C,Out} = {D[15:0], D[31:16]};
               5'd17: {C,Out} = {D[14:0], D[31:15]};
```

```
            5'd18: {C,Out} = {D[13:0], D[31:14]};
            5'd19: {C,Out} = {D[12:0], D[31:13]};
            5'd20: {C,Out} = {D[11:0], D[31:12]};
            5'd21: {C,Out} = {D[10:0], D[31:11]};
            5'd22: {C,Out} = {D[9:0],  D[31:10]};
            5'd23: {C,Out} = {D[8:0],  D[31:9]};
            5'd24: {C,Out} = {D[7:0],  D[31:8]};
            5'd25: {C,Out} = {D[6:0],  D[31:7]};
            5'd26: {C,Out} = {D[5:0],  D[31:6]};
            5'd27: {C,Out} = {D[4:0],  D[31:5]};
            5'd28: {C,Out} = {D[3:0],  D[31:4]};
            5'd29: {C,Out} = {D[2:0],  D[31:3]};
            5'd30: {C,Out} = {D[1:0],  D[31:2]};
            5'd31: {C,Out} = {D[0],    D[31:1]};
         endcase
      5'h1B: // Rotate Right
         case(shamt)
            5'd0:  {C,Out} = D;
            5'd1:  {C,Out} = {D[0],    D[31:1]};
            5'd2:  {C,Out} = {D[1:0],  D[31:2]};
            5'd3:  {C,Out} = {D[2:0],  D[31:3]};
            5'd4:  {C,Out} = {D[3:0],  D[31:4]};
            5'd5:  {C,Out} = {D[4:0],  D[31:5]};
            5'd6:  {C,Out} = {D[5:0],  D[31:6]};
            5'd7:  {C,Out} = {D[6:0],  D[31:7]};
            5'd8:  {C,Out} = {D[7:0],  D[31:8]};
            5'd9:  {C,Out} = {D[8:0],  D[31:9]};
            5'd10: {C,Out} = {D[9:0],  D[31:10]};
            5'd11: {C,Out} = {D[10:0], D[31:11]};
            5'd12: {C,Out} = {D[11:0], D[31:12]};
            5'd13: {C,Out} = {D[12:0], D[31:13]};
            5'd14: {C,Out} = {D[13:0], D[31:14]};
            5'd15: {C,Out} = {D[14:0], D[31:15]};
            5'd16: {C,Out} = {D[15:0], D[31:16]};
            5'd17: {C,Out} = {D[16:0], D[31:17]};
            5'd18: {C,Out} = {D[17:0], D[31:18]};
            5'd19: {C,Out} = {D[18:0], D[31:19]};
            5'd20: {C,Out} = {D[19:0], D[31:20]};
            5'd21: {C,Out} = {D[20:0], D[31:21]};
            5'd22: {C,Out} = {D[21:0], D[31:22]};
            5'd23: {C,Out} = {D[22:0], D[31:23]};
            5'd24: {C,Out} = {D[23:0], D[31:24]};
            5'd25: {C,Out} = {D[24:0], D[31:25]};
            5'd26: {C,Out} = {D[25:0], D[31:26]};
            5'd27: {C,Out} = {D[26:0], D[31:27]};
            5'd28: {C,Out} = {D[27:0], D[31:28]};
            5'd29: {C,Out} = {D[28:0], D[31:29]};
            5'd30: {C,Out} = {D[29:0], D[31:30]};
            5'd31: {C,Out} = {D[30:0], D[31]};
         endcase
      endcase

endmodule
```

```
`timescale 1ns / 1ps
/****************************** C E C S  4 4 0 ******************************
 *
 * File Name:  MPY_SIMD.v
 * Project:    SIMD MIPS
 * Designer:   Thomas Nguyen
 * Email:      tholinngu@gmail.com
 * Rev. No.:   Version 1.0
 * Rev. Date:  01/24/2019
 *
 * Purpose: The multiplication module is part of the function select opcode
 * that uses two 32 bit inputs of S and T data and converts those registers
 * to integer and does the calculation. The SIMD version allows to select
 * 4 16 or 2 32 bits of output to a 64 bit register.
 *
 * Notes:
 *
 *****************************************************************************/
module MPY_SIMD(simd_sel, S, T, prdct);
   input       [1:0] simd_sel;      //SIMD mode selector
   input      [31:0] S, T;          //Inputs
   output reg [63:0] prdct;         //Output Product

   integer           int_S, int_T;
   parameter simd8 = 2'b01, simd16 = 2'b10;
   always@(S, T) begin
      int_S = S;                    //Convert to integer
      int_T = T;
      case(simd_sel)
         simd8: begin
            prdct[15:0] = int_S[7:0] * int_T[7:0];
            prdct[31:16] = int_S[15:8] * int_T[15:8];
            prdct[47:32] = int_S[23:16] * int_T[23:16];
            prdct[63:48] = int_S[31:24] * int_T[31:24];
         end
         simd16: begin
            prdct[31:0] = int_S[15:0] * int_T[15:0];
            prdct[63:32] = int_S[31:16] * int_T[31:16];
         end
         default:
            prdct = int_S * int_T;
      endcase
   end

endmodule
```

```
`timescale 1ns / 1ps
/****************************** C E C S  4 4 0 ******************************
 *
 * File Name:  DIV_SIMD.v
 * Project:    SIMD MIPS
 * Designer:   Thomas Nguyen
 * Email:      tholinngu@gmail.com
 * Rev. No.:   Version 1.0
 * Rev. Date:  04/18/2019
 *
 * Purpose: The division module is part of the function select opcode that
 * uses two 32 bit inputs S and T and two 32 bit output. The SIMD version
 * allows the output to be a vector of multiple data. The SIMD select let's
 * the user choose between 8, 16 or the normal 32 bits. Quotient is loaded
 * into one register and remainder another.
 *
 * Notes:
 *
 ****************************************************************************/
module DIV_SIMD(simd_sel, S, T, quote, rem);
   input       [1:0] simd_sel;      //SIMD mode select
   input      [31:0] S, T;          //Inputs
   output reg [31:0] quote, rem;    //Outputs Quotient and Remainder

   integer           int_S, int_T;

   parameter simd8 = 2'b01, simd16 = 2'b10;

   always@(S, T, simd_sel) begin
      int_S = S;                    //Convert to integer
      int_T = T;
      case(simd_sel)
         simd8: begin
            quote[7:0] = int_S[7:0] / int_T[7:0];
            quote[15:8] = int_S[15:8] / int_T[15:8];
            quote[23:16] = int_S[23:16] / int_T[23:16];
            quote[31:24] = int_S[31:24] / int_T[31:24];
            rem[7:0] = int_S[7:0] % int_T[7:0];
            rem[15:8] = int_S[15:8] % int_T[15:8];
            rem[23:16] = int_S[23:16] % int_T[23:16];
            rem[31:24] = int_S[31:24] % int_T[31:24];
         end
         simd16: begin
            quote[15:0] = int_S[15:0] / int_T[15:0];
            quote[31:16] = int_S[31:16] / int_T[31:16];
            rem[15:0] = int_S[15:0] % int_T[15:0];
            rem[31:16] = int_S[31:16] % int_T[31:16];
         end
         default: begin
            quote = int_S / int_T;
            rem = int_S % int_T;
         end
   endcase
   end

endmodule
```

```
`timescale 1ns / 1ps
/****************************** C E C S  4 4 0 ******************************
 *
 * File Name:  MIPS_SIMD.v
 * Project:    SIMD MIPS
 * Designer:   Thomas Nguyen
 * Email:      tholinngu@gmail.com
 * Rev. No.:   Version 1.0
 * Rev. Date:  04/18/2019
 *
 * Purpose: An SIMD version of the MIPS_32 module. This will do similar ALU
 * instructions outputting 32 bits in vectors.
 *
 * Notes:
 *
 ****************************************************************************/
module MIPS_SIMD(SIMD_Sel, S, T, FS, Y);
   input       [1:0] SIMD_Sel; //SIMD mode select
   input      [31:0] S, T;     //Inputs
   input       [4:0] FS;       //FS opcode
   output reg [31:0] Y;        //Output

   //Integers to store Inputs
   integer           int_S, int_T;

   parameter
   //SIMD Modes
   SIMD8 = 2'b01,  SIMD16 = 2'b10,
   PASS_S = 5'h00, PASS_T = 5'h01,
   //SIMD8
   ADD8 = 5'h02,   SUB8 = 5'h04,

   AND8 = 5'h08,   OR8 = 5'h09,    XOR8 = 5'h0a,    NOR8 = 5'h0b,
   SRL8 = 5'h0c,   SRA8 = 5'h0d,   SLL8 = 5'h0e,
   //SIMD16
   ADD16 = 5'h02,  SUB16 = 5'h04,
   AND16 = 5'h08,  OR16 = 5'h09,   XOR16 = 5'h0a,   NOR16 = 5'h0b,

   //Other Operations
   INC_8 = 5'h0f,  INC4_8 = 5'h10, DEC_8 = 5'h11,   DEC4_8= 5'h12,
   INC_16 = 5'h0f, INC4_16 = 5'h10, DEC_16 = 5'h11, DEC4_16= 5'h12,
   ZEROS = 5'h13,  ONES = 5'h14;

   always@(S or T or FS or SIMD_Sel) begin
      int_S = S;
      int_T = T;
      case(SIMD_Sel)
         SIMD8: begin
            case(FS)
               PASS_S : Y = S;
```

```
PASS_T : Y = T;
ADD8   : begin
            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
            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
            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
            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
XOR8   : begin
            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
NOR8   : begin
            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

INC_8  : begin
            Y[7:0] = S[7:0] + 1'b1;
            Y[15:8] = S[15:8] + 1'b1;
            Y[23:16] = S[23:16] + 1'b1;
            Y[31:24] = S[31:24] + 1'b1;
         end

INC4_8 : begin
            Y[7:0] = S[7:0] + 4;
            Y[15:8] = S[15:8] + 4;
            Y[23:16] = S[23:16] + 4;
            Y[31:24] = S[31:24] + 4;
```

```
               end

      DEC_8  : begin
                  Y[7:0] = S[7:0] - 1'b1;
                  Y[15:8] = S[15:8] - 1'b1;
                  Y[23:16] = S[23:16] - 1'b1;
                  Y[31:24] = S[31:24] - 1'b1;
               end

      DEC4_8 : begin
                  Y[7:0] = S[7:0] - 4;
                  Y[15:8] = S[15:8] - 4;
                  Y[23:16] = S[23:16] - 4;
                  Y[31:24] = S[31:24] - 4;
               end

      default: Y = S; //Default Pass_S
   endcase
end //SIMD8 end
SIMD16: begin
   case(FS)
      PASS_S : Y = S;
      PASS_T : Y = T;
      ADD16  : begin
                  Y[15:0] = S[15:0] + T[15:0];
                  Y[31:16] = S[31:16] + T[31:16];
               end
      SUB16  : begin
                  Y[15:0] = S[15:0] - T[15:0];
                  Y[31:16] = S[31:16] - T[31:16];
               end
      AND16  : begin
                  Y[15:0] = S[15:0] & T[15:0];
                  Y[31:16] = S[31:16] & T[31:16];
               end
      OR16   : begin
                  Y[15:0] = S[15:0] | T[15:0];
                  Y[31:16] = S[31:16] | T[31:16];
               end
      XOR16  : begin
                  Y[15:0] = S[15:0] ^ T[15:0];
                  Y[31:16] = S[31:16] ^ T[31:16];
               end
      NOR16  : begin
                  Y[15:0] = ~(S[15:0] | T[15:0]);
                  Y[31:16] = ~(S[31:16] | T[31:16]);
               end
      INC_16 : begin
                  Y[15:0] = S[15:0] + 1'b1;
                  Y[31:16] = S[31:16] + 1'b1;
```

```
                        end
               INC4_16: begin
                           Y[15:0] = S[15:0] + 4;
                           Y[31:16] = S[31:16] + 4;
                        end
               DEC_16 : begin
                           Y[15:0] = S[15:0] - 1'b1;
                           Y[31:16] = S[31:16] - 1'b1;
                        end
               DEC4_16: begin
                           Y[15:0] = S[15:0] - 4;
                           Y[31:16] = S[31:16] - 4;
                        end

               default: Y = S; //Default Pass_S
            endcase
         end //SIMD16 end
         default: begin //Normal mode
               Y = S;
            end
         endcase
      end //always end

endmodule
```

```
`timescale 1ns / 1ps
/****************************** C E C S  4 4 0 ******************************
 *
 * File Name:  BarrellShift_SIMD.v
 * Project:    Final_Project
 * Designer:   Thomas Nguyen and Reed Ellison
 * Email:      Tholinngu@gmail.com and notwreed@gmail.com
 * Rev. No.:   Version 1.0
 * Rev. Date:  04/20/2019
 *
 * Purpose: SIMD Barrel Shifter Module to shift vector content in registers.
 *
 * NoDes:
 ****************************************************************************/
module BarrelShift_SIMD(simd_sel, D, type, shamt, Out);
   input [1:0] simd_sel;
   input       [4:0]   type;           // Function Select from MCU
   input       [4:0]   shamt;          // Shifting amount from IR[10:6]
   input      [31:0]   D;              // Data input for shifiDing
   output reg [31:0]   Out;            // Shifted Output

   always@(*)
      case(simd_sel)
      2'b01: //simd8
         case(type)
            5'h0E: // Shift Left Logical
               case(shamt)
                  5'd0:  Out = D;
                  5'd1:  {Out[31:24], Out[23:16], Out[15:8], Out[7:0]}
                             = {{D[30:24], 1'b0}, {D[22:16], 1'b0},
                                {D[14:8],  1'b0}, {D[6:0],   1'b0}};
                  5'd2:  {Out[31:24], Out[23:16], Out[15:8], Out[7:0]}
                             = {{D[29:24], 2'b0}, {D[21:16], 2'b0},
                                {D[13:8],  2'b0}, {D[5:0],   2'b0}};
                  5'd3:  {Out[31:24], Out[23:16], Out[15:8], Out[7:0]}
                             = {{D[28:24], 3'b0}, {D[20:16], 3'b0},
                                {D[12:8],  3'b0}, {D[4:0],   3'b0}};
                  5'd4:  {Out[31:24], Out[23:16], Out[15:8], Out[7:0]}
                             = {{D[27:24], 4'b0}, {D[19:16], 4'b0},
                                {D[11:8],  4'b0}, {D[3:0],   4'b0}};
                  5'd5:  {Out[31:24], Out[23:16], Out[15:8], Out[7:0]}
                             = {{D[26:24], 5'b0}, {D[18:16], 5'b0},
                                {D[10:8],  5'b0}, {D[2:0],   5'b0}};
                  5'd6:  {Out[31:24], Out[23:16], Out[15:8], Out[7:0]}
                             = {{D[25:24], 5'b0}, {D[17:16], 5'b0},
                                {D[9:8],   5'b0}, {D[1:0],   5'b0}};
                  5'd7:  {Out[31:24], Out[23:16], Out[15:8], Out[7:0]}
                             = {{D[24], 6'b0}, {D[17], 6'b0},
                                {D[8],  6'b0}, {D[0],  6'b0}};
                  default: Out = 32'b0;
               endcase
            5'h0C: // Shift Right Logical
               case(shamt)
                  5'd0:  Out = D;
                  5'd1:  {Out[31:24], Out[23:16], Out[15:8], Out[7:0]}
                             = {{1'b0, D[31:25]}, {1'b0, D[23:17]},
                                {1'b0, D[15:9]},  {1'b0, D[7:1]}};
                  5'd2:  {Out[31:24], Out[23:16], Out[15:8], Out[7:0]}
                             = {{2'b0, D[31:26]}, {2'b0, D[23:18]},
                                {2'b0, D[15:10]}, {2'b0, D[7:2]}};
                  5'd3:  {Out[31:24], Out[23:16], Out[15:8], Out[7:0]}
                             = {{3'b0, D[31:27]}, {3'b0, D[23:19]},
                                {3'b0, D[15:11]}, {3'b0, D[7:3]}};
                  5'd4:  {Out[31:24], Out[23:16], Out[15:8], Out[7:0]}
                             = {{4'b0, D[31:28]}, {4'b0, D[23:20]},
                                {4'b0, D[15:12]}, {4'b0, D[7:4]}};
                  5'd5:  {Out[31:24], Out[23:16], Out[15:8], Out[7:0]}
                             = {{5'b0, D[31:29]}, {5'b0, D[23:21]},
                                {5'b0, D[15:13]}, {5'b0, D[7:5]}};
                  5'd6:  {Out[31:24], Out[23:16], Out[15:8], Out[7:0]}
                             = {{6'b0, D[31:30]}, {6'b0, D[23:22]},
                                {6'b0, D[15:14]}, {6'b0, D[7:6]}};
                  5'd7:  {Out[31:24], Out[23:16], Out[15:8], Out[7:0]}
                             = {{7'b0, D[31]}, {7'b0, D[23]},
                                {7'b0, D[15]}, {7'b0, D[7]}};
                  default: Out = 32'b0;
               endcase
            5'h0D: // Shift Right Arithmetic
               case(shamt)
                  5'd0:  Out = D;
                  5'd1:  {Out[31:24], Out[23:16], Out[15:8], Out[7:0]}
                             = {{D[31], D[31:25]}, {D[23], D[23:17]},
                                {D[15], D[15:9]},  {D[7], D[7:1]}};
                  5'd2:  {Out[31:24], Out[23:16], Out[15:8], Out[7:0]}
                             = {{{2{D[31]}}, D[31:26]}, {{2{D[23]}}, D[23:18]},
                                {{2{D[15]}}, D[15:10]}, {{2{D[7]}}, D[7:2]}};
                  5'd3:  {Out[31:24], Out[23:16], Out[15:8], Out[7:0]}
                             = {{{3{D[31]}}, D[31:27]}, {{3{D[23]}}, D[23:19]},
                                {{3{D[15]}}, D[15:11]}, {{3{D[7]}}, D[7:3]}};
                  5'd4:  {Out[31:24], Out[23:16], Out[15:8], Out[7:0]}
                             = {{{4{D[31]}}, D[31:28]}, {{4{D[23]}}, D[23:20]},
                                {{4{D[15]}}, D[15:12]}, {{4{D[7]}}, D[7:4]}};
                  5'd5:  {Out[31:24], Out[23:16], Out[15:8], Out[7:0]}
                             = {{{5{D[31]}}, D[31:29]}, {{5{D[23]}}, D[23:21]},
                                {{5{D[15]}}, D[15:13]}, {{5{D[7]}}, D[7:5]}};
                  5'd6:  {Out[31:24], Out[23:16], Out[15:8], Out[7:0]}
                             = {{{6{D[31]}}, D[31:30]}, {{6{D[23]}}, D[23:22]},
                                {{6{D[15]}}, D[15:14]}, {{6{D[7]}}, D[7:6]}};
                  default:  Out = {{{7{D[31]}}, D[31]}, {{7{D[23]}}, D[23]},
                                  {{7{D[15]}}, D[15]},  {{7{D[7]}}, D[7]}};
               endcase
            5'h1A: // Rotate Left
               case(shamt)
                  5'd0:  Out = D;
                  5'd1:  {Out[31:24], Out[23:16], Out[15:8], Out[7:0]}
                             = {{D[30:24], D[31]}, {D[22:16], D[23]},
                                {D[14:8],  D[15]}, {D[6:0],   D[7]}};
                  5'd2:  {Out[31:24], Out[23:16], Out[15:8], Out[7:0]}
                             = {{D[29:24], D[31:30]}, {D[21:16], D[23:22]},
                                {D[13:8],  D[15:14]}, {D[5:0],   D[7:6]}};
                  5'd3:  {Out[31:24], Out[23:16], Out[15:8], Out[7:0]}
                             = {{D[28:24], D[31:29]}, {D[20:16], D[23:21]},
                                {D[12:8],  D[15:13]}, {D[4:0],   D[7:5]}};
                  5'd4:  {Out[31:24], Out[23:16], Out[15:8], Out[7:0]}
                             = {{D[27:24], D[31:28]}, {D[19:16], D[23:20]},
                                {D[11:8],  D[15:12]}, {D[3:0],   D[7:4]}};
                  5'd5:  {Out[31:24], Out[23:16], Out[15:8], Out[7:0]}
                             = {{D[26:24], D[31:27]}, {D[18:16], D[23:19]},
                                {D[10:8],  D[15:11]}, {D[2:0],   D[7:3]}};
                  5'd6:  {Out[31:24], Out[23:16], Out[15:8], Out[7:0]}
                             = {{D[25:24], D[31:26]}, {D[17:16], D[23:18]},
                                {D[9:8],   D[15:10]}, {D[1:0],   D[7:2]}};
                  5'd7:  {Out[31:24], Out[23:16], Out[15:8], Out[7:0]}
                             = {{D[24], D[31:25]}, {D[16], D[23:17]},
                                {D[8],  D[15:9]},  {D[0],  D[7:1]}};
                  default: Out = D;
               endcase
            5'h1B: // Rotate Right
               case(shamt)
                  5'd0:  Out = D;
                  5'd1:  {Out[31:24], Out[23:16], Out[15:8], Out[7:0]}
                             = {{D[24], D[31:25]}, {D[16], D[23:17]},
                                {D[8],  D[15:9]},  {D[0],  D[7:1]}};
                  5'd2:  {Out[31:24], Out[23:16], Out[15:8], Out[7:0]}
                             = {{D[25:24], D[31:26]}, {D[17:16], D[23:18]},
                                {D[9:8],   D[15:10]}, {D[1:0],   D[7:2]}};
                  5'd3:  {Out[31:24], Out[23:16], Out[15:8], Out[7:0]}
                             = {{D[26:24], D[31:27]}, {D[18:17], D[23:19]},
                                {D[10:8],  D[15:11]}, {D[2:0],   D[7:3]}};
                  5'd4:  {Out[31:24], Out[23:16], Out[15:8], Out[7:0]}
                             = {{D[27:24], D[31:28]}, {D[19:17], D[23:20]},
                                {D[11:8],  D[15:12]}, {D[3:0],   D[7:4]}};
                  5'd5:  {Out[31:24], Out[23:16], Out[15:8], Out[7:0]}
                             = {{D[28:24], D[31:29]}, {D[20:17], D[23:21]},
                                {D[12:8],  D[15:13]}, {D[4:0],   D[7:5]}};
                  5'd6:  {Out[31:24], Out[23:16], Out[15:8], Out[7:0]}
                             = {{D[29:24], D[31:30]}, {D[21:17], D[23:22]},
                                {D[13:8],  D[15:14]}, {D[5:0],   D[7:6]}};
                  5'd7:  {Out[31:24], Out[23:16], Out[15:8], Out[7:0]}
                             = {{D[30:24], D[31]}, {D[22:17], D[23]},
                                {D[14:8],  D[15]}, {D[6:0],   D[7]}};
                  default: Out = D;
               endcase
         endcase
      2'b10: //simd16
         case(type)
            5'h0E: // Shift Left Logical
               case(shamt)     // 31:16 bits,            15:0 bits
                  5'd0:  Out = D;
                  5'd1:  {Out[31:16], Out[15:0]} =
                             {{D[30:16], 1'b0}, {D[14:0], 1'b0}};
                  5'd2:  {Out[31:16], Out[15:0]} =
                             {{D[29:16], 2'b0}, {D[13:0], 2'b0}};
                  5'd3:  {Out[31:16], Out[15:0]} =
                             {{D[28:16], 3'b0}, {D[12:0], 3'b0}};
                  5'd4:  {Out[31:16], Out[15:0]} =
                             {{D[27:16], 4'b0}, {D[11:0], 4'b0}};
                  5'd5:  {Out[31:16], Out[15:0]} =
                             {{D[26:16], 5'b0}, {D[10:0], 5'b0}};
                  5'd6:  {Out[31:16], Out[15:0]} =
                             {{D[25:16], 6'b0}, {D[9:0], 6'b0}};
                  5'd7:  {Out[31:16], Out[15:0]} =
                             {{D[24:16], 7'b0}, {D[8:0], 7'b0}};
                  5'd8:  {Out[31:16], Out[15:0]} =
                             {{D[23:16], 8'b0}, {D[7:0], 8'b0}};
                  5'd9:  {Out[31:16], Out[15:0]} =
                             {{D[22:16], 9'b0}, {D[6:0], 9'b0}};
                  5'd10:  {Out[31:16], Out[15:0]} =
                             {{D[21:16], 10'b0}, {D[5:0], 10'b0}};
                  5'd11:  {Out[31:16], Out[15:0]} =
                             {{D[20:16], 11'b0}, {D[4:0], 11'b0}};
                  5'd12:  {Out[31:16], Out[15:0]} =
                             {{D[19:16], 12'b0}, {D[3:0], 12'b0}};
                  5'd13:  {Out[31:16], Out[15:0]} =
                             {{D[18:16], 13'b0}, {D[2:0], 13'b0}};
                  5'd14:  {Out[31:16], Out[15:0]} =
                             {{D[17:16], 14'b0}, {D[1:0], 14'b0}};
                  5'd15:  {Out[31:16], Out[15:0]} =
                             {{D[16], 15'b0}, {D[0], 15'b0}};
                  default: Out = 32'b0;
               endcase
            5'h0C: // Shift Right Logical
               case(shamt)     // 31:16 bits,            15:0 bits
                  5'd0:  Out = D;
                  5'd1:  {Out[31:16],Out[15:0]} =
                             {{1'b0, D[31:17]}, {1'b0, D[15:1]}};
                  5'd2:  {Out[31:16], Out[15:0]} =
                             {{2'b0, D[31:18]}, {2'b0, D[15:2]}};
                  5'd3:  {Out[31:16], Out[15:0]} =
                             {{3'b0, D[31:19]}, {3'b0, D[15:3]}};
                  5'd4:  {Out[31:16], Out[15:0]} =
                             {{4'b0, D[31:20]}, {4'b0, D[15:4]}};
                  5'd5:  {Out[31:16], Out[15:0]} =
                             {{5'b0, D[31:21]}, {5'b0, D[15:5]}};
                  5'd6:  {Out[31:16], Out[15:0]} =
                             {{6'b0, D[31:22]}, {6'b0, D[15:6]}};
                  5'd7:  {Out[31:16], Out[15:0]} =
                             {{7'b0, D[31:23]}, {7'b0, D[15:7]}};
                  5'd8:  {Out[31:16], Out[15:0]} =
                             {{8'b0, D[31:24]}, {8'b0, D[15:8]}};
                  5'd9:  {Out[31:16], Out[15:0]} =
                             {{9'b0, D[31:25]}, {9'b0, D[15:9]}};
                  5'd10:  {Out[31:16], Out[15:0]} =
                             {{10'b0, D[31:26]}, {10'b0, D[15:10]}};
                  5'd11:  {Out[31:16], Out[15:0]} =
                             {{11'b0, D[31:27]}, {11'b0, D[15:11]}};
                  5'd12:  {Out[31:16], Out[15:0]} =
                             {{12'b0, D[31:28]}, {12'b0, D[15:12]}};
                  5'd13:  {Out[31:16], Out[15:0]} =
                             {{13'b0, D[31:29]}, {13'b0, D[15:13]}};
                  5'd14:  {Out[31:16], Out[15:0]} =
                             {{14'b0, D[31:30]}, {14'b0, D[15:14]}};
                  5'd15:  {Out[31:16], Out[15:0]} =
                             {{15'b0, D[31]},    {15'b0, D[15]}};
                  default: Out = 32'b0;
               endcase
            5'h0D: // Shift Right Arithmetic
               case(shamt)     // 31:16 bits,            15:0 bits
                  5'd0:  Out = D;
                  5'd1:  {Out[31:16], Out[15:0]} =
                             {{{1{D[31]}}, D[31:17]}, {{1{D[15]}}, D[15:1]}};
                  5'd2:  {Out[31:16], Out[15:0]} =
                             {{{2{D[31]}}, D[31:18]}, {{2{D[15]}}, D[15:2]}};
                  5'd3:  {Out[31:16], Out[15:0]} =
                             {{{3{D[31]}}, D[31:19]}, {{3{D[15]}}, D[15:3]}};
                  5'd4:  {Out[31:16], Out[15:0]} =
                             {{{4{D[31]}}, D[31:20]}, {{4{D[15]}}, D[15:4]}};
                  5'd5:  {Out[31:16], Out[15:0]} =
                             {{{5{D[31]}}, D[31:21]}, {{5{D[15]}}, D[15:5]}};
                  5'd6:  {Out[31:16], Out[15:0]} =
                             {{{6{D[31]}}, D[31:22]}, {{6{D[15]}}, D[15:6]}};
                  5'd7:  {Out[31:16], Out[15:0]} =
                             {{{7{D[31]}}, D[31:23]}, {{7{D[15]}}, D[15:7]}};
                  5'd8:  {Out[31:16], Out[15:0]} =
                             {{{8{D[31]}}, D[31:24]}, {{8{D[15]}}, D[15:8]}};
                  5'd9:  {Out[31:16], Out[15:0]} =
                             {{{9{D[31]}}, D[31:25]}, {{9{D[15]}}, D[15:9]}};
                  5'd10:  {Out[31:16], Out[15:0]} =
                             {{{10{D[31]}}, D[31:26]}, {{10{D[15]}}, D[15:10]}};
                  5'd11:  {Out[31:16], Out[15:0]} =
                             {{{11{D[31]}}, D[31:27]}, {{11{D[15]}}, D[15:11]}};
                  5'd12:  {Out[31:16], Out[15:0]} =
                             {{{12{D[31]}}, D[31:28]}, {{12{D[15]}}, D[15:12]}};
                  5'd13:  {Out[31:16], Out[15:0]} =
                             {{{13{D[31]}}, D[31:29]}, {{13{D[15]}}, D[15:13]}};
                  5'd14:  {Out[31:16], Out[15:0]} =
                             {{{14{D[31]}}, D[31:30]}, {{14{D[15]}}, D[15:14]}};
                  5'd15:  {Out[31:16], Out[15:0]} =
                             {{{15{D[31]}}, D[31]},    {{15{D[15]}}, D[15]}};
                  default:  Out = {{16{D[31]}}, {16{D[15]}}};
               endcase
            5'h1A: // Rotate Left
               case(shamt)
                  5'd0:  Out = D;
                  5'd1:  {Out[31:16], Out[15:0]} =
                             {{D[30:16], D[31]},    {D[14:0], D[15]}};
                  5'd2:  {Out[31:16], Out[15:0]} =
                             {{D[29:16], D[31:30]}, {D[13:0], D[15:14]}};
                  5'd3:  {Out[31:16], Out[15:0]} =
                             {{D[28:16], D[31:29]}, {D[12:0], D[15:13]}};
                  5'd4:  {Out[31:16], Out[15:0]} =
                             {{D[27:16], D[31:28]}, {D[11:0], D[15:12]}};
                  5'd5:  {Out[31:16], Out[15:0]} =
                             {{D[26:16], D[31:27]}, {D[10:0], D[15:11]}};
                  5'd6:  {Out[31:16], Out[15:0]} =
                             {{D[25:16], D[31:26]}, {D[9:0],  D[15:10]}};
                  5'd7:  {Out[31:16], Out[15:0]} =
                             {{D[24:16], D[31:25]}, {D[8:0],  D[15:9]}};
                  5'd8:  {Out[31:16], Out[15:0]} =
                             {{D[23:16], D[31:24]}, {D[7:0],  D[15:8]}};
                  5'd9:  {Out[31:16], Out[15:0]} =
                             {{D[22:16], D[31:23]}, {D[6:0],  D[15:7]}};
                  5'd10:  {Out[31:16], Out[15:0]} =
                             {{D[21:16], D[31:22]}, {D[5:0], D[15:6]}};
                  5'd11:  {Out[31:16], Out[15:0]} =
                             {{D[20:16], D[31:21]}, {D[4:0], D[15:5]}};
                  5'd12:  {Out[31:16], Out[15:0]} =
                             {{D[19:16], D[31:20]}, {D[3:0], D[15:4]}};
                  5'd13:  {Out[31:16], Out[15:0]} =
                             {{D[18:16], D[31:19]}, {D[2:0], D[15:3]}};
                  5'd14:  {Out[31:16], Out[15:0]} =
                             {{D[17:16], D[31:18]}, {D[1:0], D[15:2]}};
                  5'd15:  {Out[31:16], Out[15:0]} =
                             {{D[16],    D[31:17]}, {D[0], D[15:1]}};
                  default: Out = D;
               endcase
            5'h1B: // Rotate Right
               case(shamt)
                  5'd0:  Out = D;
                  5'd1:  {Out[31:16], Out[15:0]} =
                             {{D[16], D[31:17]},    {D[0], D[15:1]}};
                  5'd2:  {Out[31:16], Out[15:0]} =
                             {{D[17:16], D[31:18]}, {D[1:0], D[15:2]}};
                  5'd3:  {Out[31:16], Out[15:0]} =
                             {{D[18:16], D[31:19]}, {D[2:0], D[15:3]}};
                  5'd4:  {Out[31:16], Out[15:0]} =
                             {{D[19:16], D[31:20]}, {D[3:0], D[15:4]}};
                  5'd5:  {Out[31:16], Out[15:0]} =
                             {{D[20:16], D[31:21]}, {D[4:0], D[15:5]}};
                  5'd6:  {Out[31:16], Out[15:0]} =
                             {{D[21:16], D[31:22]}, {D[5:0], D[15:6]}};
                  5'd7:  {Out[31:16], Out[15:0]} =
                             {{D[22:16], D[31:23]}, {D[6:0], D[15:7]}};
                  5'd8:  {Out[31:16], Out[15:0]} =
                             {{D[23:16], D[31:24]}, {D[7:0], D[15:8]}};
                  5'd9:  {Out[31:16], Out[15:0]} =
                             {{D[24:16], D[31:25]}, {D[8:0], D[15:9]}};
                  5'd10:  {Out[31:16], Out[15:0]} =
                             {{D[25:16], D[31:26]}, {D[9:0], D[15:10]}};
                  5'd11:  {Out[31:16], Out[15:0]} =
                             {{D[26:16], D[31:27]}, {D[10:0], D[15:11]}};
                  5'd12:  {Out[31:16], Out[15:0]} =
                             {{D[27:16], D[31:28]}, {D[11:0], D[15:12]}};
                  5'd13:  {Out[31:16], Out[15:0]} =
                             {{D[28:16], D[31:29]}, {D[12:0], D[15:13]}};
                  5'd14:  {Out[31:16], Out[15:0]} =
                             {{D[29:16], D[31:30]}, {D[13:0], D[15:14]}};
                  5'd15:  {Out[31:16], Out[15:0]} =
                             {{D[30:16], D[31]},    {D[14:0], D[15]}};
                  default: Out = D;
               endcase
         endcase
         default : Out = D;
      endcase

endmodule
```

C. Memory Modules

These are the instruction memory modules used to verify instruction execution and interrupt handling.

1. Memory Module 01

```
Instruction Memory
@0
3c 01 12 34  // main:      lui  $01, 0x1234
34 21 56 78  //            ori  $01, 0x5678        # LI   R01,  0x12345678
3c 02 87 65  //            lui  $02, 0x8765
34 42 43 21  //            ori  $02, 0x4321        # LI   R02,  0x87654321
00 01 18 20  //            add  $03, $00, $01      # COPY R03, R01

10 22 00 01  //            beq  $01, $02, no_eq    # should not branch
10 23 00 03  //            beq  $01, $03, yes_eq   # should branch
3c 0e ff ff  // no_eq:     lui  $14, 0xFFFF
35 ce ff ff  //            ori  $14, 0xFFFF        # LI   R14,  0xFFFFFFFF  "fail flag"
00 00 00 0d  //            breaK

00 00 70 20  // yes_eq:    add  $14, $0, $0        # CLR  R14  "pass flag"

14 23 00 01  //            bne  $01, $03, no_ne    # should not branch
14 22 00 03  //            bne  $01, $02, yes_ne   # should branch
3c 0f ff ff  // no_ne:     lui  $15, 0xFFFF
35 ef ff ff  //            ori  $15, 0xFFFF        # LI   R15,  0xFFFFFFFF  "fail flag"
00 00 00 0d  //            break

00 00 78 20  // yes_ne:    add  $15, $0, $0        # CLR  R15  "pass flag"
3c 0d 10 01  //            lui  $13, 0x1001
35 ad 00 c0  //            ori  $13, 0x00C0        # LI   R13,  0x100100C0
ad a1 00 00  //            sw   $01, 0($13)        # ST  [R13], R01
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  //0x04
24 68 AC E0    // 0x14:17
13 57 9B DF    // 0x18:1B
0F 0F 0F 0F    // 0x1C:1F
F0 F0 F0 F0    // 0x20:23  //0x08
00 00 00 09    // 0x24:27
00 00 00 0A    // 0x28:2B
00 00 00 0B    // 0x2C:2F
00 00 00 0C    // 0x30:33  //0x0C
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
```

2. Memory Module 02

Instruction Memory

```
@0
3c 01 ff ff  // main: lui  $01, 0xFFFF
34 21 ff ff  //       ori  $01, 0xFFFF       # LI   R01,  0xFFFFFFF
20 02 00 10  //       addi $02, $00, 0x10    # LI   R02,  0x10
3c 0f 10 01  //       lui  $15, 0x1001
35 ef 00 c0  //       ori  $15, 0x00C0       # LI   R15,  0x100100C0

00 01 08 42  // top:  srl  $01, $01, 1       # logical shift right 1 bit
ad e1 00 00  //       sw   $01, 0($15)       # ST  [R15], R01
21 ef 00 04  //       addi $15, $15, 4       # inc memory pointer 4 bytes
20 42 ff ff  //       addi $02, $02, -1      # decrement the loop counter
14 40 ff fb  //       bne  $02, $00, top     #  and jmp to top if not finished
08 10 00 0c  //       j       exit           # jump around a halt instruction
00 00 00 0d  //       break

3c 0e 5a 5a  // exit: lui  $14, 0x5A5A
35 ce 3c 3c  //       ori  $14, 0x3C3C       # LI   R14,  0x5A5A3C3C
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  //0x04
24 68 AC E0    // 0x14:17
13 57 9B DF    // 0x18:1B
0F 0F 0F 0F    // 0x1C:1F
F0 F0 F0 F0    // 0x20:23  //0x08
00 00 00 09    // 0x24:27
00 00 00 0A    // 0x28:2B
00 00 00 0B    // 0x2C:2F
00 00 00 0C    // 0x30:33  //0x0C
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
```

3. Memory Module 03

Instruction Memory

```
@0
3c 01 80 00  // main:         lui  $01, 0x8000
34 21 ff ff  //               ori  $01, 0xFFFF       # LI   R01,  0x8000FFFF
20 02 00 10  //               addi $02, $00, 0x10    # LI   R02,  0x10
3c 0f 10 01  //               lui  $15, 0x1001
35 ef 00 c0  //               ori  $15, 0x00C0       # LI   R15,  0x100100C0

00 01 08 43  // top:          sra  $01, $01, 1       # logical shift right 1 bit
ad e1 00 00  //               sw   $01, 0($15)       # ST  [R15], R01
21 ef 00 04  //               addi $15, $15, 4       # increment the memory pointer 4 bytes
20 42 ff ff  //               addi $02, $02, -1      # decrement the loop counter
14 40 ff fb  //               bne  $02, $00, top     #  and jmp to top if not finished

08 10 00 0c  //               j      exit            # jump around a halt instruction
00 00 00 0d  //               break

3c 0e 5a 5a  // exit:         lui  $14, 0x5A5A
35 ce 3c 3c  //               ori  $14, 0x3C3C       # LI   R14,  0x5A5A3C3C
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  //0x04
24 68 AC E0    // 0x14:17
13 57 9B DF    // 0x18:1B
0F 0F 0F 0F    // 0x1C:1F
F0 F0 F0 F0    // 0x20:23  //0x08
00 00 00 09    // 0x24:27
00 00 00 0A    // 0x28:2B
00 00 00 0B    // 0x2C:2F
00 00 00 0C    // 0x30:33  //0x0C
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
```

4. Memory Module 04

Instruction Memory

```
@0
3c 01 ff ff  // main:         lui  $01, 0xFFFF
34 21 ff ff  //               ori  $01, 0xFFFF      # LI   R01,  0xFFFFFFF
20 02 00 10  //               addi $02, $00, 0x10   # LI   R02,  0x10
3c 0f 10 01  //               lui  $15, 0x1001
35 ef 00 c0  //               ori  $15, 0x00C0      # LI   R15,  0x100100C0

00 01 08 40  // top:          sll  $01, $01, 1      # logical shift left 1 bit
ad e1 00 00  //               sw   $01, 0($15)      # ST  [R15], R01
21 ef 00 04  //               addi $15, $15, 4      # increment the memory pointer 4 bytes
20 42 ff ff  //               addi $02, $02, -1     # decrement the loop counter
00 02 18 2a  //               slt  $03, $00, $02    # r3 <--1 if r0 < r2
14 60 ff fa  //               bne  $03, $00, top    # jmp if r3==1

08 10 00 0d  //               j      exit           # jump around a halt instruction
00 00 00 0d  //               break

3c 0e 5a 5a  // exit:         lui  $14, 0x5A5A
35 ce 3c 3c  //               ori  $14, 0x3C3C      # LI   R14,  0x5A5A3C3C
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  //0x04
24 68 AC E0    // 0x14:17
13 57 9B DF    // 0x18:1B
0F 0F 0F 0F    // 0x1C:1F
F0 F0 F0 F0    // 0x20:23  //0x08
00 00 00 09    // 0x24:27
00 00 00 0A    // 0x28:2B
00 00 00 0B    // 0x2C:2F
00 00 00 0C    // 0x30:33  //0x0C
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
```

5. Memory Module 05

Instruction Memory

```
@0
3c 01 ff ff  // main:         lui  $01, 0xFFFF
34 21 ff ff  //               ori  $01, 0xFFFF       # LI   R01,  0xFFFFFFF
20 02 ff f0  //               addi $02, $00, -16     # LI   R02,  -16
3c 0f 10 01  //               lui  $15, 0x1001
35 ef 00 c0  //               ori  $15, 0x00C0       # LI   R15,  0x100100C0

00 01 08 40  // top:          sll  $01, $01, 1       # logical shift left 1 bit
ad e1 00 00  //               sw   $01, 0($15)       # ST  [R15], R01
21 ef 00 04  //               addi $15, $15, 4       # increment the memory pointer 4 bytes
20 42 00 01  //               addi $02, $02, 1       # increment the loop counter
28 43 00 00  //               slti $03, $02, 0       # r3 <--1 if r2 < 0
14 60 ff fa  //               bne  $03, $00, top     # jmp if r3==1

08 10 00 0d  //               j      exit            # jump around a halt instruction
00 00 00 0d  //               break

3c 0e 5a 5a  // exit:         lui  $14, 0x5A5A
35 ce 3c 3c  //               ori  $14, 0x3C3C       # LI   R14,  0x5A5A3C3C
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  //0x04
24 68 AC E0    // 0x14:17
13 57 9B DF    // 0x18:1B
0F 0F 0F 0F    // 0x1C:1F
F0 F0 F0 F0    // 0x20:23  //0x08
00 00 00 09    // 0x24:27
00 00 00 0A    // 0x28:2B
00 00 00 0B    // 0x2C:2F
00 00 00 0C    // 0x30:33  //0x0C
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
```

6. Memory Module 06

Instruction Memory

```
@0
3c 0f 10 01  // lui  $15, 0x1001
35 ef 00 00  // ori  $15, 0x0000      # LI   R15,  0x10010000  dest data pointer
3c 0e 10 01  // lui  $14, 0x1001
35 ce 00 c0  // ori  $14, 0x00C0      # LI   R14,  0x100100C0  dest data pointer
20 0d 00 10  // addi $13, $00, 16     # LI   R13,  16       loop counter
8d e1 00 04  // lw   $01, 04($15)     # Load
8d e2 00 08  // lw   $02, 08($15)     #   R01
8d e3 00 0c  // lw   $03, 12($15)     #      to
8d e4 00 10  // lw   $04, 16($15)     #      R12
8d e5 00 14  // lw   $05, 20($15)
8d e6 00 18  // lw   $06, 24($15)
8d e7 00 1c  // lw   $07, 28($15)
8d e8 00 20  // lw   $08, 32($15)
8d e9 00 24  // lw   $09, 36($15)
8d ea 00 28  // lw   $10, 40($15)
8d eb 00 2c  // lw   $11, 44($15)
8d ec 00 30  // lw   $12, 48($15)

               // mem2mem:
8d f1 00 00  // lw   $17, 00($15)     # do mem to
ad d1 00 00  // sw   $17, 00($14)     #   mem transfer
21 ef 00 04  // addi $15, $15, 04     # bump both source
21 ce 00 04  // addi $14, $14, 04     #   and dest pointers
21 ad ff ff  // addi $13, $13, -1     # dec the loop counter
15 a0 ff fa  // bne  $13, $00, mem2mem  #   and continue till done
00 00 00 0d  // break
```

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
```

7. Memory Module 07

Instruction Memory

```
@0
3c 0f 10 01  // main:         lui  $15, 0x1001
35 ef 00 00  //               ori  $15, 0x0000       # LI   R15,  0x10010000  dest data pointer
3c 0e 10 01  //               lui  $14, 0x1001
35 ce 00 c0  //               ori  $14, 0x00C0       # LI   R14,  0x100100C0  dest data pointer
20 0d 00 10  //               addi $13, $00, 16      # LI   R13,  16       loop counter
8d e1 00 04  //               lw   $01, 04($15)      # Load
8d e2 00 08  //               lw   $02, 08($15)      #   R01
8d e3 00 0c  //               lw   $03, 12($15)      #      to
8d e4 00 10  //               lw   $04, 16($15)      #      R12
8d e5 00 14  //               lw   $05, 20($15)
8d e6 00 18  //               lw   $06, 24($15)
8d e7 00 1c  //               lw   $07, 28($15)
8d e8 00 20  //               lw   $08, 32($15)
8d e9 00 24  //               lw   $09, 36($15)
8d ea 00 28  //               lw   $10, 40($15)
8d eb 00 2c  //               lw   $11, 44($15)
8d ec 00 30  //               lw   $12, 48($15)

0c 10 00 15  //               jal  mem2mem
3c 0f ff ff  //               lui  $15, 0xFFFF
35 ef ff ff  //               ori  $15, 0xFFFF       # LI   R15,  0xFFFFFFFF  "pass flag"
00 00 00 0d  //               break

8d f1 00 00  // mem2mem:   lw   $17, 00($15)         # do mem to
ad d1 00 00  //               sw   $17, 00($14)      #   mem transfer
21 ef 00 04  //               addi $15, $15, 04      # bump both source
21 ce 00 04  //               addi $14, $14, 04      #   and dest pointers
21 ad ff ff  //               addi $13, $13, -1      # dec the loop counter
15 a0 ff fa  //               bne  $13, $00, mem2mem  #   and continue till done
03 e0 00 08  //               jr   $31               # return to calling code
00 00 00 0d  //               break                  # safety net
```

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
```

8. Memory Module 08

Instruction Memory

```
@0
3c 0f 10 01  // main:         lui  $15, 0x1001
35 ef 00 00  //               ori  $15, 0x0000      # $r15 <-- 0x10010000 (src pointer)
8d e1 00 00  //               lw   $01, 00($15)     # $r01 <--     25
8d e2 00 04  //               lw   $02, 04($15)     # $r02 <--   1000
8d e3 00 08  //               lw   $03, 08($15)     # $r03 <--    -25
8d e4 00 0c  //               lw   $04, 12($15)     # $r04 <--  -1000
8d e5 00 10  //               lw   $05, 16($15)     # $r05 <--  25000
8d e6 00 14  //               lw   $06, 20($15)     # $r06 <-- -25000
8d e7 00 18  //               lw   $07, 24($15)     # $r07 <--     -1
00 22 00 18  //               mult $01, $02
00 00 40 12  //               mflo $08              # rs=pos rt=pos rd=pos
14 a8 00 10  //               bne  $05, $08, fail1
00 62 00 18  //               mult $03, $02
00 00 48 12  //               mflo $09              # rs=neg rt=pos rd=neg
00 00 50 10  //               mfhi $10
14 c9 00 0f  //               bne  $06, $09, fail2L
14 ea 00 11  //               bne  $07, $10, fail2H
00 24 00 18  //               mult $01, $04
00 00 58 12  //               mflo $11              # rs=pos rt=neg rd=neg
00 00 60 10  //               mfhi $12
14 cb 00 10  //               bne  $06, $11, fail3L
14 ec 00 12  //               bne  $07, $12, fail3H
00 64 00 18  //               mult $03, $04
00 00 68 12  //               mflo $13              # rs=neg rt=neg rd=pos
14 ad 00 12  //               bne  $05, $13, fail4

3c 0e 00 00  // pass:         lui  $14, 0x0000
35 ce 00 00  //               ori  $14, 0x0000      # $r14 <-- 0x00000000  (Pass flag)
00 00 00 0d  //               break
3c 0e ff ff  // fail1:        lui  $14, 0xFFFF
35 ce ff ff  //               ori  $14, 0xFFFF      # $r14 <-- 0xFFFFFFFF  (Fail flag 1)
00 00 00 0d  //               break
3c 0e ff ff  // fail2L:       lui  $14, 0xFFFF
35 ce ff fe  //               ori  $14, 0xFFFE      # $r14 <-- 0xFFFFFFFE  (Fail flag 2L)
00 00 00 0d  //               break
3c 0e ff ff  // fail2H:       lui  $14, 0xFFFF
35 ce ff fd  //               ori  $14, 0xFFFD      # $r14 <-- 0xFFFFFFFD  (Fail flag 2H)
00 00 00 0d  //               break
3c 0e ff ff  // fail3L:       lui  $14, 0xFFFF
35 ce ff fc  //               ori  $14, 0xFFFC      # $r14 <-- 0xFFFFFFFC  (Fail flag 3L)
00 00 00 0d  //               break
3c 0e ff ff  // fail3H:       lui  $14, 0xFFFF
35 ce ff fb  //               ori  $14, 0xFFFB      # $r14 <-- 0xFFFFFFFB  (Fail flag 3H)
00 00 00 0d  //               break
3c 0e ff ff  // fail4:        lui  $14, 0xFFFF
35 ce ff fa  //               ori  $14, 0xFFFA      # $r14 <-- 0xFFFFFFFA  (Fail flag 4)
00 00 00 0d  //               break
```

Data Memory

```
@0             // Big Endian Format

00 00 00 19    // 0x00:03  //word 00 =       25
00 00 03 E8    // 0x04:07  //word 01 =   1000
FF FF FF E7    // 0x08:0B  //word 02 =       -25
FF FF FC 18    // 0x0C:0F  //word 03 =  -1000
00 00 61 A8    // 0x10:13  //word 04 =  25000
FF FF 9E 58    // 0x14:17  //word 05 = -25000
FF FF FF FF    // 0x18:1B  //word 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
```

9. Memory Module 09

Instruction Memory

```
@0
3c 0f 10 01  // main:         lui  $15, 0x1001
35 ef 00 c0  //               ori  $15, 0x00C0      # $r15 <-- 0x100100C0  (dest pointer)
20 01 ff 8a  //               addi $01, $00, -118   # $r01 <-- 0xFFFFFF8A
20 02 00 8a  //               addi $02  $00,  138   # $r02 <-- 0x0000008A
0c 10 00 22  //               jal  slt_tests

3c 0d 77 88  //               lui  $13, 0x7788
35 ad 77 88  //               ori  $13, 0x7788      # $r13 <-- 0x77887788  (pattern1)
3c 0c 88 77  //               lui  $12, 0x8877
35 8c 88 77  //               ori  $12, 0x8877      # $r12 <-- 0x88778877  (pattern2)
3c 0b ff ff  //               lui  $11, 0xFFFF
35 6b ff ff  //               ori  $11, 0xFFFF      # $r11 <-- 0xFFFFFFFF  (pattern3)

01 ac 50 26  //               xor  $10, $13, $12    # $r10 <-- 0xFFFFFFFF
11 4b 00 02  //               beq  $10, $11, xor_pass
20 0e ff fb  //               addi $14, $00, -5     # fail flag5 r14 <-- 0xFFFF_FFFB
00 00 00 0d  //               break
01 ac 48 24  // xor_pass:  and  $09, $13, $12       # $r09 <-- 0x00000000
11 20 00 02  //               beq  $09, $00, and_pass
20 0e ff fa  //               addi $14, $00, -6     # fail flag6 r14 <-- 0xFFFF_FFFA
00 00 00 0d  //               break
01 e2 48 25  // and_pass:  or   $09, $15, $02       # $r09 <-- 0x100100CA
3c 08 10 01  //               lui  $08, 0x1001
35 08 00 ca  //               ori  $08, 0x00CA      # $r08 <-- 0x100100CA
11 09 00 02  //               beq  $08, $09, or_pass
20 0e ff f9  //               addi $14, $00, -7     # fail flag7 r14 <-- 0xFFFF_FFF9
00 00 00 0d  //               break
01 e2 48 27  // or_pass:   nor  $09, $15, $02       # $r09 <-- 0xEFFEFF35
3c 08 ef fe  //               lui  $08, 0xEFFE
35 08 ff 35  //               ori  $08, 0xFF35      # $r08 <-- 0xEFFEFF35
11 09 00 02  //               beq  $08, $09, nor_pass
20 0e ff f8  //               addi $14, $00, -8     # fail flag8 r14 <-- 0xFFFF_FFF8
00 00 00 0d  //               break
ad e8 00 10  // nor_pass:  sw   $08, 0x10($15)      # M[D0] <-- 0xEFFEFF35
00 00 70 20  //               add  $14, $00, $00    # clear r14 indicating "passed all"
00 00 00 0d  //               break                 # should stop here, having
               //                                   #   completed all the tests

00 22 18 2a  // slt_tests: slt  $03, $01, $02       # for signed# r01 < r02
14 60 00 02  //               bne  $03, $00, slt1   #   thus, we should branch
20 0e ff ff  //               addi $14, $00, -1     # fail flag1 r14 <-- FFFF_FFFF
00 00 00 0d  //               break
20 04 00 c0  // slt1:         addi $04, $00, 0xC0   # pass flag1 M[C0] <-- C0
ad e4 00 00  //               sw   $04, 0x00($15)

00 41 18 2b  //               sltu $03, $02, $01    # for unsigned# r02 < r01
14 60 00 02  //               bne  $03, $00, slt2   #   thus, we should branch
20 0e ff fe  //               addi $14, $00, -2     # fail flag2 r14 <-- FFFF_FFFE
00 00 00 0d  //               break
20 05 00 c4  // slt2:         addi $05, $00, 0xC4   # pass flag1 M[C4] <-- C4
ad e5 00 04  //               sw   $05, 0x04($15)
```

```
00 41 18 2a  //              slt  $03, $02, $01     # for signed# r02 !< r01
10 60 00 02  //              beq  $03, $00, slt3    #   thus, we should branch
20 0e ff fd  //              addi $14, $00, -3      # fail flag3 r14 <-- FFFF_FFFD
00 00 00 0d  //              break
20 06 00 c8  // slt3:        addi $06, $00, 0xC8   # pass flag3 M[C8] <-- C8
ad e6 00 08  //              sw   $06, 0x08($15)

00 22 18 2b  //              sltu $03, $01, $02     # for unsigned# r01 !< r02
10 60 00 02  //              beq  $03, $00, slt4    #   thus, we should branch
20 0e ff fc  //              addi $14, $00, -4      # fail flag4 r14 <-- FFFF_FFFC
00 00 00 0d  //              break
20 07 00 cc  // slt4:        addi $07, $00, 0xCC    # pass flag4 M[CC] <-- CC
ad e7 00 0c  //              sw   $07, 0x0C($15)
03 e0 00 08  //              jr   $31               # return from subroutine
```

Data Memory

```
@0             // Big Endian Format

00 04 09 11    // 0x00:03  //word 00 =  264465
00 00 03 E8    // 0x04:07  //word 01 =       1000
FF FB F6 EF    // 0x08:0B  //word 02 = -264465
FF FF FC 18    // 0x0C:0F  //word 03 =   -1000
00 00 01 08    // 0x10:13  //word 04 =       264   Quot1,4  w00 div w01, w02 div w03
00 00 01 D1    // 0x14:17  //word 05 =       465   Rem 1,3  w00 mod w01, w00 mod w03
FF FF FE F8    // 0x18:1B  //word 06 =       -264   Quot2,3 w02 div w01, w00 div w03
FF FF FE 2F    // 0x1C:1F  //word 07 =       -465   Rem 2,4 w02 mod w01, w02 mod w03
00 00 00 08    // 0x20:23  //word 08 =
00 00 00 09    // 0x24:27  //word 09 =
00 00 00 0A    // 0x28:2B  //word 10 =
00 00 00 0B    // 0x2C:2F  //word 11 =
00 00 00 0C    // 0x30:33  //word 12 =
00 00 00 0D    // 0x34:37  //word 13 =
00 00 00 0E    // 0x38:3B  //word 14 =
00 00 00 0F    // 0x3C:3F  //word 15 =

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

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

10. Memory Module 10

Instruction Memory

```
@0
3c 0f 10 01  // main:         lui  $15, 0x1001
35 ef 00 00  //               ori  $15, 0x0000      # $r15 <-- 0x10010000  (source pointer)
8d e1 00 00  //               lw   $01, 00($15)     # $r01 <--  264465
8d e2 00 04  //               lw   $02, 04($15)     # $r02 <--     1000
8d e3 00 08  //               lw   $03, 08($15)     # $r03 <-- -264465
8d e4 00 0c  //               lw   $04, 12($15)     # $r04 <--   -1000
8d e5 00 10  //               lw   $05, 16($15)     # $r05 <--     264   Quot1,4  w01 div w02,
w03 div w04
8d e6 00 14  //               lw   $06, 20($15)     # $r06 <--     465   Rem 1,3  w01 rem w02,
w01 rem w04
8d e7 00 18  //               lw   $07, 24($15)     # $r07 <--     -264   Quot2,3 w03 div w02,
w01 div w04
8d e8 00 1c  //               lw   $08, 28($15)     # $r08 <--     -465   Rem 2,4 w03 re0 w02,
w03 rem w04

00 22 00 1a  //               div  $01, $02
00 00 48 12  //               mflo $09              # rs=pos / rt=pos, rem=pos quot=pos
00 00 50 10  //               mfhi $10
15 25 00 16  //               bne  $09, $05, fail1Q
15 46 00 18  //               bne  $10, $06, fail1R

00 62 00 1a  //               div  $03, $02
00 00 48 12  //               mflo $09              # rs=neg / rt=pos, rem=neg quot=neg
00 00 50 10  //               mfhi $10
15 27 00 17  //               bne  $09, $07, fail2Q
15 48 00 19  //               bne  $10, $08, fail2R

00 24 00 1a  //               div  $01, $04
00 00 48 12  //               mflo $09              # rs=pos / rt=neg, rem=pos quot=neg
00 00 50 10  //               mfhi $10
15 27 00 18  //               bne  $09, $07, fail3Q
15 46 00 1a  //               bne  $10, $06, fail3R

00 64 00 1a  //               div  $03, $04
00 00 48 12  //               mflo $09              # rs=neg / rt=neg, rem=neg quot=pos
00 00 50 10  //               mfhi $10
15 25 00 19  //               bne  $09, $05, fail4Q
15 48 00 1b  //               bne  $10, $08, fail4R

3c 0b 00 00  // pass:         lui  $11, 0x0000
35 6b 00 00  //               ori  $11, 0x0000      # $r11 <-- 0x00000000  (Pass flag)
00 0b 60 20  //               add  $12, $00, $11    # $r12 <-- Pass
00 0b 68 20  //               add  $13, $00, $11    # $r13 <-- Pass
00 0b 70 20  //               add  $14, $00, $11    # $r14 <-- Pass
00 00 00 0d  //               break

3c 0e ff ff  // fail1Q:       lui  $14, 0xFFFF
35 ce ff ff  //               ori  $14, 0xFFFF      # $r14 <-- 0xFFFFFFFF  (Fail flag 1 Quot)
00 00 00 0d  //               break
3c 0e ff ff  // fail1R:       lui  $14, 0xFFFF
35 ce ff fe  //               ori  $14, 0xFFFE      # $r14 <-- 0xFFFFFFFE  (Fail flag 1 Rem)
00 00 00 0d  //               break
```

```
3c 0e ff ff  // fail2Q:       lui  $14, 0xFFFF
35 ce ff fd  //               ori  $14, 0xFFFD      # $r14 <-- 0xFFFFFFFD  (Fail flag 2 Quot)
00 00 00 0d  //               break
3c 0e ff ff  // fail2R:       lui  $14, 0xFFFF
35 ce ff fc  //               ori  $14, 0xFFFC      # $r14 <-- 0xFFFFFFFC  (Fail flag 2 Rem)
00 00 00 0d  //               break
3c 0e ff ff  // fail3Q:       lui  $14, 0xFFFF
35 ce ff fb  //               ori  $14, 0xFFFB      # $r14 <-- 0xFFFFFFFB  (Fail flag 3 Quot)
00 00 00 0d  //               break
3c 0e ff ff  // fail3R:       lui  $14, 0xFFFF
35 ce ff fa  //               ori  $14, 0xFFFA      # $r14 <-- 0xFFFFFFFA  (Fail flag 3 Rem)
00 00 00 0d  //               break
3c 0e ff ff  // fail4Q:       lui  $14, 0xFFFF
35 ce ff f9  //               ori  $14, 0xFFF9      # $r14 <-- 0xFFFFFFF9  (Fail flag 4 Quot)
00 00 00 0d  //               break
3c 0e ff ff  // fail4R:       lui  $14, 0xFFFF
35 ce ff f8  //               ori  $14, 0xFFF8      # $r14 <-- 0xFFFFFFF8  (Fail flag 4 Rem)
00 00 00 0d  //               break
```

Data memory

```
@0             // Big Endian Format

00 04 09 11    // 0x00:03  //word 00 =  264465
00 00 03 E8    // 0x04:07  //word 01 =       1000
FF FB F6 EF    // 0x08:0B  //word 02 = -264465
FF FF FC 18    // 0x0C:0F  //word 03 =   -1000
00 00 01 08    // 0x10:13  //word 04 =       264   Quot1,4  w00 div w01, w02 div w03
00 00 01 D1    // 0x14:17  //word 05 =       465   Rem 1,3  w00 mod w01, w00 mod w03
FF FF FE F8    // 0x18:1B  //word 06 =       -264   Quot2,3 w02 div w01, w00 div w03
FF FF FE 2F    // 0x1C:1F  //word 07 =       -465   Rem 2,4 w02 mod w01, w02 mod w03
00 00 00 08    // 0x20:23  //word 08 =
00 00 00 09    // 0x24:27  //word 09 =
00 00 00 0A    // 0x28:2B  //word 10 =
00 00 00 0B    // 0x2C:2F  //word 11 =
00 00 00 0C    // 0x30:33  //word 12 =
00 00 00 0D    // 0x34:37  //word 13 =
00 00 00 0E    // 0x38:3B  //word 14 =
00 00 00 0F    // 0x3C:3F  //word 15 =

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

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

11. Memory Module 11

Instruction Memory

```
@0
3c 0f 10 01  // main:         lui  $15, 0x1001
35 ef 00 c0  //               ori  $15, 0x00C0       # $r15 <-- 0x100100C0  (dest pointer)
20 01 ff 8a  //               addi $01, $00, -118    # $r01 <-- 0xFFFFFF8A
20 02 00 8a  //               addi $02  $00,  138    # $r02 <-- 0x0000008A
0c 10 00 1a  //               jal  sltiu_tests

3c 0d ff ff  //               lui  $13, 0xFFFF
35 ad 55 55  //               ori  $13, 0x5555       # $r13 <-- 0xFFFF5555  (pattern1)
3c 0c ff ff  //               lui  $12, 0xFFFF
35 8c fa f5  //               ori  $12, 0xFAF5       # $r12 <-- 0xFFFFFAF5  (pattern2)
3c 0b ff ff  //               lui  $11, 0xFFFF
35 6b ff ff  //               ori  $11, 0xFFFF       # $r11 <-- 0xFFFFFFFF  (pattern3)
3c 0a 00 00  //               lui  $10, 0x0000
35 4a f0 f0  //               ori  $10, 0xF0F0       # $r10 <-- 0x0000F0F0  (pattern4)

39 a9 aa aa  //               xori $09, $13, 0xAAAA  # $r09 <-- 0xFFFFFFFF
01 2b 40 22  //               sub  $08, $09, $11     # $r00 <-- 0
11 00 00 02  //               beq  $08, $00, xor_p1  # should branch
20 0e ff f9  //               addi $14, $00, -7      # fail flag7 r14 <-- FFFF_FFF9
00 00 00 0d  //               break
31 87 f5 fa  // xor_p1:       andi $07, $12, 0xF5FA  # $r07 <-- 0x0000F0F0
00 ea 40 22  //               sub  $08, $07, $10
11 00 00 02  //               beq  $08, $00, xor_p2  # should branch
20 0e ff f8  //               addi $14, $00, -8      # fail flag8 r14 <-- FFFF_FFF8
00 00 00 0d  //               break
ad e1 00 18  // xor_p2:       sw   $01, 0x18($15)    # M[D8] <-- FFFFFF8A
00 00 00 0d  //               break                  # should stop here, having
00 00 00 0d  //               break                  #   completed all the tests

               // sltiu_tests:
2c 23 ff 8b  //               sltiu  $03, $01, -117  # for unsigned# r01 < se(0xFF8B)
14 60 00 02  //               bne     $03, $00, slt1_p1  #   thus, we should branch
20 0e ff ff  //               addi   $14, $00, -1    # fail flag1 r14 <-- FFFF_FFFF
00 00 00 0d  //               break
20 04 00 c0  // slt1_p1:   addi   $04, $00, 0xC0     # pass flag1 M[C0] <-- C0
ad e4 00 00  //               sw      $04, 0x00($15)

2c 23 ff 89  //               sltiu  $03, $01, -119  # for unsigned# r01 !< se(0xFF89)
10 60 00 02  //               beq     $03, $00, slt_p2   #   thus, we should branch
20 0e ff fe  //               addi   $14, $00, -2    # fail flag2 r14 <-- FFFF_FFFE
00 00 00 0d  //               break
20 05 00 c4  // slt_p2:       addi   $05, $00, 0xC4  # pass flag2 M[C4] <-- C4
ad e5 00 04  //               sw      $05, 0x04($15)

2c 23 ff 8a  //               sltiu  $03, $01, -118  # for unsigned# r01 !< se(0xFF8A)
10 60 00 02  //               beq     $03, $00, slt_p3   #   thus, we should branch
20 0e ff fd  //               addi   $14, $00, -3    # fail flag3 r14 <-- FFFF_FFFD
00 00 00 0d  //               break
20 06 00 c8  // slt_p3:       addi   $06, $00, 0xC8  # pass flag3 M[C8] <-- C8
ad e6 00 08  //               sw      $06, 0x08($15)

2c 43 00 8b  //               sltiu  $03, $02, 0x008B   # for unsigned# r02 < se(0x008B)
14 60 00 02  //               bne     $03, $00, slt1_p4  #   thus, we should branch
20 0e ff fc  //               addi   $14, $00, -4    # fail flag4 r14 <-- FFFF_FFFC
00 00 00 0d  //               break
20 07 00 cc  // slt1_p4:   addi   $07, $00, 0xCC     # pass flag4 M[CC] <-- CC
ad e7 00 0c  //               sw      $07, 0x0C($15)

2c 43 00 89  //               sltiu  $03, $02, 0x0089   # for unsigned# r02 !< se(0x0089)
10 60 00 02  //               beq     $03, $00, slt_p5   #   thus, we should branch
```

```
20 0e ff fb  //               addi   $14, $00, -5    # fail flag5 r14 <-- FFFF_FFFB
00 00 00 0d  //               break
20 08 00 d0  // slt_p5:       addi   $08, $00, 0xD0  # pass flag5 M[D0] <-- D0
ad e8 00 10  //               sw      $08 0x10($15)

2c 43 00 8a  //               sltiu  $03, $02, 0x008A   # for unsigned# r02 !< se(0x008A)
10 60 00 02  //               beq     $03, $00, slt_p6   #   thus, we should branch
20 0e ff fa  //               addi   $14, $00, -6    # fail flag6 r14 <-- FFFF_FFFA
00 00 00 0d  //               break
20 06 00 d4  // slt_p6:       addi   $06, $00, 0xD4  # pass flag6 M[D4] <-- D4
ad e6 00 14  //               sw      $06, 0x14($15)
20 0e 00 00  //               addi   $14, $00, 0     # set $r14 to 0000_0000
03 e0 00 08  //               jr   $31               # return from subroutine
```

Data memory

```
@0             // Big Endian Format

00 04 09 11    // 0x00:03
00 00 03 E8    // 0x04:07
FF FB F6 EF    // 0x08:0B
FF FF FC 18    // 0x0C:0F
00 00 01 08    // 0x10:13
00 00 01 D1    // 0x14:17
FF FF FE F8    // 0x18:1B
FF FF FE 2F    // 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
```

12. Memory Module 12

Instruction Memory

```
@0
3c 0f 10 01  // main:         lui  $15, 0x1001
35 ef 00 c0  //               ori  $15, 0x00C0       # $r15 <-- 0x100100C0  (dest pointer)

20 01 ff 8a  //               addi $01, $00, -118    # $r01 <-- 0xFFFFFF8A
20 02 00 8a  //               addi $02  $00,  138    # $r02 <-- 0x0000008A
0c 10 00 08  //               jal  blt_tests
ad e1 00 18  //               sw   $01, 0x18($15)    # M[D8] <-- 0xFFFFFF8A
ad e2 00 1c  //               sw   $02, 0x1C($15)    # M[DC] <-- 0x0000008A
00 00 00 0d  //               break

18 20 00 02  // blt_tests: blez $01, blez_p1         # this should branch
20 0e ff ff  //               addi $14, $00, -1      # fail flag1 r14 <-- FFFF_FFFF
00 00 00 0d  //               break
20 03 00 c0  // blez_p1:   addi $03, $00, 0xC0       # pass flag1 M[C0] <-- C0
ad e3 00 00  //               sw   $03, 0x00($15)
18 40 00 03  //               blez $02, blez_f2      # this should not branch
20 04 00 c4  //               addi $04, $00, 0xC4    # pass flag2 M[C4] <-- C4
ad e4 00 04  //               sw   $04, 0x04($15)
08 10 00 13  //               j      blez_p2
20 0e ff fe  // blez_f2:   addi $14, $00, -2         # fail flag2 r14 <-- FFFF_FFFE
00 00 00 0d  //               break
18 00 00 02  // blez_p2:   blez $0, blez_p3          # this should branch
20 0e ff fd  //               addi $14, $00, -3      # fail flag3 r14 <-- FFFF_FFFD
00 00 00 0d  //               break
20 05 00 c8  // blez_p3:   addi $05, $00, 0xC8       # pass flag3 M[C8] <-- C8
ad e5 00 08  //               sw   $05, 0x08($15)

1c 40 00 02  //               bgtz $02, bgtz_p1      # this should pass
20 0e ff fc  //               addi $14, $00, -4      # fail flag3 r14 <-- FFFF_FFFC
00 00 00 0d  //               break
20 06 00 cc  // bgtz_p1:   addi $06, $00, 0xCC       # pass flag4 M[C0] <-- CC
ad e6 00 0c  //               sw   $06, 0x0C($15)
1c 20 00 03  //               bgtz $01, bgtz_f2      # this should not branch
20 07 00 d0  //               addi $07, $00, 0xD0    # pass flag5 M[D0] <-- D0
ad e7 00 10  //               sw   $07, 0x10($15)
08 10 00 23  //               j      bgtz_p2
20 0e ff fb  // bgtz_f2:   addi $14, $00, -5         # fail flag5 r14 <-- FFFF_FFFB
00 00 00 0d  //               break
1c 20 00 03  // bgtz_p2:   bgtz $01, bgtz_f3         # this should not branch
20 08 00 d4  //               addi $08, $00, 0xD4    # pass flag6 M[D0] <-- D4
ad e8 00 14  //               sw   $08, 0x14($15)
08 10 00 29  //               j      bgtz_p3
20 0e ff fa  // bgtz_f3:   addi $14, $00, -6         # fail flag6 r14 <-- FFFF_FFFA
00 00 00 0d  //               break
20 0e 00 00  // bgtz_p3:   addi $14, $00, 0          # set $r14 to 0000_0000
03 e0 00 08  //               jr   $31               # return from subroutine
```

Data Memory

```
@0              // Big Endian Format

00 04 09 11     // 0x00:03
00 00 03 E8     // 0x04:07
FF FB F6 EF     // 0x08:0B
FF FF FC 18     // 0x0C:0F
00 00 01 08     // 0x10:13
00 00 01 D1     // 0x14:17
FF FF FE F8     // 0x18:1B
FF FF FE 2F     // 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
```

13. Memory Module 13

Instruction Memory

```
@0
00 00 00 1f  // main:         setie
3c 01 12 34  //               lui  $01, 0x1234
34 21 56 78  //               ori  $01, 0x5678      # LI  R01,  0x12345678
3c 02 87 65  //               lui  $02, 0x8765
34 42 43 21  //               ori  $02, 0x4321      # LI  R02,  0x87654321
3c 03 ab cd  //               lui  $03, 0xABCD
34 63 ef 01  //               ori  $03, 0xEF01      # LI  R03,  0xABCDEF01
3c 04 01 fe  //               lui  $04, 0x01FE
34 84 dc ba  //               ori  $04, 0xDCBA      # LI  R04,  0x01FEDCBA
3c 05 5a 5a  //               lui  $05, 0x5A5A
34 a5 5a 5a  //               ori  $05, 0x5A5A      # LI  R05,  0x5A5A5A5A
3c 06 ff ff  //               lui  $06, 0xFFFF
34 c6 ff ff  //               ori  $06, 0xFFFF      # LI  R06,  0xFFFFFFFF
3c 07 ff ff  //               lui  $07, 0xFFFF
34 e7 ff 00  //               ori  $07, 0xFF00      # LI  R07,  0xFFFFFF00

00 c7 40 20  //               add  $08, $06, $07
00 c8 48 20  //               add  $09, $06, $08
00 c9 50 20  //               add  $10, $06, $09
00 ca 58 20  //               add  $11, $06, $10
00 cb 60 20  //               add  $12, $06, $11
00 cc 68 20  //               add  $13, $06, $12
00 cd 70 20  //               add  $14, $06, $13
00 ce 78 20  //               add  $15, $06, $14

3c 07 10 01  //               lui  $07, 0x1001
34 e7 03 f0  //               ori  $07, 0x03F0      # LI   R07,  0x100103F0
ac ef 00 00  //               sw   $15, 0($07)      # ST  [R07], R15
00 00 00 0d  //               break

@200
                      //************************************************
                      // In this ISR, we will implement writing
                      // some patterns to the IO space, and then
                      // reading them back.
                      // Note: this "ISR" expects the "return address"
                      //      to have been saved in $ra (not the stack)
                      //************************************************
3c 10 10 01  // isr:          lui     $16, 0x1001    #load destination IO address
36 10 00 c0  //               ori     $16, 0x00C0    #  0x100100C0 into r16
3c 11 80 00  //               lui     $17, 0x8000    #initialize the pattern of
36 31 ff ff  //               ori     $17, 0xFFFF    #  0x8000FFFF into r17
20 12 00 10  //               addi   $18, $0, 0x10   #loop counter set to 16

76 11 00 00  // out_IO:       output $17, 0($16)     # output  [R16], R17
00 11 88 83  //               sra     $17, $17, 2    # change the pattern by shifting twice
22 10 00 04  //               addi   $16, $16, 4     # increment the memory pointer 4 bytes
22 52 ff ff  //               addi   $18, $18, -1    # decrement the loop counter
16 40 ff fb  //               bne     $18, $00, out_IO #  and jmp to top if not finished

3c 10 10 01  //               lui     $16, 0x1001    #load source IO address
36 10 00 c0  //               ori     $16, 0x00C0    #  0x100100C0 into r16
72 13 00 00  //               input  $19,  0($16)    #  and input from 6
72 14 00 04  //               input  $20,  4($16)    #      the IO locations,
72 15 00 08  //               input  $21,  8($16)    #      starting from 0xC0
72 16 00 0c  //               input  $22, 12($16)
72 17 00 10  //               input  $23, 16($16)
72 18 00 14  //               input  $24, 20($16)
03 e0 00 08  //               jr      $31            # return from interrupt (v1, using $ra)
```

Data Memory

```
@0

C3 C3 C3 C3
12 34 56 78
89 AB CD EF
A5 A5 A5 A5
5A 5A 5A 5A
24 68 AC E0
13 57 9B DF
0F 0F 0F 0F
F0 F0 F0 F0
00 00 00 09
00 00 00 0A
00 00 00 0B
00 00 00 0C
00 00 00 0D
FF FF FF F8
00 00 75 CC

@1CC
AB CD EF 01

@3F8
00 00 00 00

@3FC
00 00 02 00
```

14. Memory Module 14

Instruction Memory

```
@0
00 00 00 1f  // main:      setie
3c 01 12 34  //            lui  $01, 0x1234
34 21 56 78  //            ori  $01, 0x5678      # LI  R01,  0x12345678
3c 02 87 65  //            lui  $02, 0x8765
34 42 43 21  //            ori  $02, 0x4321      # LI  R02,  0x87654321
3c 03 ab cd  //            lui  $03, 0xABCD
34 63 ef 01  //            ori  $03, 0xEF01      # LI  R03,  0xABCDEF01
3c 04 01 fe  //            lui  $04, 0x01FE
34 84 dc ba  //            ori  $04, 0xDCBA      # LI  R04,  0x01FEDCBA
3c 05 5a 5a  //            lui  $05, 0x5A5A
34 a5 5a 5a  //            ori  $05, 0x5A5A      # LI  R05,  0x5A5A5A5A
3c 06 ff ff  //            lui  $06, 0xFFFF
34 c6 ff ff  //            ori  $06, 0xFFFF      # LI  R06,  0xFFFFFFFF
3c 07 ff ff  //            lui  $07, 0xFFFF
34 e7 ff 00  //            ori  $07, 0xFF00      # LI  R07,  0xFFFFFF00

00 c7 40 20  //            add  $08, $06, $07
00 c8 48 20  //            add  $09, $06, $08
00 c9 50 20  //            add  $10, $06, $09
00 ca 58 20  //            add  $11, $06, $10
00 cb 60 20  //            add  $12, $06, $11
00 cc 68 20  //            add  $13, $06, $12
00 cd 70 20  //            add  $14, $06, $13
00 ce 78 20  //            add  $15, $06, $14

3c 07 10 01  //            lui  $07, 0x1001
34 e7 03 f0  //            ori  $07, 0x03F0      # LI   R07,  0x100103F0
ac ef 00 00  //            sw   $15, 0($07)      # ST  [R07], R15
00 00 00 0d  //            break

@200
                         //**********************************************
                         // In this ISR, we will implement writing
                         // some patterns to the IO space, and then
                         // reading them back.
                         // Note: this "ISR" expects the "return address"
                         //       to have been saved on the stack (not $ra)
                         //**********************************************
3c 10 10 01  // isr:       lui    $16, 0x1001      #load destination IO address
36 10 00 c0  //            ori    $16, 0x00C0      #  0x100100C0 into r16
3c 11 80 00  //            lui    $17, 0x8000      #initialize the pattern of
36 31 ff ff  //            ori    $17, 0xFFFF      #  0x8000FFFF into r17
20 12 00 10  //            addi   $18, $0, 0x10    #loop counter set to 16

76 11 00 00  // out_IO:    output $17, 0($16)      # output  [R16], R17
00 11 88 83  //            sra    $17, $17, 2      # change the pattern by shifting twice
22 10 00 04  //            addi   $16, $16, 4      # increment the memory pointer 4 bytes
22 52 ff ff  //            addi   $18, $18, -1     # decrement the loop counter
16 40 ff fb  //            bne    $18, $00, out_IO #  and jmp to top if not finished

3c 10 10 01  //            lui    $16, 0x1001      #load source IO address
36 10 00 c0  //            ori    $16, 0x00C0      #  0x100100C0 into r16
72 13 00 00  //            input  $19,  0($16)     #  and input from 6
72 14 00 04  //            input  $20,  4($16)     #    the IO locations,
72 15 00 08  //            input  $21,  8($16)     #    starting from 0xC0
72 16 00 0c  //            input  $22, 12($16)
72 17 00 10  //            input  $23, 16($16)
72 18 00 14  //            input  $24, 20($16)
7B A0 00 00  //            reti                    # return from interrupt (v2, using M[sp] as saved
PC)
                                                   # Note opcode=0x1E and $rs=0x1D, i.e. $sp
```

Data Memory

```
@0

C3 C3 C3 C3
12 34 56 78
89 AB CD EF
A5 A5 A5 A5
5A 5A 5A 5A
24 68 AC E0
13 57 9B DF
0F 0F 0F 0F
F0 F0 F0 F0
00 00 00 09
00 00 00 0A
00 00 00 0B
00 00 00 0C
00 00 00 0D
FF FF FF F8
00 00 75 CC

@1CC
AB CD EF 01

@3F8
00 00 00 00

@3FC
00 00 02 00
```

15. Memory Module 15 Vector

Instruction Memory

```
@0
3c 01 f5 f6  //              lui  $01, 0xf5f6
34 21 f7 f8  //              ori  $01, 0xf7f8     # LI  R01,  0xf5f6f7f8
3c 02 10 51  //              lui  $02, 0x1051
34 42 10 e1  //              ori  $02, 0x10e1     # LI  R02,  0x105110e1

//Add and Multiple
b0 22 18 20  //              add8 $03, $01, $02   # R03 <-- 0x05_47_07_d9
b4 22 20 20  //              add16 $04, $01, $02  # R04 <-- 0x06_48_08_d9
b0 22 00 18  //              mult8  $01, $02
00 00 28 12  //              mflo $05             # R05 <-- 0x0f70_d9f8
00 00 30 10  //              mfhi $06             # R06 <-- 0x0f50_4dd6
b4 22 00 18  //              mult16 $01, $02
00 00 38 12  //              mflo $07             # R06 <-- 0x105970f8
00 00 40 10  //              mfhi $08             # R07 <-- 0x0fad32d6

3c 0e e9 56  //              lui  $14, 0xe956
35 ce e0 03  //              ori  $14, 0xe003     # LI  R14,  0xe956e003
3c 0f 4b b4  //              lui  $15, 0x4bb4
35 ef 01 1e  //              ori  $15, 0x011e     # LI  R15,  0x4bb4011e
//Subtract and Divide
b1 cf 98 22  //              sub8, $19, $14, $15  # R19 <-- 0x9e_a2_df_e5
b5 cf a0 22  //              sub16, $20, $14, $15 # R20 <-- 0x9da2_dee5
b1 cf 00 1a  //              div, $14, $15
00 00 a8 12  //              mflo $21             # R21 <-- 0x00_e0_00_5a
00 cf b0 10  //              mfhi $22             # R22 <-- 0x44_43_3c_78
b5 cf 00 1a  //              div16, $14, $15
00 00 b8 12  //              mflo $23             # R23 <-- 0x00fa_435a
00 00 c0 10  //              mfhi $24             # R24 <-- 0x4500_4278
00 00 00 0d  //              break                # end
```

16. Memory Module 15 Vector_2

Instruction Memory

```
@0
3c 01 af af  //               lui  $01, 0xafaf
34 21 af af  //               ori  $01, 0xafaf    # LI  R01,  0xafafafaf
3c 02 a5 5a  //               lui  $02, 0xa55a
34 42 80 0f  //               ori  $02, 0x0107    # LI  R02,  0x055a0107

//Shift Instructions
b0 01 48 40  //               sll8 $09, $01, 1    # R09 <-- 0x5e_5e_5e_5e
b4 01 50 40  //               sll16 $10,$01, 1    # R10 <-- 0x5f5e_5f5e
b0 01 58 42  //               srl8 $11, $01, 1    # R11 <-- 0x57_57_57_57
b4 01 60 42  //               srl16 $12, $02, 1   # R12 <-- 0x57d7_57d7
b0 02 68 83  //               sra8 $13, $02, 2    # R13 <-- 0xe9_16_e0_03
b4 02 70 83  //               sra16 $14, $02, 2   # R14 <-- 0xe956_e003
b0 02 78 44  //               rotl8 $15, $02, 1   # R15 <-- 0x4b_b4_01_1e
b4 02 80 44  //               rotl16 $16,$02, 1   # R16 <-- 0x4ab5_001f
b0 02 88 45  //               rotr8 $17, $02, 1   # R17 <-- 0xd2_2d_40_87
b4 02 90 45  //               rotr16 $18,$02, 1   # R18 <-- 0x52ad_c007
00 00 00 0d  //               break               # end
```

17. Memory Module 15 Vector_3

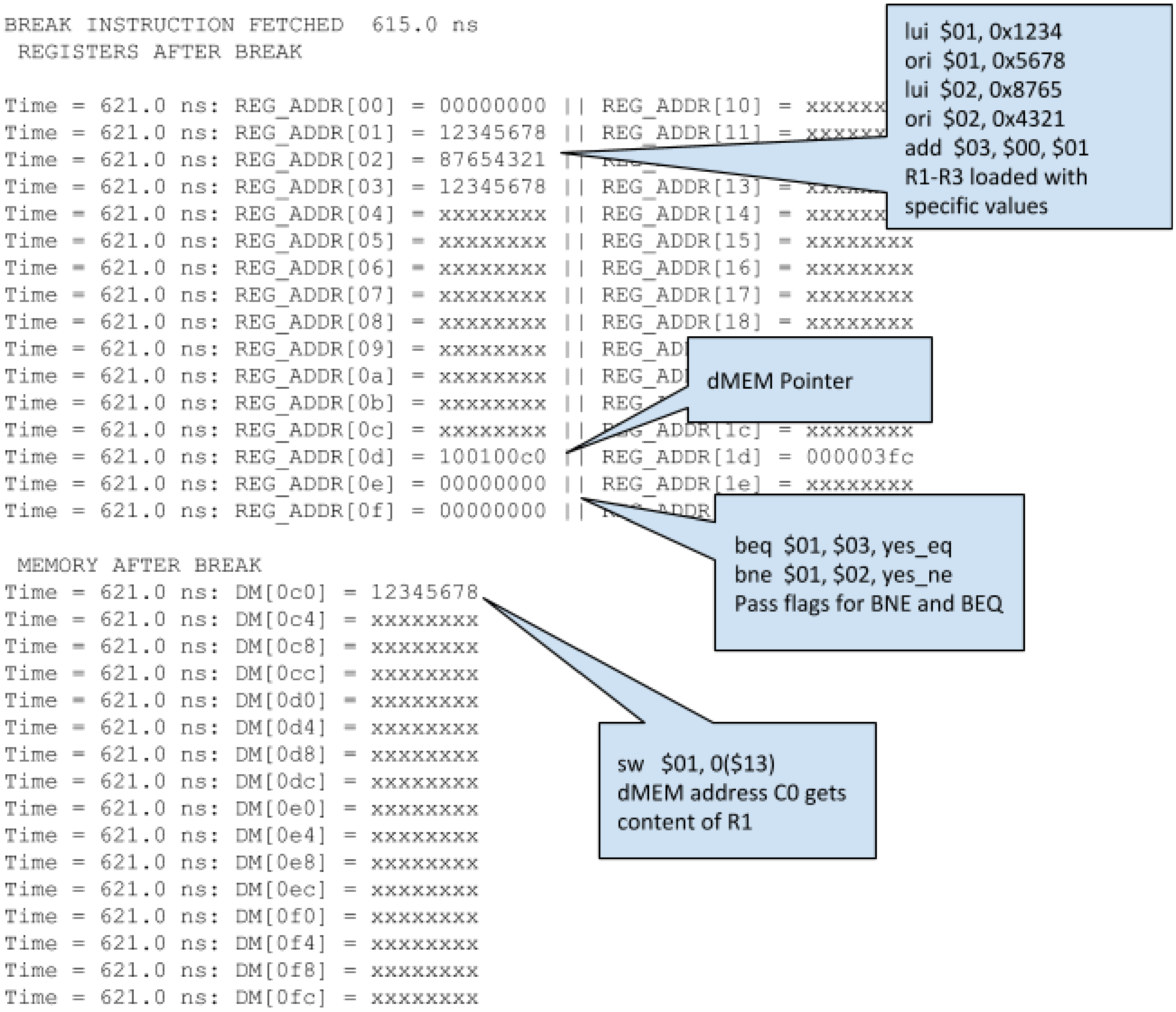
Instruction Memory

```
@0
3c 01 af af  //              lui  $01, 0xafaf
34 21 af af  //              ori  $01, 0xafaf     # LI  R01,  0xafafafaf
3c 02 a5 5a  //              lui  $02, 0xa55a
34 42 80 0f  //              ori  $02, 0x0107     # LI  R02,  0x055a0107

//Logical operators
b0 22 18 24  //              and8 $03, $01, $02   # R03 <-- A5_0A_80_0F
b4 22 20 24  //              and16 $04, $01, $02  # R04 <-- A50A_800F
b0 22 28 25  //              or8 $05, $01, $02    # R05 <-- AF_FF_AF_AF
b4 22 30 25  //              or16 $06, $01, $02   # R06 <-- AFFF_AFAF
b0 22 38 26  //              xor8 $07, $01, $02   # R07 <-- 0A_F5_2F_A0
b4 22 40 26  //              xor16 $08, $01, $02  # R08 <-- 0AF5_2FA0
b0 22 48 27  //              nor8 $09, $01, $02   # R09 <-- 50_00_50_50
b4 22 50 27  //              nor16 $10, $01, $02  # R10 <-- 5000_5050
00 00 00 0d  //              break                # end
```

D. Annotated Memory Modules

1. Memory Module 01 Results



```
BREAK INSTRUCTION FETCHED  615.0 ns
 REGISTERS AFTER BREAK

Time = 621.0 ns: REG_ADDR[00] = 00000000 || REG_ADDR[10] = xxxxxx
Time = 621.0 ns: REG_ADDR[01] = 12345678 || REG_ADDR[11] = xxxxxx
Time = 621.0 ns: REG_ADDR[02] = 87654321 || REG
Time = 621.0 ns: REG_ADDR[03] = 12345678 || REG_ADDR[13] = xxxx
Time = 621.0 ns: REG_ADDR[04] = xxxxxxxx || REG_ADDR[14] = xxxxxx
Time = 621.0 ns: REG_ADDR[05] = xxxxxxxx || REG_ADDR[15] = xxxxxxxx
Time = 621.0 ns: REG_ADDR[06] = xxxxxxxx || REG_ADDR[16] = xxxxxxxx
Time = 621.0 ns: REG_ADDR[07] = xxxxxxxx || REG_ADDR[17] = xxxxxxxx
Time = 621.0 ns: REG_ADDR[08] = xxxxxxxx || REG_ADDR[18] = xxxxxxxx
Time = 621.0 ns: REG_ADDR[09] = xxxxxxxx || REG_AD
Time = 621.0 ns: REG_ADDR[0a] = xxxxxxxx || REG_AD
Time = 621.0 ns: REG_ADDR[0b] = xxxxxxxx || REG
Time = 621.0 ns: REG_ADDR[0c] = xxxxxxxx || REG_ADDR[1c] = xxxxxxxx
Time = 621.0 ns: REG_ADDR[0d] = 100100c0 || REG_ADDR[1d] = 000003fc
Time = 621.0 ns: REG_ADDR[0e] = 00000000 || REG_ADDR[1e] = xxxxxxxx
Time = 621.0 ns: REG_ADDR[0f] = 00000000 || REG_ADDR

 MEMORY AFTER BREAK
Time = 621.0 ns: DM[0c0] = 12345678
Time = 621.0 ns: DM[0c4] = xxxxxxxx
Time = 621.0 ns: DM[0c8] = xxxxxxxx
Time = 621.0 ns: DM[0cc] = xxxxxxxx
Time = 621.0 ns: DM[0d0] = xxxxxxxx
Time = 621.0 ns: DM[0d4] = xxxxxxxx
Time = 621.0 ns: DM[0d8] = xxxxxxxx
Time = 621.0 ns: DM[0dc] = xxxxxxxx
Time = 621.0 ns: DM[0e0] = xxxxxxxx
Time = 621.0 ns: DM[0e4] = xxxxxxxx
Time = 621.0 ns: DM[0e8] = xxxxxxxx
Time = 621.0 ns: DM[0ec] = xxxxxxxx
Time = 621.0 ns: DM[0f0] = xxxxxxxx
Time = 621.0 ns: DM[0f4] = xxxxxxxx
Time = 621.0 ns: DM[0f8] = xxxxxxxx
Time = 621.0 ns: DM[0fc] = xxxxxxxx
```

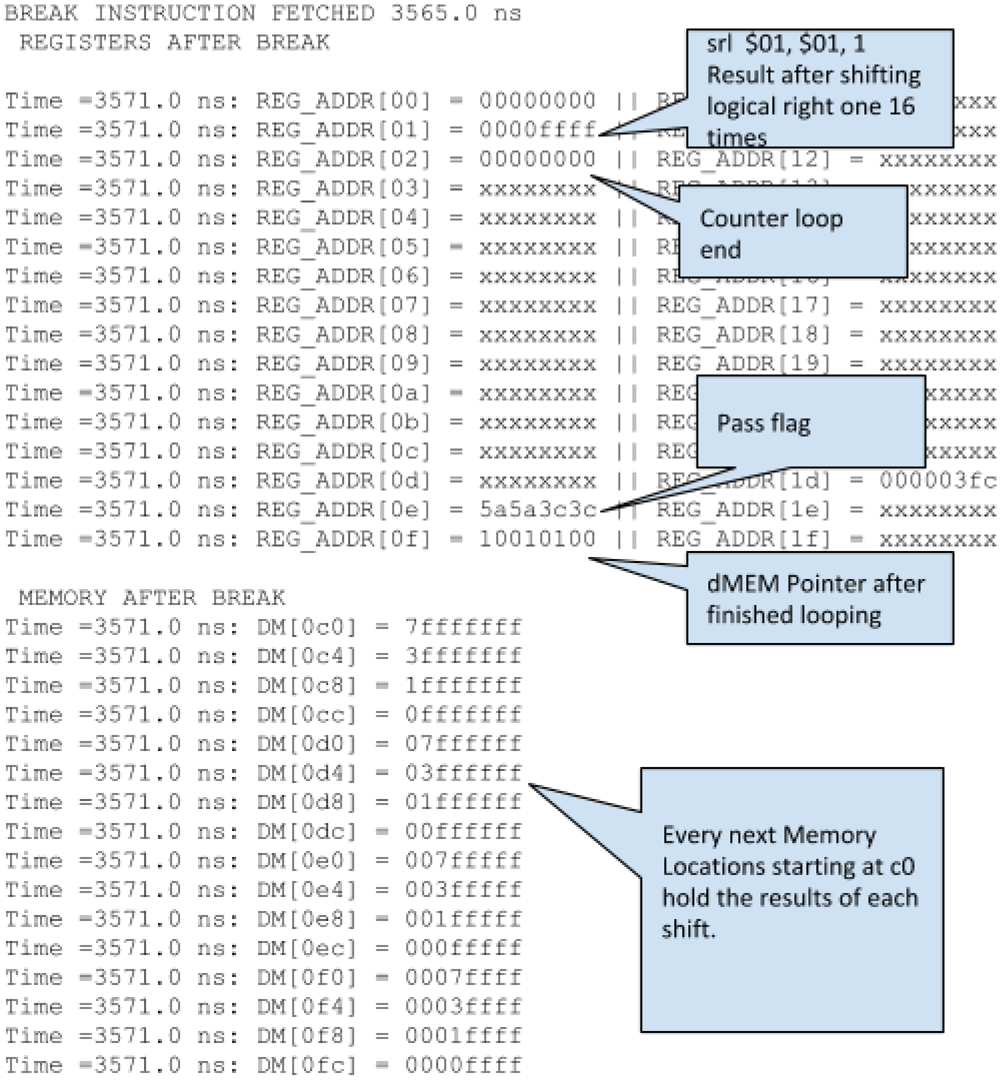
lui $01, 0x1234
ori $01, 0x5678
lui $02, 0x8765
ori $02, 0x4321
add $03, $00, $01
R1-R3 loaded with specific values

dMEM Pointer

beq $01, $03, yes_eq
bne $01, $02, yes_ne
Pass flags for BNE and BEQ

sw $01, 0($13)
dMEM address C0 gets content of R1

2. Memory Module 02 Results



```
BREAK INSTRUCTION FETCHED 3565.0 ns
 REGISTERS AFTER BREAK

Time =3571.0 ns: REG_ADDR[00] = 00000000 || RE                          xxx
Time =3571.0 ns: REG_ADDR[01] = 0000ffff                                xxx
Time =3571.0 ns: REG_ADDR[02] = 00000000 || REG_ADDR[12] = xxxxxxxx
Time =3571.0 ns: REG_ADDR[03] = xxxxxxxx    RE                      xxxxxx
Time =3571.0 ns: REG_ADDR[04] = xxxxxxxx ||                         xxxxxx
Time =3571.0 ns: REG_ADDR[05] = xxxxxxxx || RE                      xxxxxx
Time =3571.0 ns: REG_ADDR[06] = xxxxxxxx || RE                    xxxxxxxx
Time =3571.0 ns: REG_ADDR[07] = xxxxxxxx || REG_ADDR[17] = xxxxxxxx
Time =3571.0 ns: REG_ADDR[08] = xxxxxxxx || REG_ADDR[18] = xxxxxxxx
Time =3571.0 ns: REG_ADDR[09] = xxxxxxxx || REG_ADDR[19] = xxxxxxxx
Time =3571.0 ns: REG_ADDR[0a] = xxxxxxxx || REG                     xxxxx
Time =3571.0 ns: REG_ADDR[0b] = xxxxxxxx || REG                     xxxxx
Time =3571.0 ns: REG_ADDR[0c] = xxxxxxxx || REG                     xxxxx
Time =3571.0 ns: REG_ADDR[0d] = xxxxxxxx || RE   DDR[1d] = 000003fc
Time =3571.0 ns: REG_ADDR[0e] = 5a5a3c3c    REG_ADDR[1e] = xxxxxxxx
Time =3571.0 ns: REG_ADDR[0f] = 10010100 || REG_ADDR[1f] = xxxxxxxx

 MEMORY AFTER BREAK
Time =3571.0 ns: DM[0c0] = 7fffffff
Time =3571.0 ns: DM[0c4] = 3fffffff
Time =3571.0 ns: DM[0c8] = 1fffffff
Time =3571.0 ns: DM[0cc] = 0fffffff
Time =3571.0 ns: DM[0d0] = 07ffffff
Time =3571.0 ns: DM[0d4] = 03ffffff
Time =3571.0 ns: DM[0d8] = 01ffffff
Time =3571.0 ns: DM[0dc] = 00ffffff
Time =3571.0 ns: DM[0e0] = 007fffff
Time =3571.0 ns: DM[0e4] = 003fffff
Time =3571.0 ns: DM[0e8] = 001fffff
Time =3571.0 ns: DM[0ec] = 000fffff
Time =3571.0 ns: DM[0f0] = 0007ffff
Time =3571.0 ns: DM[0f4] = 0003ffff
Time =3571.0 ns: DM[0f8] = 0001ffff
Time =3571.0 ns: DM[0fc] = 0000ffff
```

srl $01, $01, 1
Result after shifting logical right one 16 times

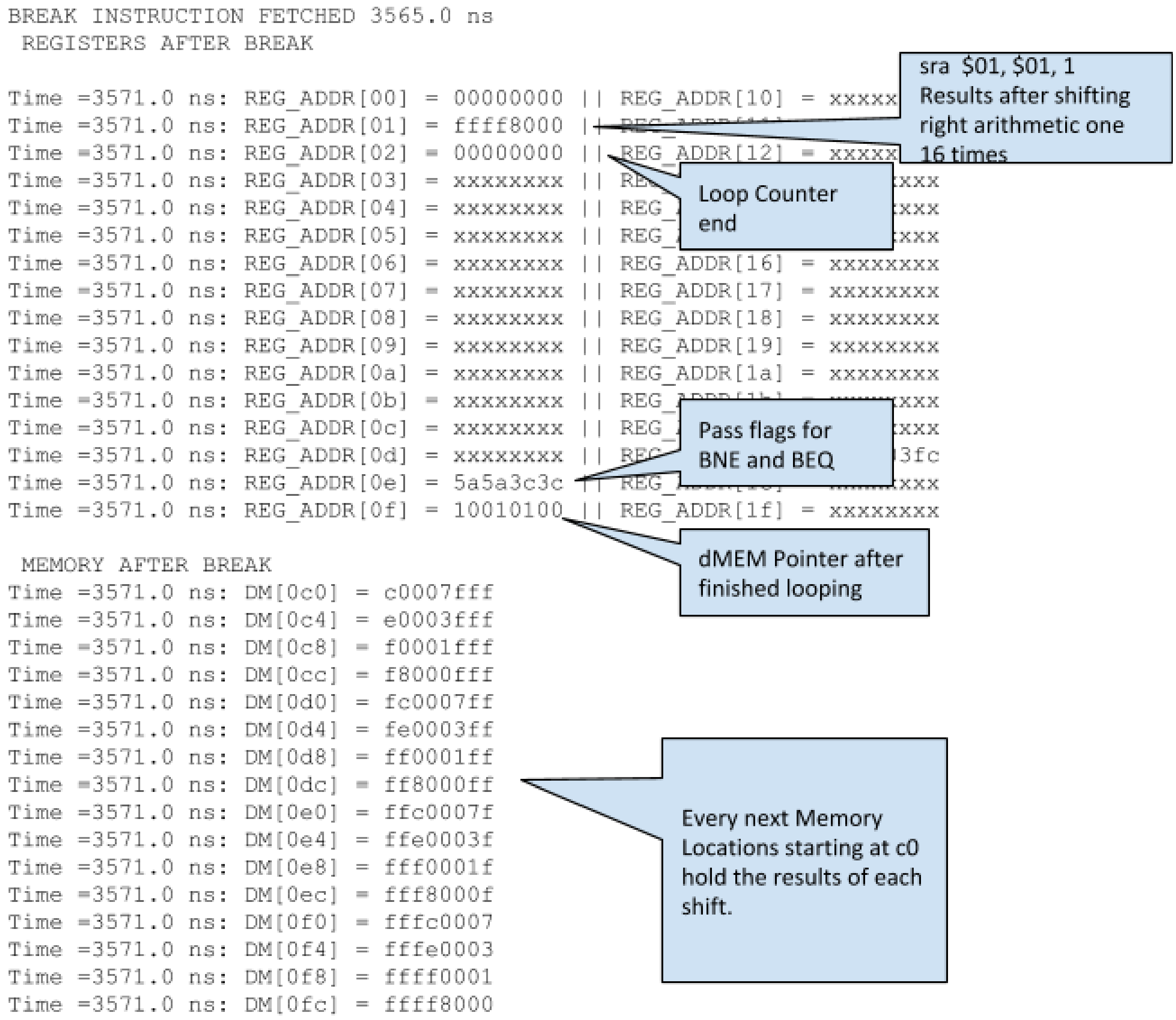
Counter loop end

Pass flag

dMEM Pointer after finished looping

Every next Memory Locations starting at c0 hold the results of each shift.

3. Memory Module 03 Results



```
BREAK INSTRUCTION FETCHED 3565.0 ns
 REGISTERS AFTER BREAK

Time =3571.0 ns: REG_ADDR[00] = 00000000 || REG_ADDR[10] = xxxxx
Time =3571.0 ns: REG_ADDR[01] = ffff8000 |
Time =3571.0 ns: REG_ADDR[02] = 00000000 || REG_ADDR[12] = xxxxx
Time =3571.0 ns: REG_ADDR[03] = xxxxxxxx || RE                xxx
Time =3571.0 ns: REG_ADDR[04] = xxxxxxxx || REG               xxx
Time =3571.0 ns: REG_ADDR[05] = xxxxxxxx || REG               xxx
Time =3571.0 ns: REG_ADDR[06] = xxxxxxxx || REG_ADDR[16] = xxxxxxxx
Time =3571.0 ns: REG_ADDR[07] = xxxxxxxx || REG_ADDR[17] = xxxxxxxx
Time =3571.0 ns: REG_ADDR[08] = xxxxxxxx || REG_ADDR[18] = xxxxxxxx
Time =3571.0 ns: REG_ADDR[09] = xxxxxxxx || REG_ADDR[19] = xxxxxxxx
Time =3571.0 ns: REG_ADDR[0a] = xxxxxxxx || REG_ADDR[1a] = xxxxxxxx
Time =3571.0 ns: REG_ADDR[0b] = xxxxxxxx || REG               xxx
Time =3571.0 ns: REG_ADDR[0c] = xxxxxxxx || REG               xxx
Time =3571.0 ns: REG_ADDR[0d] = xxxxxxxx || REG               3fc
Time =3571.0 ns: REG_ADDR[0e] = 5a5a3c3c || REG               xxx
Time =3571.0 ns: REG_ADDR[0f] = 10010100 || REG_ADDR[1f] = xxxxxxxx

 MEMORY AFTER BREAK
Time =3571.0 ns: DM[0c0] = c0007fff
Time =3571.0 ns: DM[0c4] = e0003fff
Time =3571.0 ns: DM[0c8] = f0001fff
Time =3571.0 ns: DM[0cc] = f8000fff
Time =3571.0 ns: DM[0d0] = fc0007ff
Time =3571.0 ns: DM[0d4] = fe0003ff
Time =3571.0 ns: DM[0d8] = ff0001ff
Time =3571.0 ns: DM[0dc] = ff8000ff
Time =3571.0 ns: DM[0e0] = ffc0007f
Time =3571.0 ns: DM[0e4] = ffe0003f
Time =3571.0 ns: DM[0e8] = fff0001f
Time =3571.0 ns: DM[0ec] = fff8000f
Time =3571.0 ns: DM[0f0] = fffc0007
Time =3571.0 ns: DM[0f4] = fffe0003
Time =3571.0 ns: DM[0f8] = ffff0001
Time =3571.0 ns: DM[0fc] = ffff8000
```

sra $01, $01, 1
Results after shifting right arithmetic one 16 times

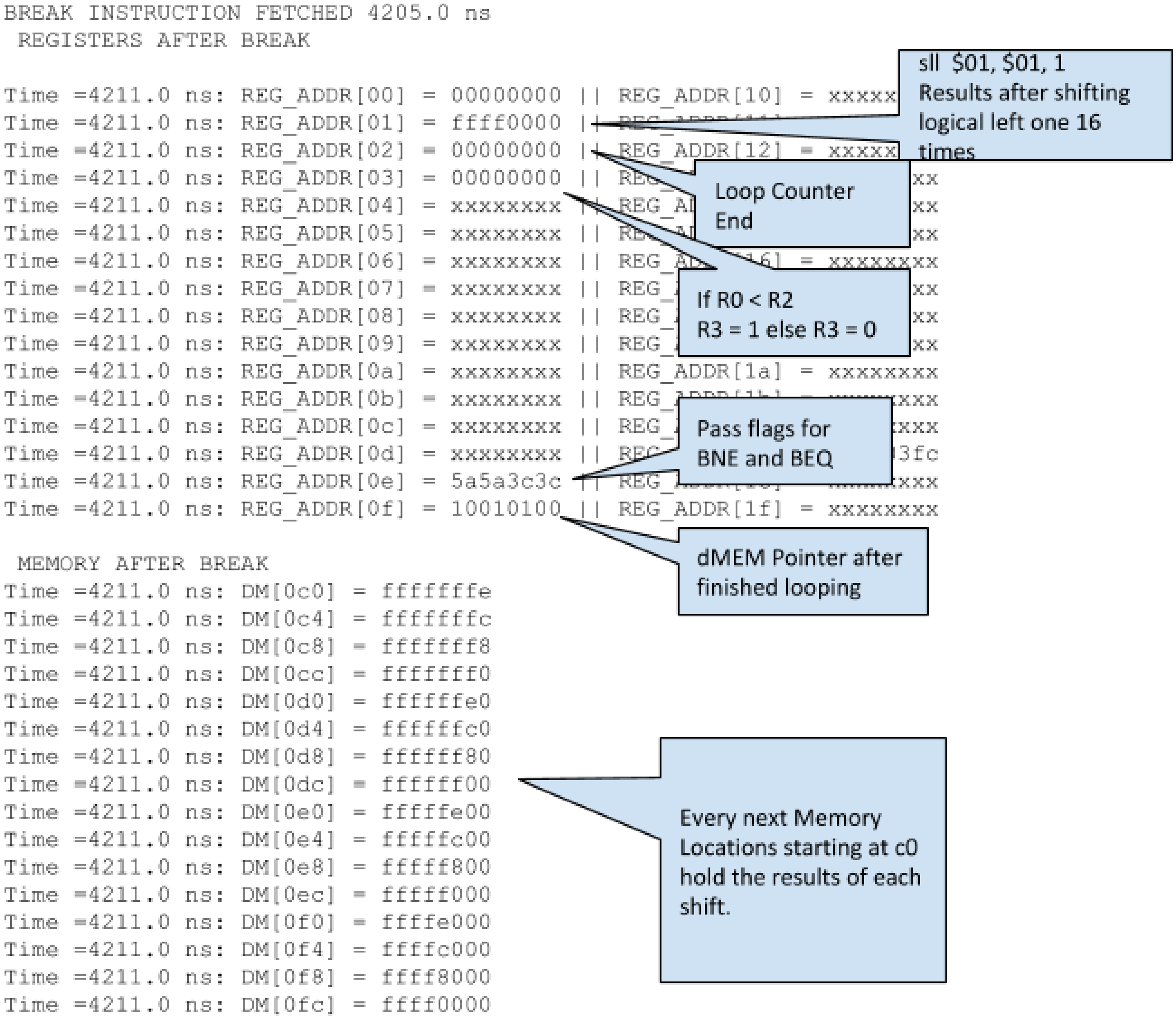
Loop Counter end

Pass flags for BNE and BEQ

dMEM Pointer after finished looping

Every next Memory Locations starting at c0 hold the results of each shift.

4. Memory Module 04 Results



```
BREAK INSTRUCTION FETCHED 4205.0 ns
 REGISTERS AFTER BREAK

Time =4211.0 ns: REG_ADDR[00] = 00000000 || REG_ADDR[10] = xxxxx
Time =4211.0 ns: REG_ADDR[01] = ffff0000 || REG_ADDR[11]
Time =4211.0 ns: REG_ADDR[02] = 00000000 || REG_ADDR[12] = xxxxx
Time =4211.0 ns: REG_ADDR[03] = 00000000 || REG                   xx
Time =4211.0 ns: REG_ADDR[04] = xxxxxxxx || REG_AD                xx
Time =4211.0 ns: REG_ADDR[05] = xxxxxxxx || REG_AD                xx
Time =4211.0 ns: REG_ADDR[06] = xxxxxxxx || REG_AD  16] = xxxxxxxx
Time =4211.0 ns: REG_ADDR[07] = xxxxxxxx || REG_                  xx
Time =4211.0 ns: REG_ADDR[08] = xxxxxxxx || REG_                  xx
Time =4211.0 ns: REG_ADDR[09] = xxxxxxxx || REG_                  xx
Time =4211.0 ns: REG_ADDR[0a] = xxxxxxxx || REG_ADDR[1a] = xxxxxxxx
Time =4211.0 ns: REG_ADDR[0b] = xxxxxxxx || REG_ADDR[1b] = xxxxxxxx
Time =4211.0 ns: REG_ADDR[0c] = xxxxxxxx || REG_                xxxx
Time =4211.0 ns: REG_ADDR[0d] = xxxxxxxx || REG                  3fc
Time =4211.0 ns: REG_ADDR[0e] = 5a5a3c3c || REG                 xxxx
Time =4211.0 ns: REG_ADDR[0f] = 10010100 || REG_ADDR[1f] = xxxxxxxx

 MEMORY AFTER BREAK
Time =4211.0 ns: DM[0c0] = fffffffe
Time =4211.0 ns: DM[0c4] = fffffffc
Time =4211.0 ns: DM[0c8] = fffffff8
Time =4211.0 ns: DM[0cc] = fffffff0
Time =4211.0 ns: DM[0d0] = ffffffe0
Time =4211.0 ns: DM[0d4] = ffffffc0
Time =4211.0 ns: DM[0d8] = ffffff80
Time =4211.0 ns: DM[0dc] = ffffff00
Time =4211.0 ns: DM[0e0] = fffffe00
Time =4211.0 ns: DM[0e4] = fffffc00
Time =4211.0 ns: DM[0e8] = fffff800
Time =4211.0 ns: DM[0ec] = fffff000
Time =4211.0 ns: DM[0f0] = ffffe000
Time =4211.0 ns: DM[0f4] = ffffc000
Time =4211.0 ns: DM[0f8] = ffff8000
Time =4211.0 ns: DM[0fc] = ffff0000
```

sll $01, $01, 1
Results after shifting logical left one 16 times

Loop Counter End

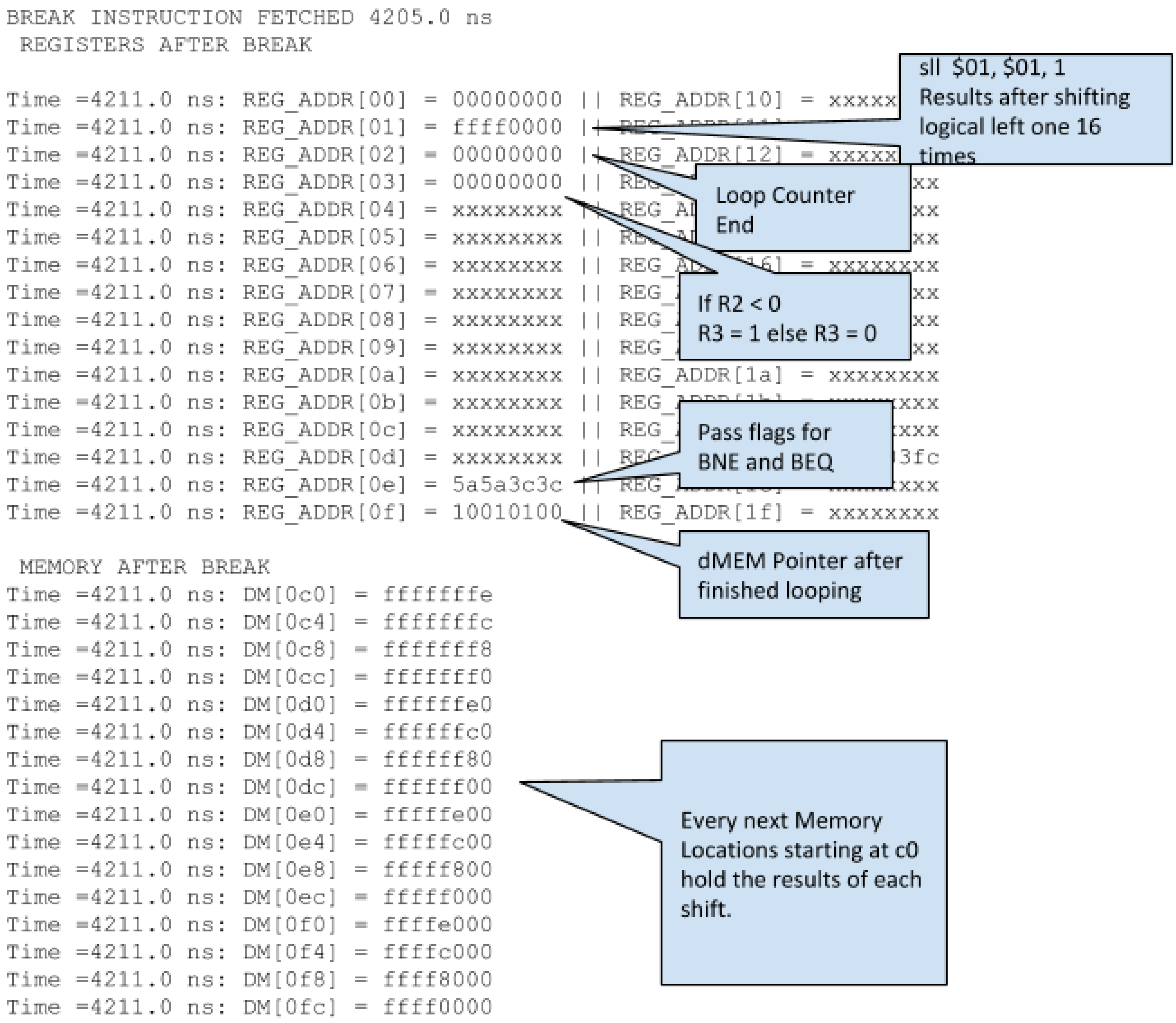
If R0 < R2
R3 = 1 else R3 = 0

Pass flags for BNE and BEQ

dMEM Pointer after finished looping

Every next Memory Locations starting at c0 hold the results of each shift.

5. Memory Module 05 Results



```
BREAK INSTRUCTION FETCHED 4205.0 ns
 REGISTERS AFTER BREAK

Time =4211.0 ns: REG_ADDR[00] = 00000000 || REG_ADDR[10] = xxxxx
Time =4211.0 ns: REG_ADDR[01] = ffff0000 || REG_ADDR[11]
Time =4211.0 ns: REG_ADDR[02] = 00000000 || REG_ADDR[12] = xxxxx
Time =4211.0 ns: REG_ADDR[03] = 00000000 || REG                xx
Time =4211.0 ns: REG_ADDR[04] = xxxxxxxx || REG_AD             xx
Time =4211.0 ns: REG_ADDR[05] = xxxxxxxx || REG_AD             xx
Time =4211.0 ns: REG_ADDR[06] = xxxxxxxx || REG_ADDR[16] = xxxxxxxx
Time =4211.0 ns: REG_ADDR[07] = xxxxxxxx || REG_               xx
Time =4211.0 ns: REG_ADDR[08] = xxxxxxxx || REG_               xx
Time =4211.0 ns: REG_ADDR[09] = xxxxxxxx || REG_               xx
Time =4211.0 ns: REG_ADDR[0a] = xxxxxxxx || REG_ADDR[1a] = xxxxxxxx
Time =4211.0 ns: REG_ADDR[0b] = xxxxxxxx || REG_ADDR[1b] = xxxxxxxx
Time =4211.0 ns: REG_ADDR[0c] = xxxxxxxx || REG_             xxxx
Time =4211.0 ns: REG_ADDR[0d] = xxxxxxxx || REG               3fc
Time =4211.0 ns: REG_ADDR[0e] = 5a5a3c3c || REG_ADDR[1e] = xxxxxxxx
Time =4211.0 ns: REG_ADDR[0f] = 10010100 || REG_ADDR[1f] = xxxxxxxx

 MEMORY AFTER BREAK
Time =4211.0 ns: DM[0c0] = fffffffe
Time =4211.0 ns: DM[0c4] = fffffffc
Time =4211.0 ns: DM[0c8] = fffffff8
Time =4211.0 ns: DM[0cc] = fffffff0
Time =4211.0 ns: DM[0d0] = ffffffe0
Time =4211.0 ns: DM[0d4] = ffffffc0
Time =4211.0 ns: DM[0d8] = ffffff80
Time =4211.0 ns: DM[0dc] = ffffff00
Time =4211.0 ns: DM[0e0] = fffffe00
Time =4211.0 ns: DM[0e4] = fffffc00
Time =4211.0 ns: DM[0e8] = fffff800
Time =4211.0 ns: DM[0ec] = fffff000
Time =4211.0 ns: DM[0f0] = ffffe000
Time =4211.0 ns: DM[0f4] = ffffc000
Time =4211.0 ns: DM[0f8] = ffff8000
Time =4211.0 ns: DM[0fc] = ffff0000
```

sll $01, $01, 1
Results after shifting logical left one 16 times

Loop Counter End

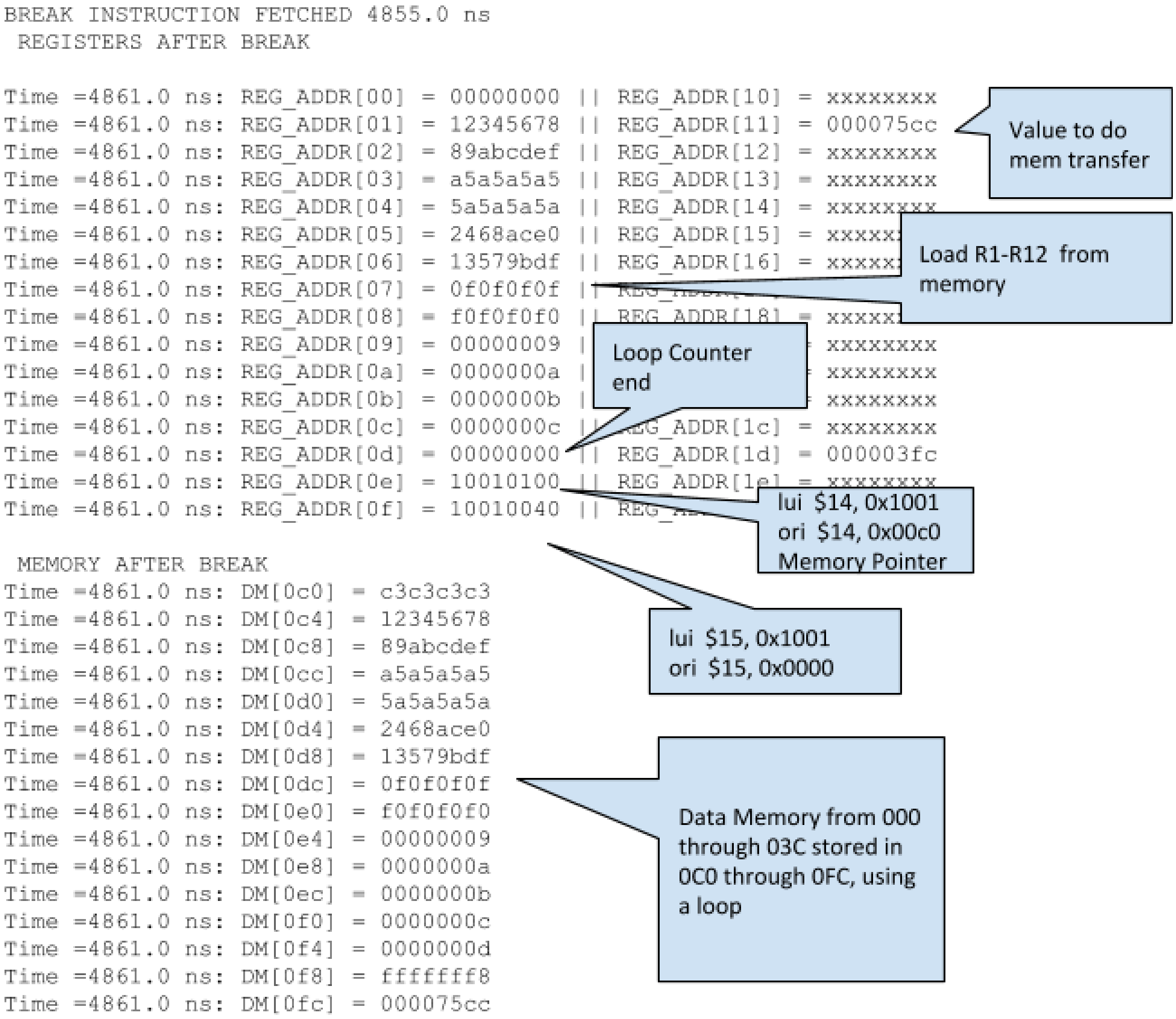
If R2 < 0
R3 = 1 else R3 = 0

Pass flags for BNE and BEQ

dMEM Pointer after finished looping

Every next Memory Locations starting at c0 hold the results of each shift.

6. Memory Module 06 Results



```
BREAK INSTRUCTION FETCHED 4855.0 ns
 REGISTERS AFTER BREAK

Time =4861.0 ns: REG_ADDR[00] = 00000000 || REG_ADDR[10] = xxxxxxxx
Time =4861.0 ns: REG_ADDR[01] = 12345678 || REG_ADDR[11] = 000075cc
Time =4861.0 ns: REG_ADDR[02] = 89abcdef || REG_ADDR[12] = xxxxxxxx
Time =4861.0 ns: REG_ADDR[03] = a5a5a5a5 || REG_ADDR[13] = xxxxxxxx
Time =4861.0 ns: REG_ADDR[04] = 5a5a5a5a || REG_ADDR[14] = xxxxxxxx
Time =4861.0 ns: REG_ADDR[05] = 2468ace0 || REG_ADDR[15] = xxxxx
Time =4861.0 ns: REG_ADDR[06] = 13579bdf || REG_ADDR[16] = xxxxx
Time =4861.0 ns: REG_ADDR[07] = 0f0f0f0f ||
Time =4861.0 ns: REG_ADDR[08] = f0f0f0f0 || REG_ADDR[18] = xxxxx
Time =4861.0 ns: REG_ADDR[09] = 00000009 |                 = xxxxxxxx
Time =4861.0 ns: REG_ADDR[0a] = 0000000a |                 = xxxxxxxx
Time =4861.0 ns: REG_ADDR[0b] = 0000000b |                 = xxxxxxxx
Time =4861.0 ns: REG_ADDR[0c] = 0000000c || REG_ADDR[1c] = xxxxxxxx
Time =4861.0 ns: REG_ADDR[0d] = 00000000 || REG_ADDR[1d] = 000003fc
Time =4861.0 ns: REG_ADDR[0e] = 10010100 || REG_ADDR[1e] = xxxxxxxx
Time =4861.0 ns: REG_ADDR[0f] = 10010040 || REG_AD

 MEMORY AFTER BREAK
Time =4861.0 ns: DM[0c0] = c3c3c3c3
Time =4861.0 ns: DM[0c4] = 12345678
Time =4861.0 ns: DM[0c8] = 89abcdef
Time =4861.0 ns: DM[0cc] = a5a5a5a5
Time =4861.0 ns: DM[0d0] = 5a5a5a5a
Time =4861.0 ns: DM[0d4] = 2468ace0
Time =4861.0 ns: DM[0d8] = 13579bdf
Time =4861.0 ns: DM[0dc] = 0f0f0f0f
Time =4861.0 ns: DM[0e0] = f0f0f0f0
Time =4861.0 ns: DM[0e4] = 00000009
Time =4861.0 ns: DM[0e8] = 0000000a
Time =4861.0 ns: DM[0ec] = 0000000b
Time =4861.0 ns: DM[0f0] = 0000000c
Time =4861.0 ns: DM[0f4] = 0000000d
Time =4861.0 ns: DM[0f8] = fffffff8
Time =4861.0 ns: DM[0fc] = 000075cc
```

Value to do mem transfer

Load R1-R12 from memory

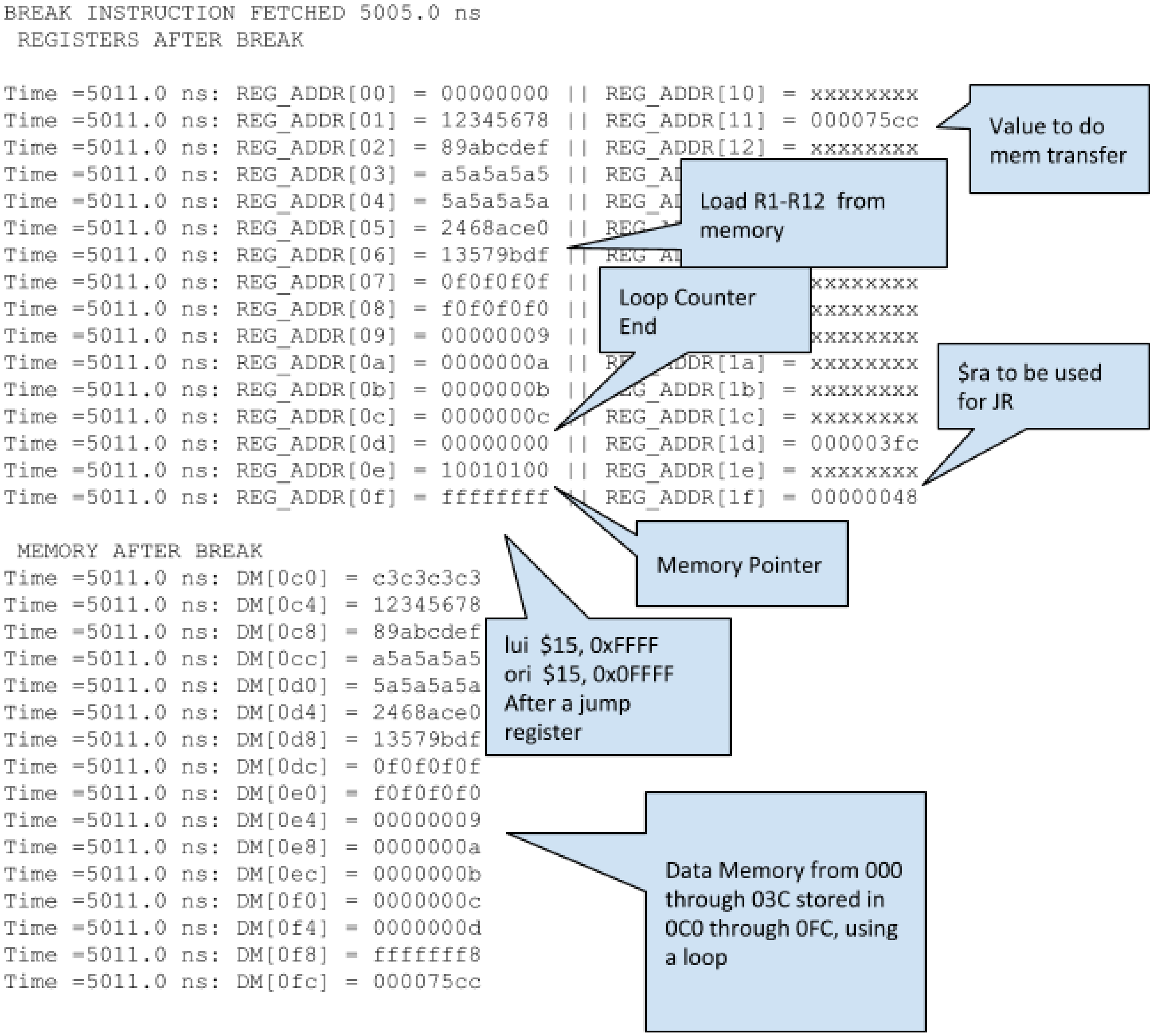
Loop Counter end

lui $14, 0x1001
ori $14, 0x00c0
Memory Pointer

lui $15, 0x1001
ori $15, 0x0000

Data Memory from 000 through 03C stored in 0C0 through 0FC, using a loop

7. Memory Module 07 Results



```
BREAK INSTRUCTION FETCHED 5005.0 ns
 REGISTERS AFTER BREAK

Time =5011.0 ns: REG_ADDR[00] = 00000000 || REG_ADDR[10] = xxxxxxxx
Time =5011.0 ns: REG_ADDR[01] = 12345678 || REG_ADDR[11] = 000075cc
Time =5011.0 ns: REG_ADDR[02] = 89abcdef || REG_ADDR[12] = xxxxxxxx
Time =5011.0 ns: REG_ADDR[03] = a5a5a5a5 || REG_AD
Time =5011.0 ns: REG_ADDR[04] = 5a5a5a5a || REG_AD
Time =5011.0 ns: REG_ADDR[05] = 2468ace0 || REG_
Time =5011.0 ns: REG_ADDR[06] = 13579bdf || REG_AD
Time =5011.0 ns: REG_ADDR[07] = 0f0f0f0f ||                  xxxxxxxx
Time =5011.0 ns: REG_ADDR[08] = f0f0f0f0 ||                  xxxxxxxx
Time =5011.0 ns: REG_ADDR[09] = 00000009 ||                  xxxxxxxx
Time =5011.0 ns: REG_ADDR[0a] = 0000000a || R    DDR[1a] = xxxxxxxx
Time =5011.0 ns: REG_ADDR[0b] = 0000000b ||   G_ADDR[1b] = xxxxxxxx
Time =5011.0 ns: REG_ADDR[0c] = 0000000c    REG_ADDR[1c] = xxxxxxxx
Time =5011.0 ns: REG_ADDR[0d] = 00000000 || REG_ADDR[1d] = 000003fc
Time =5011.0 ns: REG_ADDR[0e] = 10010100 || REG_ADDR[1e] = xxxxxxxx
Time =5011.0 ns: REG_ADDR[0f] = ffffffff    REG_ADDR[1f] = 00000048

 MEMORY AFTER BREAK
Time =5011.0 ns: DM[0c0] = c3c3c3c3
Time =5011.0 ns: DM[0c4] = 12345678
Time =5011.0 ns: DM[0c8] = 89abcdef
Time =5011.0 ns: DM[0cc] = a5a5a5a5
Time =5011.0 ns: DM[0d0] = 5a5a5a5a
Time =5011.0 ns: DM[0d4] = 2468ace0
Time =5011.0 ns: DM[0d8] = 13579bdf
Time =5011.0 ns: DM[0dc] = 0f0f0f0f
Time =5011.0 ns: DM[0e0] = f0f0f0f0
Time =5011.0 ns: DM[0e4] = 00000009
Time =5011.0 ns: DM[0e8] = 0000000a
Time =5011.0 ns: DM[0ec] = 0000000b
Time =5011.0 ns: DM[0f0] = 0000000c
Time =5011.0 ns: DM[0f4] = 0000000d
Time =5011.0 ns: DM[0f8] = fffffff8
Time =5011.0 ns: DM[0fc] = 000075cc
```

Value to do mem transfer

Load R1-R12 from memory

Loop Counter End

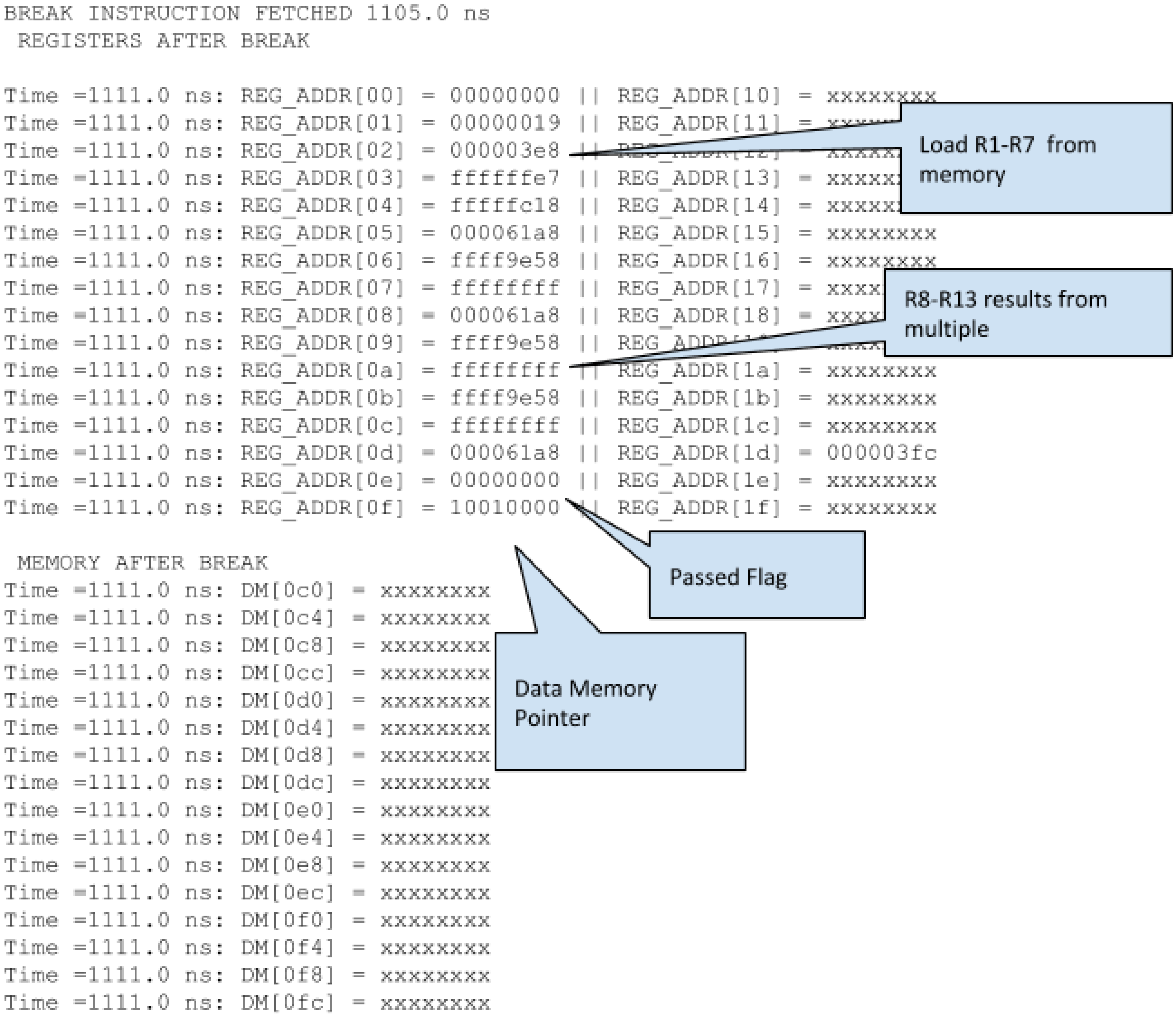
$ra to be used for JR

Memory Pointer

lui $15, 0xFFFF
ori $15, 0x0FFFF
After a jump register

Data Memory from 000 through 03C stored in 0C0 through 0FC, using a loop

8. Memory Module 08 Results



```
BREAK INSTRUCTION FETCHED 1105.0 ns
 REGISTERS AFTER BREAK

Time =1111.0 ns: REG_ADDR[00] = 00000000 || REG_ADDR[10] = xxxxxxxx
Time =1111.0 ns: REG_ADDR[01] = 00000019 || REG_ADDR[11] = xxxxx
Time =1111.0 ns: REG_ADDR[02] = 000003e8 || REG_ADDR[12] = xxxxx
Time =1111.0 ns: REG_ADDR[03] = ffffffe7 || REG_ADDR[13] = xxxxx
Time =1111.0 ns: REG_ADDR[04] = fffffc18 || REG_ADDR[14] = xxxxx
Time =1111.0 ns: REG_ADDR[05] = 000061a8 || REG_ADDR[15] = xxxxxxxx
Time =1111.0 ns: REG_ADDR[06] = ffff9e58 || REG_ADDR[16] = xxxxxxxx
Time =1111.0 ns: REG_ADDR[07] = ffffffff || REG_ADDR[17] = xxxx
Time =1111.0 ns: REG_ADDR[08] = 000061a8 || REG_ADDR[18] = xxxx
Time =1111.0 ns: REG_ADDR[09] = ffff9e58 || REG_ADDR[1
Time =1111.0 ns: REG_ADDR[0a] = ffffffff || REG_ADDR[1a] = xxxxxxxx
Time =1111.0 ns: REG_ADDR[0b] = ffff9e58 || REG_ADDR[1b] = xxxxxxxx
Time =1111.0 ns: REG_ADDR[0c] = ffffffff || REG_ADDR[1c] = xxxxxxxx
Time =1111.0 ns: REG_ADDR[0d] = 000061a8 || REG_ADDR[1d] = 000003fc
Time =1111.0 ns: REG_ADDR[0e] = 00000000 || REG_ADDR[1e] = xxxxxxxx
Time =1111.0 ns: REG_ADDR[0f] = 10010000 || REG_ADDR[1f] = xxxxxxxx

 MEMORY AFTER BREAK
Time =1111.0 ns: DM[0c0] = xxxxxxxx
Time =1111.0 ns: DM[0c4] = xxxxxxxx
Time =1111.0 ns: DM[0c8] = xxxxxxxx
Time =1111.0 ns: DM[0cc] = xxxxxxxx
Time =1111.0 ns: DM[0d0] = xxxxxxxx
Time =1111.0 ns: DM[0d4] = xxxxxxxx
Time =1111.0 ns: DM[0d8] = xxxxxxxx
Time =1111.0 ns: DM[0dc] = xxxxxxxx
Time =1111.0 ns: DM[0e0] = xxxxxxxx
Time =1111.0 ns: DM[0e4] = xxxxxxxx
Time =1111.0 ns: DM[0e8] = xxxxxxxx
Time =1111.0 ns: DM[0ec] = xxxxxxxx
Time =1111.0 ns: DM[0f0] = xxxxxxxx
Time =1111.0 ns: DM[0f4] = xxxxxxxx
Time =1111.0 ns: DM[0f8] = xxxxxxxx
Time =1111.0 ns: DM[0fc] = xxxxxxxx
```

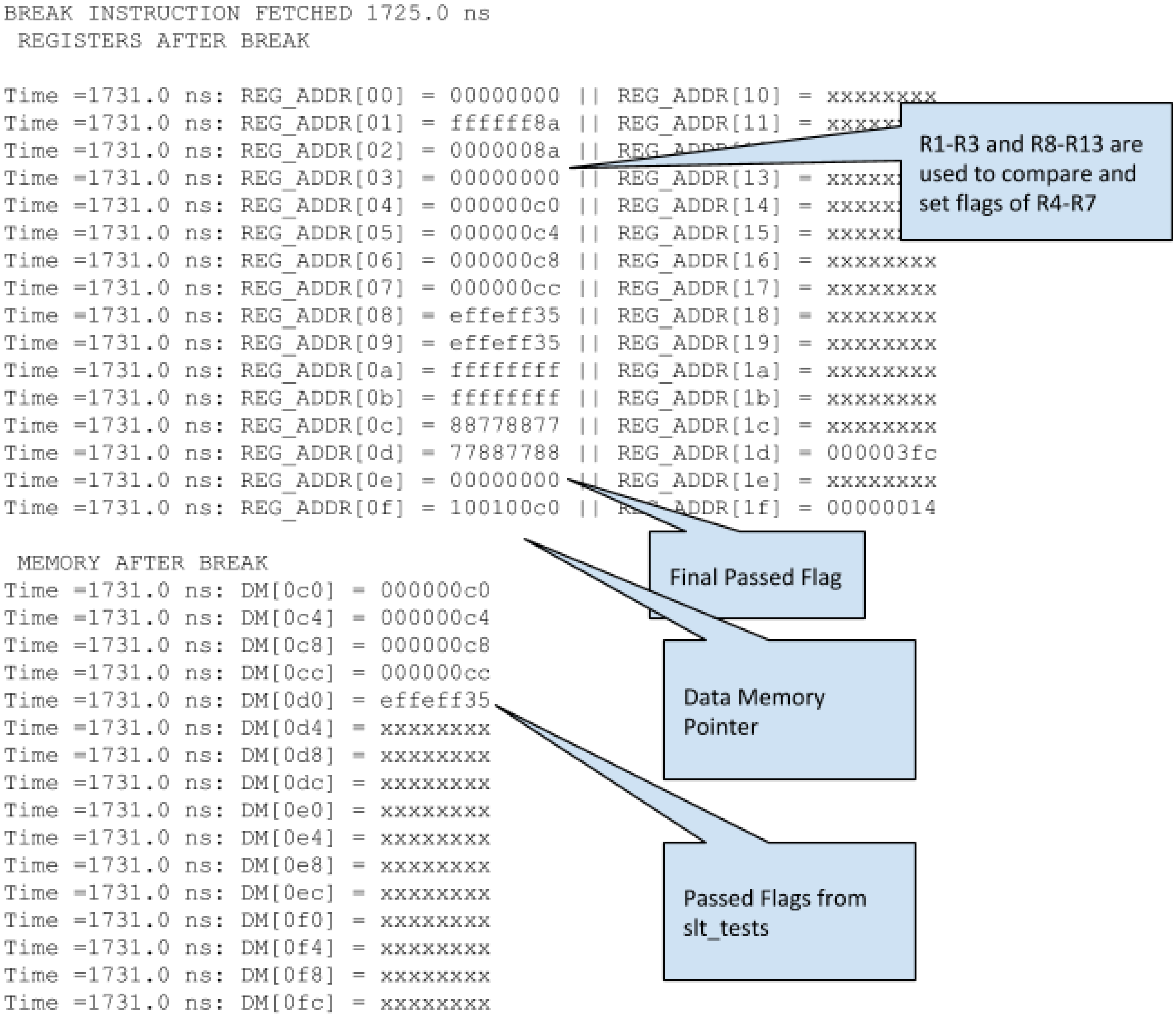
Load R1-R7 from memory

R8-R13 results from multiple

Passed Flag

Data Memory Pointer

9. Memory Module 09 Results



```
BREAK INSTRUCTION FETCHED 1725.0 ns
 REGISTERS AFTER BREAK

Time =1731.0 ns: REG_ADDR[00] = 00000000 || REG_ADDR[10] = xxxxxxxx
Time =1731.0 ns: REG_ADDR[01] = ffffff8a || REG_ADDR[11] = xxxxx
Time =1731.0 ns: REG_ADDR[02] = 0000008a || REG_ADDR[1
Time =1731.0 ns: REG_ADDR[03] = 00000000 || REG_ADDR[13] = xxxxx
Time =1731.0 ns: REG_ADDR[04] = 000000c0 || REG_ADDR[14] = xxxxx
Time =1731.0 ns: REG_ADDR[05] = 000000c4 || REG_ADDR[15] = xxxxx
Time =1731.0 ns: REG_ADDR[06] = 000000c8 || REG_ADDR[16] = xxxxxxxx
Time =1731.0 ns: REG_ADDR[07] = 000000cc || REG_ADDR[17] = xxxxxxxx
Time =1731.0 ns: REG_ADDR[08] = effeff35 || REG_ADDR[18] = xxxxxxxx
Time =1731.0 ns: REG_ADDR[09] = effeff35 || REG_ADDR[19] = xxxxxxxx
Time =1731.0 ns: REG_ADDR[0a] = ffffffff || REG_ADDR[1a] = xxxxxxxx
Time =1731.0 ns: REG_ADDR[0b] = ffffffff || REG_ADDR[1b] = xxxxxxxx
Time =1731.0 ns: REG_ADDR[0c] = 88778877 || REG_ADDR[1c] = xxxxxxxx
Time =1731.0 ns: REG_ADDR[0d] = 77887788 || REG_ADDR[1d] = 000003fc
Time =1731.0 ns: REG_ADDR[0e] = 00000000 || REG_ADDR[1e] = xxxxxxxx
Time =1731.0 ns: REG_ADDR[0f] = 100100c0 || REG_ADDR[1f] = 00000014

 MEMORY AFTER BREAK
Time =1731.0 ns: DM[0c0] = 000000c0
Time =1731.0 ns: DM[0c4] = 000000c4
Time =1731.0 ns: DM[0c8] = 000000c8
Time =1731.0 ns: DM[0cc] = 000000cc
Time =1731.0 ns: DM[0d0] = effeff35
Time =1731.0 ns: DM[0d4] = xxxxxxxx
Time =1731.0 ns: DM[0d8] = xxxxxxxx
Time =1731.0 ns: DM[0dc] = xxxxxxxx
Time =1731.0 ns: DM[0e0] = xxxxxxxx
Time =1731.0 ns: DM[0e4] = xxxxxxxx
Time =1731.0 ns: DM[0e8] = xxxxxxxx
Time =1731.0 ns: DM[0ec] = xxxxxxxx
Time =1731.0 ns: DM[0f0] = xxxxxxxx
Time =1731.0 ns: DM[0f4] = xxxxxxxx
Time =1731.0 ns: DM[0f8] = xxxxxxxx
Time =1731.0 ns: DM[0fc] = xxxxxxxx
```

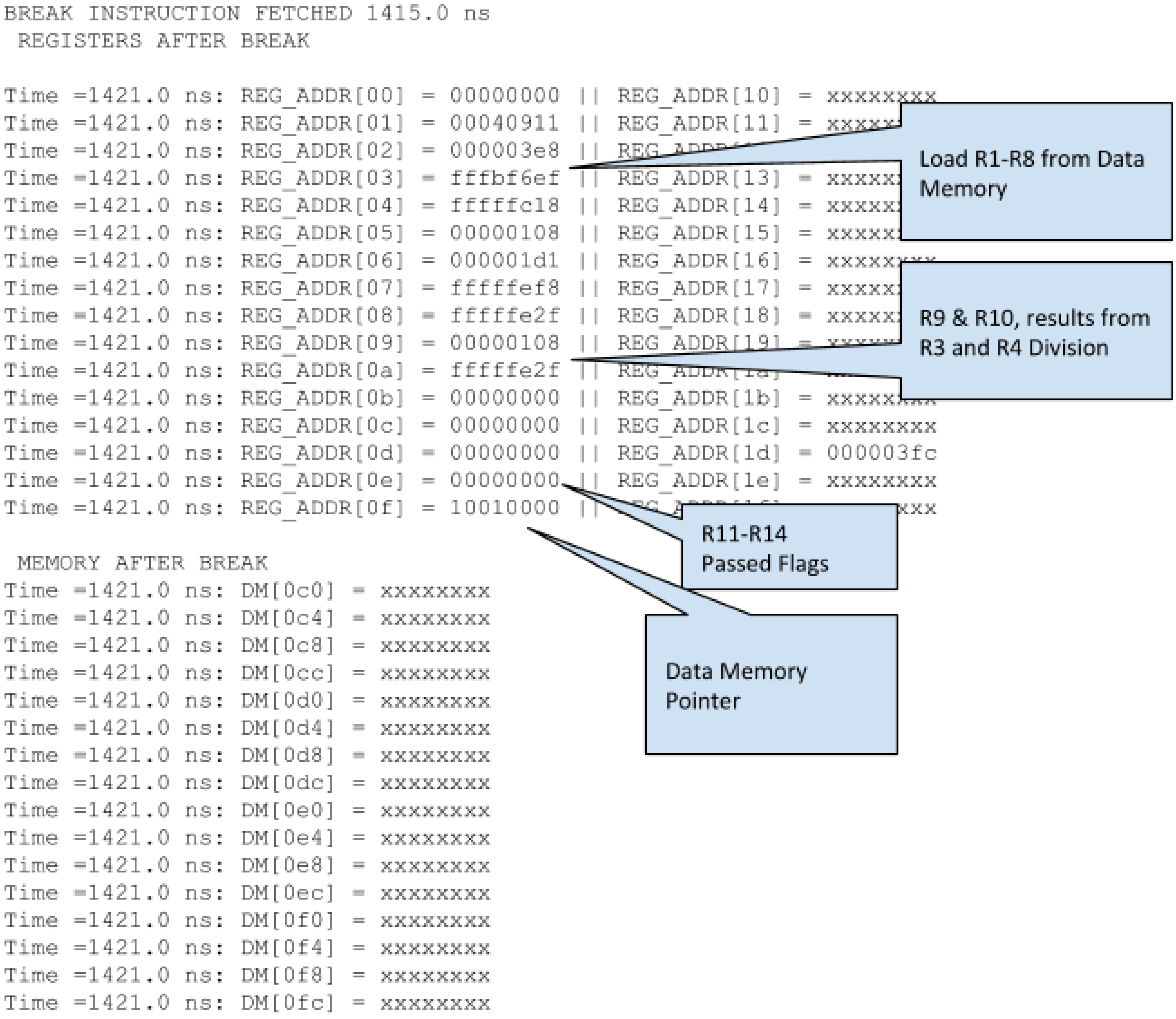
R1-R3 and R8-R13 are used to compare and set flags of R4-R7

Final Passed Flag

Data Memory Pointer

Passed Flags from slt_tests

10. Memory Module 10 Results



```
BREAK INSTRUCTION FETCHED 1415.0 ns
 REGISTERS AFTER BREAK

Time =1421.0 ns: REG_ADDR[00] = 00000000 || REG_ADDR[10] = xxxxxxxx
Time =1421.0 ns: REG_ADDR[01] = 00040911 || REG_ADDR[11] = xxxxx
Time =1421.0 ns: REG_ADDR[02] = 000003e8 || REG_ADDR[1
Time =1421.0 ns: REG_ADDR[03] = fffbf6ef || REG_ADDR[13] = xxxxx
Time =1421.0 ns: REG_ADDR[04] = fffffc18 || REG_ADDR[14] = xxxxx
Time =1421.0 ns: REG_ADDR[05] = 00000108 || REG_ADDR[15] = xxxxx
Time =1421.0 ns: REG_ADDR[06] = 000001d1 || REG_ADDR[16] = xxxxxxxx
Time =1421.0 ns: REG_ADDR[07] = fffffef8 || REG_ADDR[17] = xxxxx
Time =1421.0 ns: REG_ADDR[08] = fffffe2f || REG_ADDR[18] = xxxxx
Time =1421.0 ns: REG_ADDR[09] = 00000108 || REG_ADDR[19] = xxxxx
Time =1421.0 ns: REG_ADDR[0a] = fffffe2f || REG_ADDR[1a]
Time =1421.0 ns: REG_ADDR[0b] = 00000000 || REG_ADDR[1b] = xxxxxxxx
Time =1421.0 ns: REG_ADDR[0c] = 00000000 || REG_ADDR[1c] = xxxxxxxx
Time =1421.0 ns: REG_ADDR[0d] = 00000000 || REG_ADDR[1d] = 000003fc
Time =1421.0 ns: REG_ADDR[0e] = 00000000 || REG_ADDR[1e] = xxxxxxxx
Time =1421.0 ns: REG_ADDR[0f] = 10010000 ||               xxx

 MEMORY AFTER BREAK
Time =1421.0 ns: DM[0c0] = xxxxxxxx
Time =1421.0 ns: DM[0c4] = xxxxxxxx
Time =1421.0 ns: DM[0c8] = xxxxxxxx
Time =1421.0 ns: DM[0cc] = xxxxxxxx
Time =1421.0 ns: DM[0d0] = xxxxxxxx
Time =1421.0 ns: DM[0d4] = xxxxxxxx
Time =1421.0 ns: DM[0d8] = xxxxxxxx
Time =1421.0 ns: DM[0dc] = xxxxxxxx
Time =1421.0 ns: DM[0e0] = xxxxxxxx
Time =1421.0 ns: DM[0e4] = xxxxxxxx
Time =1421.0 ns: DM[0e8] = xxxxxxxx
Time =1421.0 ns: DM[0ec] = xxxxxxxx
Time =1421.0 ns: DM[0f0] = xxxxxxxx
Time =1421.0 ns: DM[0f4] = xxxxxxxx
Time =1421.0 ns: DM[0f8] = xxxxxxxx
Time =1421.0 ns: DM[0fc] = xxxxxxxx
```

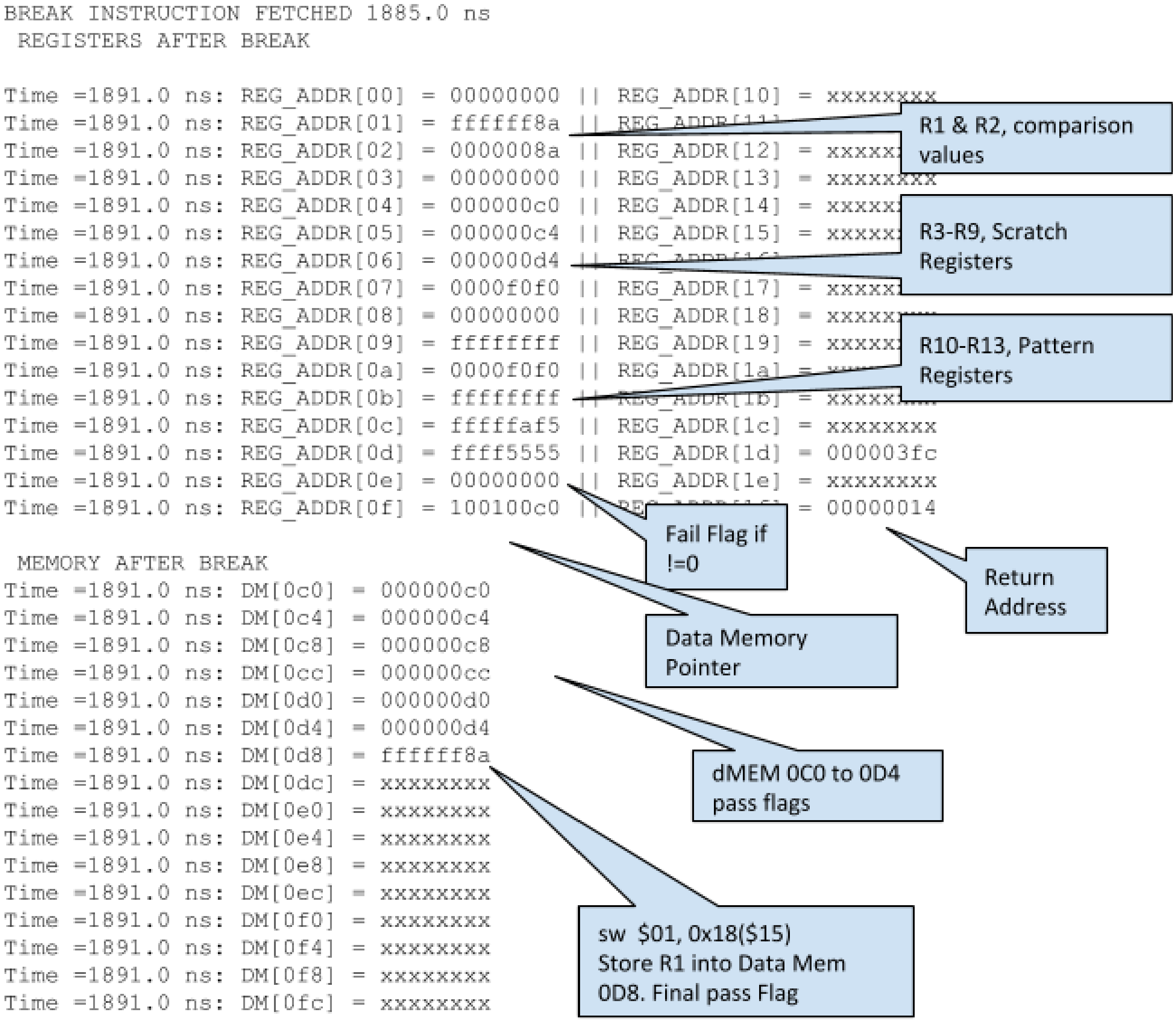
Load R1-R8 from Data Memory

R9 & R10, results from R3 and R4 Division

R11-R14 Passed Flags

Data Memory Pointer

11. Memory Module 11 Results



```
BREAK INSTRUCTION FETCHED 1885.0 ns
 REGISTERS AFTER BREAK

Time =1891.0 ns: REG_ADDR[00] = 00000000 || REG_ADDR[10] = xxxxxxxx
Time =1891.0 ns: REG_ADDR[01] = ffffff8a || REG_ADDR[11
Time =1891.0 ns: REG_ADDR[02] = 0000008a || REG_ADDR[12] = xxxxx
Time =1891.0 ns: REG_ADDR[03] = 00000000 || REG_ADDR[13] = xxxxxxxx
Time =1891.0 ns: REG_ADDR[04] = 000000c0 || REG_ADDR[14] = xxxxx
Time =1891.0 ns: REG_ADDR[05] = 000000c4 || REG_ADDR[15] = xxxxx
Time =1891.0 ns: REG_ADDR[06] = 000000d4 || REG_ADDR[16
Time =1891.0 ns: REG_ADDR[07] = 0000f0f0 || REG_ADDR[17] = xxxxx
Time =1891.0 ns: REG_ADDR[08] = 00000000 || REG_ADDR[18] = xxxxx
Time =1891.0 ns: REG_ADDR[09] = ffffffff || REG_ADDR[19] = xxxxx
Time =1891.0 ns: REG_ADDR[0a] = 0000f0f0 || REG_ADDR[1a] =
Time =1891.0 ns: REG_ADDR[0b] = ffffffff || REG_ADDR[1b] = xxxxx
Time =1891.0 ns: REG_ADDR[0c] = fffffaf5 || REG_ADDR[1c] = xxxxxxxx
Time =1891.0 ns: REG_ADDR[0d] = ffff5555 || REG_ADDR[1d] = 000003fc
Time =1891.0 ns: REG_ADDR[0e] = 00000000 || REG_ADDR[1e] = xxxxxxxx
Time =1891.0 ns: REG_ADDR[0f] = 100100c0 ||              = 00000014

 MEMORY AFTER BREAK
Time =1891.0 ns: DM[0c0] = 000000c0
Time =1891.0 ns: DM[0c4] = 000000c4
Time =1891.0 ns: DM[0c8] = 000000c8
Time =1891.0 ns: DM[0cc] = 000000cc
Time =1891.0 ns: DM[0d0] = 000000d0
Time =1891.0 ns: DM[0d4] = 000000d4
Time =1891.0 ns: DM[0d8] = ffffff8a
Time =1891.0 ns: DM[0dc] = xxxxxxxx
Time =1891.0 ns: DM[0e0] = xxxxxxxx
Time =1891.0 ns: DM[0e4] = xxxxxxxx
Time =1891.0 ns: DM[0e8] = xxxxxxxx
Time =1891.0 ns: DM[0ec] = xxxxxxxx
Time =1891.0 ns: DM[0f0] = xxxxxxxx
Time =1891.0 ns: DM[0f4] = xxxxxxxx
Time =1891.0 ns: DM[0f8] = xxxxxxxx
Time =1891.0 ns: DM[0fc] = xxxxxxxx
```

R1 & R2, comparison values

R3-R9, Scratch Registers

R10-R13, Pattern Registers

Fail Flag if !=0

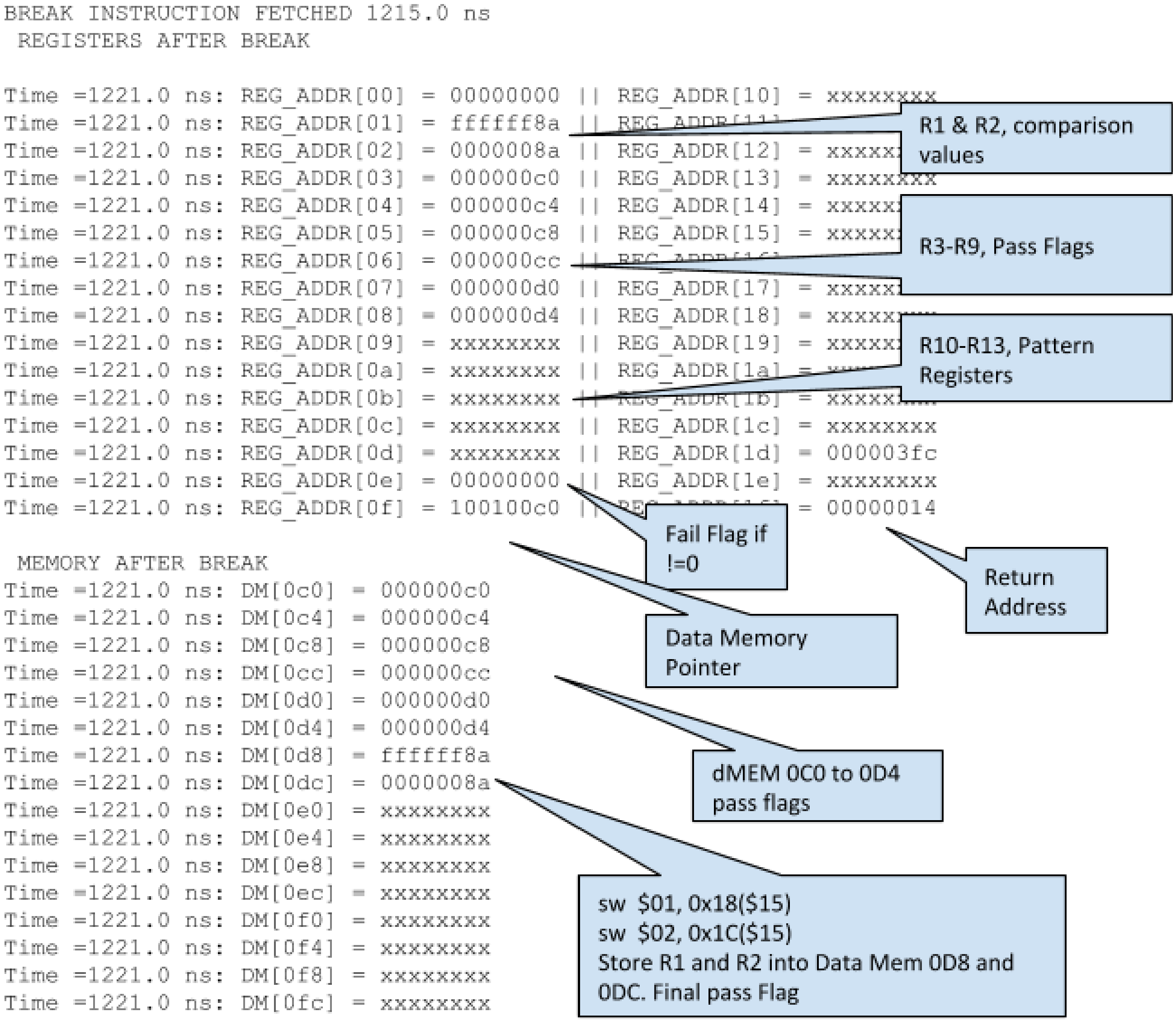
Return Address

Data Memory Pointer

dMEM 0C0 to 0D4 pass flags

sw $01, 0x18($15)
Store R1 into Data Mem 0D8. Final pass Flag

12. Memory Module 12 Results



```
BREAK INSTRUCTION FETCHED 1215.0 ns
 REGISTERS AFTER BREAK

Time =1221.0 ns: REG_ADDR[00] = 00000000 || REG_ADDR[10] = xxxxxxxx
Time =1221.0 ns: REG_ADDR[01] = ffffff8a || REG_ADDR[11]
Time =1221.0 ns: REG_ADDR[02] = 0000008a || REG_ADDR[12] = xxxxx
Time =1221.0 ns: REG_ADDR[03] = 000000c0 || REG_ADDR[13] = xxxxxxxx
Time =1221.0 ns: REG_ADDR[04] = 000000c4 || REG_ADDR[14] = xxxxx
Time =1221.0 ns: REG_ADDR[05] = 000000c8 || REG_ADDR[15] = xxxxx
Time =1221.0 ns: REG_ADDR[06] = 000000cc || REG_ADDR[16]
Time =1221.0 ns: REG_ADDR[07] = 000000d0 || REG_ADDR[17] = xxxxx
Time =1221.0 ns: REG_ADDR[08] = 000000d4 || REG_ADDR[18] = xxxxxxxx
Time =1221.0 ns: REG_ADDR[09] = xxxxxxxx || REG_ADDR[19] = xxxxx
Time =1221.0 ns: REG_ADDR[0a] = xxxxxxxx || REG_ADDR[1a] = 
Time =1221.0 ns: REG_ADDR[0b] = xxxxxxxx || REG_ADDR[1b] = xxxxxxxx
Time =1221.0 ns: REG_ADDR[0c] = xxxxxxxx || REG_ADDR[1c] = xxxxxxxx
Time =1221.0 ns: REG_ADDR[0d] = xxxxxxxx || REG_ADDR[1d] = 000003fc
Time =1221.0 ns: REG_ADDR[0e] = 00000000 || REG_ADDR[1e] = xxxxxxxx
Time =1221.0 ns: REG_ADDR[0f] = 100100c0 || REG_ADDR[1f] = 00000014

 MEMORY AFTER BREAK
Time =1221.0 ns: DM[0c0] = 000000c0
Time =1221.0 ns: DM[0c4] = 000000c4
Time =1221.0 ns: DM[0c8] = 000000c8
Time =1221.0 ns: DM[0cc] = 000000cc
Time =1221.0 ns: DM[0d0] = 000000d0
Time =1221.0 ns: DM[0d4] = 000000d4
Time =1221.0 ns: DM[0d8] = ffffff8a
Time =1221.0 ns: DM[0dc] = 0000008a
Time =1221.0 ns: DM[0e0] = xxxxxxxx
Time =1221.0 ns: DM[0e4] = xxxxxxxx
Time =1221.0 ns: DM[0e8] = xxxxxxxx
Time =1221.0 ns: DM[0ec] = xxxxxxxx
Time =1221.0 ns: DM[0f0] = xxxxxxxx
Time =1221.0 ns: DM[0f4] = xxxxxxxx
Time =1221.0 ns: DM[0f8] = xxxxxxxx
Time =1221.0 ns: DM[0fc] = xxxxxxxx
```

R1 & R2, comparison values

R3-R9, Pass Flags

R10-R13, Pattern Registers

Fail Flag if !=0

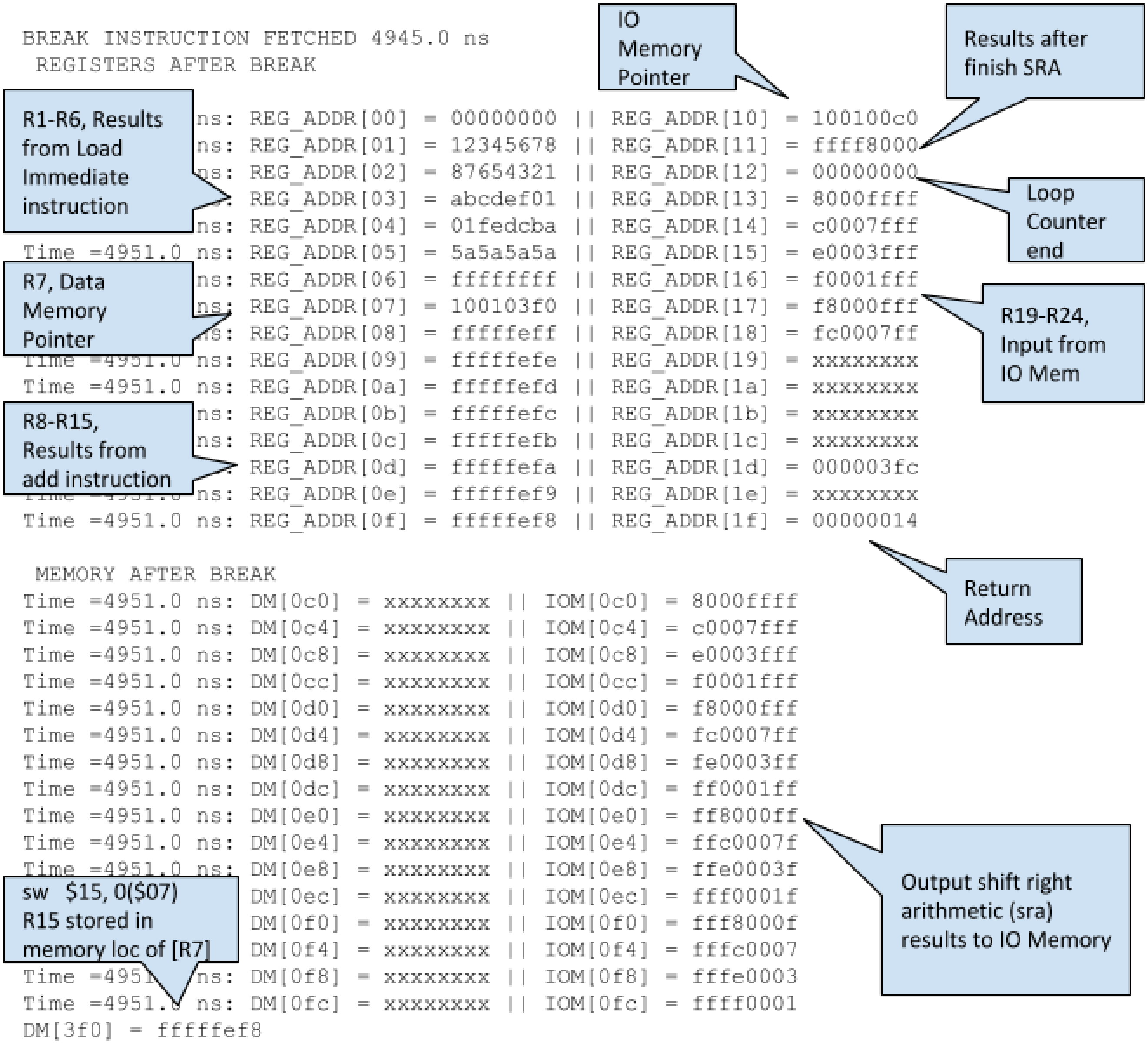
Return Address

Data Memory Pointer

dMEM 0C0 to 0D4 pass flags

sw $01, 0x18($15)
sw $02, 0x1C($15)
Store R1 and R2 into Data Mem 0D8 and 0DC. Final pass Flag

13. Memory Module 13 Results



```
BREAK INSTRUCTION FETCHED 4945.0 ns
 REGISTERS AFTER BREAK
                 ns: REG_ADDR[00] = 00000000 || REG_ADDR[10] = 100100c0
                 ns: REG_ADDR[01] = 12345678 || REG_ADDR[11] = ffff8000
                 ns: REG_ADDR[02] = 87654321 || REG_ADDR[12] = 00000000
                   : REG_ADDR[03] = abcdef01 || REG_ADDR[13] = 8000ffff
                 ns: REG_ADDR[04] = 01fedcba || REG_ADDR[14] = c0007fff
Time =4951.0 ns: REG_ADDR[05] = 5a5a5a5a || REG_ADDR[15] = e0003fff
                 ns: REG_ADDR[06] = ffffffff || REG_ADDR[16] = f0001fff
                 ns; REG_ADDR[07] = 100103f0 || REG_ADDR[17] = f8000fff
                  s: REG_ADDR[08] = fffffeff || REG_ADDR[18] = fc0007ff
Time =4951.0 ns: REG_ADDR[09] = fffffefe || REG_ADDR[19] = xxxxxxxx
Time =4951.0 ns: REG_ADDR[0a] = fffffefd || REG_ADDR[1a] = xxxxxxxx
                 ns: REG_ADDR[0b] = fffffefc || REG_ADDR[1b] = xxxxxxxx
                 ns: REG_ADDR[0c] = fffffefb || REG_ADDR[1c] = xxxxxxxx
                   : REG_ADDR[0d] = fffffefa || REG_ADDR[1d] = 000003fc
Time =4951.0 ns: REG_ADDR[0e] = fffffef9 || REG_ADDR[1e] = xxxxxxxx
Time =4951.0 ns: REG_ADDR[0f] = fffffef8 || REG_ADDR[1f] = 00000014

 MEMORY AFTER BREAK
Time =4951.0 ns: DM[0c0] = xxxxxxxx || IOM[0c0] = 8000ffff
Time =4951.0 ns: DM[0c4] = xxxxxxxx || IOM[0c4] = c0007fff
Time =4951.0 ns: DM[0c8] = xxxxxxxx || IOM[0c8] = e0003fff
Time =4951.0 ns: DM[0cc] = xxxxxxxx || IOM[0cc] = f0001fff
Time =4951.0 ns: DM[0d0] = xxxxxxxx || IOM[0d0] = f8000fff
Time =4951.0 ns: DM[0d4] = xxxxxxxx || IOM[0d4] = fc0007ff
Time =4951.0 ns: DM[0d8] = xxxxxxxx || IOM[0d8] = fe0003ff
Time =4951.0 ns: DM[0dc] = xxxxxxxx || IOM[0dc] = ff0001ff
Time =4951.0 ns: DM[0e0] = xxxxxxxx || IOM[0e0] = ff8000ff
Time =4951.0 ns: DM[0e4] = xxxxxxxx || IOM[0e4] = ffc0007f
Time =4951.0 ns: DM[0e8] = xxxxxxxx || IOM[0e8] = ffe0003f
                 DM[0ec] = xxxxxxxx || IOM[0ec] = fff0001f
                 DM[0f0] = xxxxxxxx || IOM[0f0] = fff8000f
                 DM[0f4] = xxxxxxxx || IOM[0f4] = fffc0007
Time =4951   ns: DM[0f8] = xxxxxxxx || IOM[0f8] = fffe0003
Time =4951.0 ns: DM[0fc] = xxxxxxxx || IOM[0fc] = ffff0001
DM[3f0] = fffffef8
```

IO Memory Pointer

Results after finish SRA

R1-R6, Results from Load Immediate instruction

Loop Counter end

R7, Data Memory Pointer

R19-R24, Input from IO Mem

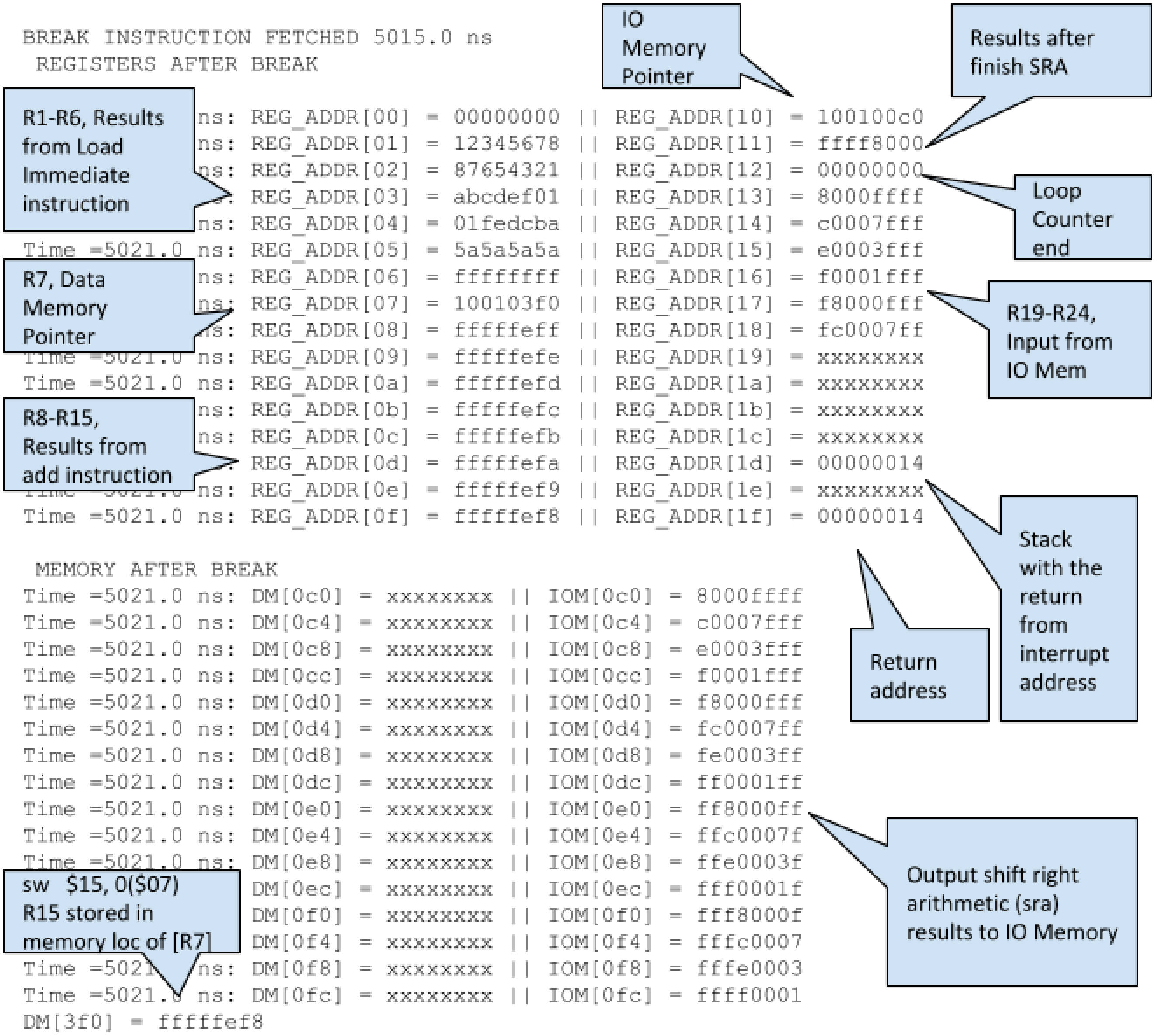
R8-R15, Results from add instruction

Return Address

sw $15, 0($07) R15 stored in memory loc of [R7]

Output shift right arithmetic (sra) results to IO Memory

14. Memory Module 14 Results

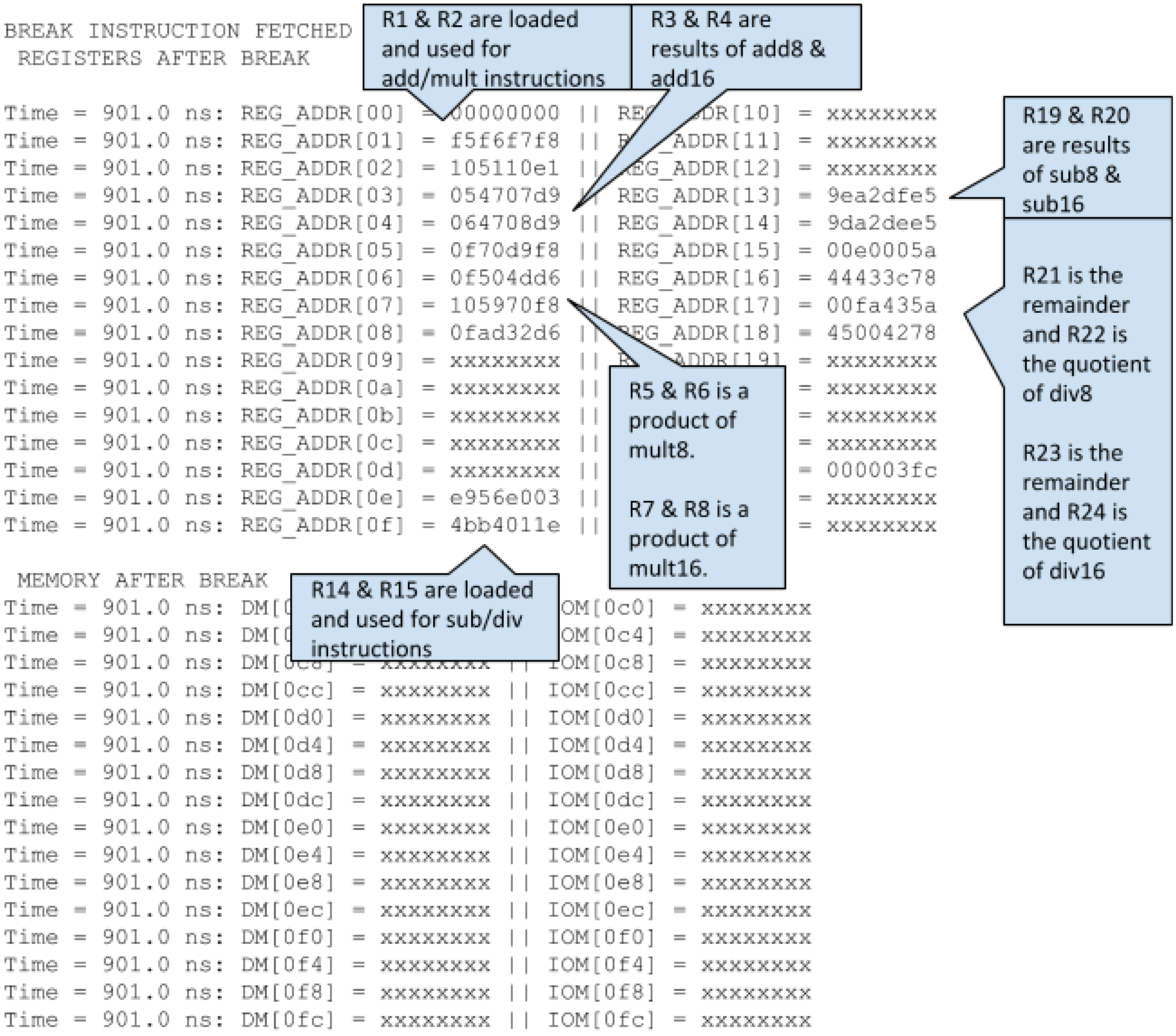


```
BREAK INSTRUCTION FETCHED 5015.0 ns
 REGISTERS AFTER BREAK
              ns: REG_ADDR[00] = 00000000 || REG_ADDR[10] = 100100c0
              ns: REG_ADDR[01] = 12345678 || REG_ADDR[11] = ffff8000
              ns: REG_ADDR[02] = 87654321 || REG_ADDR[12] = 00000000
                  REG_ADDR[03] = abcdef01 || REG_ADDR[13] = 8000ffff
              ns: REG_ADDR[04] = 01fedcba || REG_ADDR[14] = c0007fff
Time =5021.0 ns: REG_ADDR[05] = 5a5a5a5a || REG_ADDR[15] = e0003fff
              ns: REG_ADDR[06] = ffffffff || REG_ADDR[16] = f0001fff
              ns: REG_ADDR[07] = 100103f0 || REG_ADDR[17] = f8000fff
                  REG_ADDR[08] = fffffeff || REG_ADDR[18] = fc0007ff
Time =5021.0 ns: REG_ADDR[09] = fffffefe || REG_ADDR[19] = xxxxxxxx
Time =5021.0 ns: REG_ADDR[0a] = fffffefd || REG_ADDR[1a] = xxxxxxxx
              ns: REG_ADDR[0b] = fffffefc || REG_ADDR[1b] = xxxxxxxx
              ns: REG_ADDR[0c] = fffffefb || REG_ADDR[1c] = xxxxxxxx
                  REG_ADDR[0d] = fffffefa || REG_ADDR[1d] = 00000014
Time =5021.0 ns: REG_ADDR[0e] = fffffef9 || REG_ADDR[1e] = xxxxxxxx
Time =5021.0 ns: REG_ADDR[0f] = fffffef8 || REG_ADDR[1f] = 00000014

 MEMORY AFTER BREAK
Time =5021.0 ns: DM[0c0] = xxxxxxxx || IOM[0c0] = 8000ffff
Time =5021.0 ns: DM[0c4] = xxxxxxxx || IOM[0c4] = c0007fff
Time =5021.0 ns: DM[0c8] = xxxxxxxx || IOM[0c8] = e0003fff
Time =5021.0 ns: DM[0cc] = xxxxxxxx || IOM[0cc] = f0001fff
Time =5021.0 ns: DM[0d0] = xxxxxxxx || IOM[0d0] = f8000fff
Time =5021.0 ns: DM[0d4] = xxxxxxxx || IOM[0d4] = fc0007ff
Time =5021.0 ns: DM[0d8] = xxxxxxxx || IOM[0d8] = fe0003ff
Time =5021.0 ns: DM[0dc] = xxxxxxxx || IOM[0dc] = ff0001ff
Time =5021.0 ns: DM[0e0] = xxxxxxxx || IOM[0e0] = ff8000ff
Time =5021.0 ns: DM[0e4] = xxxxxxxx || IOM[0e4] = ffc0007f
Time =5021.0 ns: DM[0e8] = xxxxxxxx || IOM[0e8] = ffe0003f
                  DM[0ec] = xxxxxxxx || IOM[0ec] = fff0001f
                  DM[0f0] = xxxxxxxx || IOM[0f0] = fff8000f
                  DM[0f4] = xxxxxxxx || IOM[0f4] = fffc0007
Time =502    ns: DM[0f8] = xxxxxxxx || IOM[0f8] = fffe0003
Time =5021.  ns: DM[0fc] = xxxxxxxx || IOM[0fc] = ffff0001
DM[3f0] = fffffef8
```

IO Memory Pointer
Results after finish SRA
R1-R6, Results from Load Immediate instruction
Loop Counter end
R7, Data Memory Pointer
R19-R24, Input from IO Mem
R8-R15, Results from add instruction
Stack with the return from interrupt address
Return address
Output shift right arithmetic (sra) results to IO Memory
sw $15, 0($07) R15 stored in memory loc of [R7]

15. Memory Module 15 Vector Results



```
BREAK INSTRUCTION FETCHED
 REGISTERS AFTER BREAK

Time = 901.0 ns: REG_ADDR[00] = 00000000 || REG_ADDR[10] = xxxxxxxx
Time = 901.0 ns: REG_ADDR[01] = f5f6f7f8 || REG_ADDR[11] = xxxxxxxx
Time = 901.0 ns: REG_ADDR[02] = 105110e1 || REG_ADDR[12] = xxxxxxxx
Time = 901.0 ns: REG_ADDR[03] = 054707d9 || REG_ADDR[13] = 9ea2dfe5
Time = 901.0 ns: REG_ADDR[04] = 064708d9 || REG_ADDR[14] = 9da2dee5
Time = 901.0 ns: REG_ADDR[05] = 0f70d9f8 || REG_ADDR[15] = 00e0005a
Time = 901.0 ns: REG_ADDR[06] = 0f504dd6 || REG_ADDR[16] = 44433c78
Time = 901.0 ns: REG_ADDR[07] = 105970f8 || REG_ADDR[17] = 00fa435a
Time = 901.0 ns: REG_ADDR[08] = 0fad32d6 || REG_ADDR[18] = 45004278
Time = 901.0 ns: REG_ADDR[09] = xxxxxxxx || REG_ADDR[19] = xxxxxxxx
Time = 901.0 ns: REG_ADDR[0a] = xxxxxxxx ||                = xxxxxxxx
Time = 901.0 ns: REG_ADDR[0b] = xxxxxxxx ||                = xxxxxxxx
Time = 901.0 ns: REG_ADDR[0c] = xxxxxxxx ||                = xxxxxxxx
Time = 901.0 ns: REG_ADDR[0d] = xxxxxxxx ||                = 000003fc
Time = 901.0 ns: REG_ADDR[0e] = e956e003 ||                = xxxxxxxx
Time = 901.0 ns: REG_ADDR[0f] = 4bb4011e ||                = xxxxxxxx

 MEMORY AFTER BREAK
Time = 901.0 ns: DM[0                        OM[0c0] = xxxxxxxx
Time = 901.0 ns: DM[0                        OM[0c4] = xxxxxxxx
Time = 901.0 ns: DM[0c8] = xxxxxxxx || IOM[0c8] = xxxxxxxx
Time = 901.0 ns: DM[0cc] = xxxxxxxx || IOM[0cc] = xxxxxxxx
Time = 901.0 ns: DM[0d0] = xxxxxxxx || IOM[0d0] = xxxxxxxx
Time = 901.0 ns: DM[0d4] = xxxxxxxx || IOM[0d4] = xxxxxxxx
Time = 901.0 ns: DM[0d8] = xxxxxxxx || IOM[0d8] = xxxxxxxx
Time = 901.0 ns: DM[0dc] = xxxxxxxx || IOM[0dc] = xxxxxxxx
Time = 901.0 ns: DM[0e0] = xxxxxxxx || IOM[0e0] = xxxxxxxx
Time = 901.0 ns: DM[0e4] = xxxxxxxx || IOM[0e4] = xxxxxxxx
Time = 901.0 ns: DM[0e8] = xxxxxxxx || IOM[0e8] = xxxxxxxx
Time = 901.0 ns: DM[0ec] = xxxxxxxx || IOM[0ec] = xxxxxxxx
Time = 901.0 ns: DM[0f0] = xxxxxxxx || IOM[0f0] = xxxxxxxx
Time = 901.0 ns: DM[0f4] = xxxxxxxx || IOM[0f4] = xxxxxxxx
Time = 901.0 ns: DM[0f8] = xxxxxxxx || IOM[0f8] = xxxxxxxx
Time = 901.0 ns: DM[0fc] = xxxxxxxx || IOM[0fc] = xxxxxxxx
```

R1 & R2 are loaded and used for add/mult instructions

R3 & R4 are results of add8 & add16

R19 & R20 are results of sub8 & sub16

R21 is the remainder and R22 is the quotient of div8

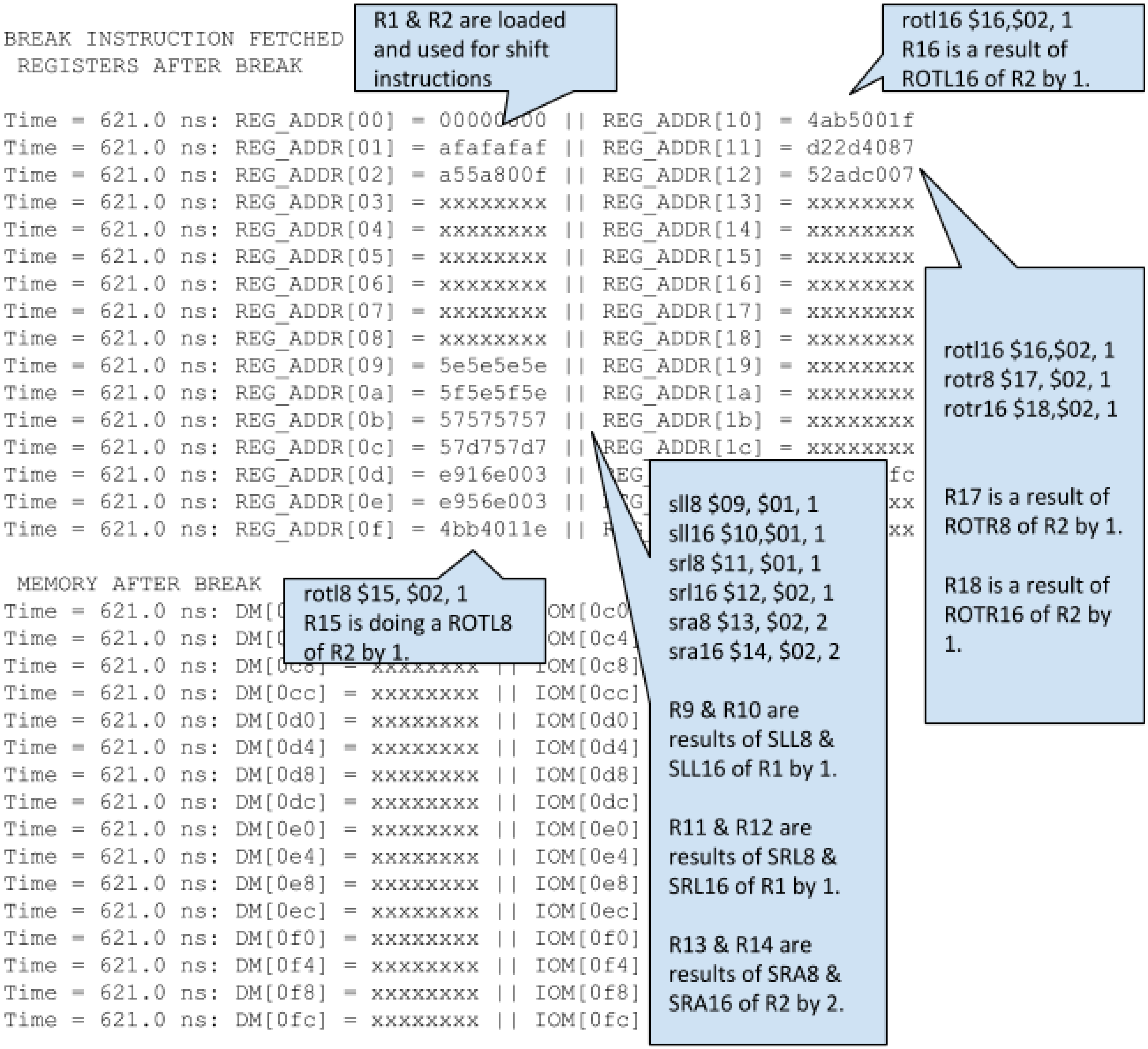
R23 is the remainder and R24 is the quotient of div16

R5 & R6 is a product of mult8.

R7 & R8 is a product of mult16.

R14 & R15 are loaded and used for sub/div instructions

16. Memory Module 15 Vector_2 Results



```
BREAK INSTRUCTION FETCHED
 REGISTERS AFTER BREAK

Time = 621.0 ns: REG_ADDR[00] = 00000000 || REG_ADDR[10] = 4ab5001f
Time = 621.0 ns: REG_ADDR[01] = afafafaf || REG_ADDR[11] = d22d4087
Time = 621.0 ns: REG_ADDR[02] = a55a800f || REG_ADDR[12] = 52adc007
Time = 621.0 ns: REG_ADDR[03] = xxxxxxxx || REG_ADDR[13] = xxxxxxxx
Time = 621.0 ns: REG_ADDR[04] = xxxxxxxx || REG_ADDR[14] = xxxxxxxx
Time = 621.0 ns: REG_ADDR[05] = xxxxxxxx || REG_ADDR[15] = xxxxxxxx
Time = 621.0 ns: REG_ADDR[06] = xxxxxxxx || REG_ADDR[16] = xxxxxxxx
Time = 621.0 ns: REG_ADDR[07] = xxxxxxxx || REG_ADDR[17] = xxxxxxxx
Time = 621.0 ns: REG_ADDR[08] = xxxxxxxx || REG_ADDR[18] = xxxxxxxx
Time = 621.0 ns: REG_ADDR[09] = 5e5e5e5e || REG_ADDR[19] = xxxxxxxx
Time = 621.0 ns: REG_ADDR[0a] = 5f5e5f5e || REG_ADDR[1a] = xxxxxxxx
Time = 621.0 ns: REG_ADDR[0b] = 57575757 || REG_ADDR[1b] = xxxxxxxx
Time = 621.0 ns: REG_ADDR[0c] = 57d757d7 || REG_ADDR[1c] = xxxxxxxx
Time = 621.0 ns: REG_ADDR[0d] = e916e003 || REG                 fc
Time = 621.0 ns: REG_ADDR[0e] = e956e003 ||  EG                 xx
Time = 621.0 ns: REG_ADDR[0f] = 4bb4011e || R                   xx

 MEMORY AFTER BREAK
Time = 621.0 ns: DM[0                    OM[0c0
Time = 621.0 ns: DM[0                    OM[0c4
Time = 621.0 ns: DM[0c8] = xxxxxxxx || IOM[0c8]
Time = 621.0 ns: DM[0cc] = xxxxxxxx || IOM[0cc]
Time = 621.0 ns: DM[0d0] = xxxxxxxx || IOM[0d0]
Time = 621.0 ns: DM[0d4] = xxxxxxxx || IOM[0d4]
Time = 621.0 ns: DM[0d8] = xxxxxxxx || IOM[0d8]
Time = 621.0 ns: DM[0dc] = xxxxxxxx || IOM[0dc]
Time = 621.0 ns: DM[0e0] = xxxxxxxx || IOM[0e0]
Time = 621.0 ns: DM[0e4] = xxxxxxxx || IOM[0e4]
Time = 621.0 ns: DM[0e8] = xxxxxxxx || IOM[0e8]
Time = 621.0 ns: DM[0ec] = xxxxxxxx || IOM[0ec]
Time = 621.0 ns: DM[0f0] = xxxxxxxx || IOM[0f0]
Time = 621.0 ns: DM[0f4] = xxxxxxxx || IOM[0f4]
Time = 621.0 ns: DM[0f8] = xxxxxxxx || IOM[0f8]
Time = 621.0 ns: DM[0fc] = xxxxxxxx || IOM[0fc]
```

R1 & R2 are loaded and used for shift instructions

rotl16 $16,$02, 1
R16 is a result of ROTL16 of R2 by 1.

rotl16 $16,$02, 1
rotr8 $17, $02, 1
rotr16 $18,$02, 1

R17 is a result of ROTR8 of R2 by 1.

R18 is a result of ROTR16 of R2 by 1.

sll8 $09, $01, 1
sll16 $10,$01, 1
srl8 $11, $01, 1
srl16 $12, $02, 1
sra8 $13, $02, 2
sra16 $14, $02, 2

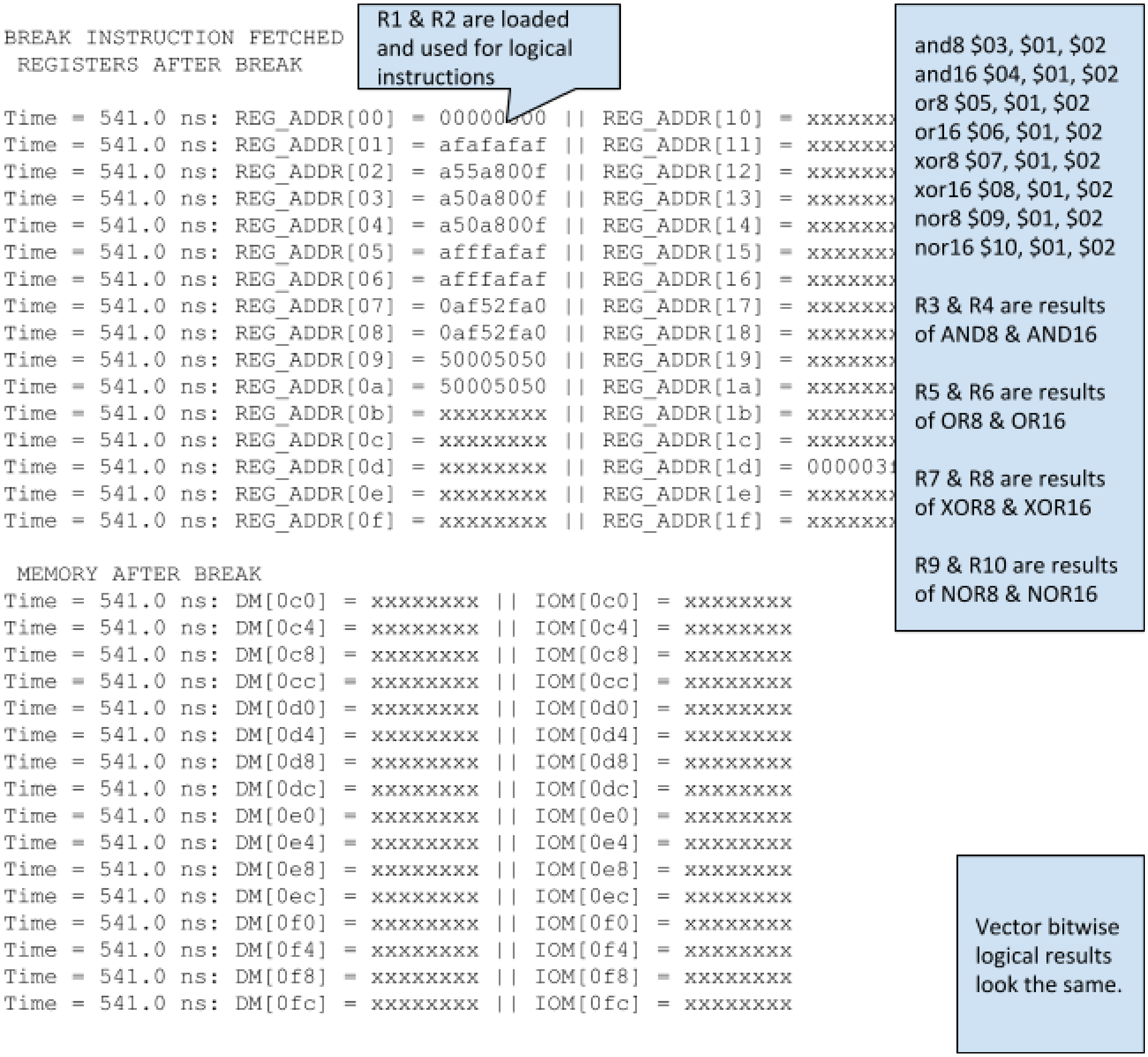
R9 & R10 are results of SLL8 & SLL16 of R1 by 1.

R11 & R12 are results of SRL8 & SRL16 of R1 by 1.

R13 & R14 are results of SRA8 & SRA16 of R2 by 2.

rotl8 $15, $02, 1
R15 is doing a ROTL8 of R2 by 1.

16. Memory Module 15 Vector_3 Results



```
BREAK INSTRUCTION FETCHED
 REGISTERS AFTER BREAK

Time = 541.0 ns: REG_ADDR[00] = 00000000 || REG_ADDR[10] = xxxxxxx
Time = 541.0 ns: REG_ADDR[01] = afafafaf || REG_ADDR[11] = xxxxxxx
Time = 541.0 ns: REG_ADDR[02] = a55a800f || REG_ADDR[12] = xxxxxxx
Time = 541.0 ns: REG_ADDR[03] = a50a800f || REG_ADDR[13] = xxxxxxx
Time = 541.0 ns: REG_ADDR[04] = a50a800f || REG_ADDR[14] = xxxxxxx
Time = 541.0 ns: REG_ADDR[05] = afffafaf || REG_ADDR[15] = xxxxxxx
Time = 541.0 ns: REG_ADDR[06] = afffafaf || REG_ADDR[16] = xxxxxxx
Time = 541.0 ns: REG_ADDR[07] = 0af52fa0 || REG_ADDR[17] = xxxxxxx
Time = 541.0 ns: REG_ADDR[08] = 0af52fa0 || REG_ADDR[18] = xxxxxxx
Time = 541.0 ns: REG_ADDR[09] = 50005050 || REG_ADDR[19] = xxxxxxx
Time = 541.0 ns: REG_ADDR[0a] = 50005050 || REG_ADDR[1a] = xxxxxxx
Time = 541.0 ns: REG_ADDR[0b] = xxxxxxxx || REG_ADDR[1b] = xxxxxxx
Time = 541.0 ns: REG_ADDR[0c] = xxxxxxxx || REG_ADDR[1c] = xxxxxxx
Time = 541.0 ns: REG_ADDR[0d] = xxxxxxxx || REG_ADDR[1d] = 000003f
Time = 541.0 ns: REG_ADDR[0e] = xxxxxxxx || REG_ADDR[1e] = xxxxxxx
Time = 541.0 ns: REG_ADDR[0f] = xxxxxxxx || REG_ADDR[1f] = xxxxxxx

 MEMORY AFTER BREAK
Time = 541.0 ns: DM[0c0] = xxxxxxxx || IOM[0c0] = xxxxxxxx
Time = 541.0 ns: DM[0c4] = xxxxxxxx || IOM[0c4] = xxxxxxxx
Time = 541.0 ns: DM[0c8] = xxxxxxxx || IOM[0c8] = xxxxxxxx
Time = 541.0 ns: DM[0cc] = xxxxxxxx || IOM[0cc] = xxxxxxxx
Time = 541.0 ns: DM[0d0] = xxxxxxxx || IOM[0d0] = xxxxxxxx
Time = 541.0 ns: DM[0d4] = xxxxxxxx || IOM[0d4] = xxxxxxxx
Time = 541.0 ns: DM[0d8] = xxxxxxxx || IOM[0d8] = xxxxxxxx
Time = 541.0 ns: DM[0dc] = xxxxxxxx || IOM[0dc] = xxxxxxxx
Time = 541.0 ns: DM[0e0] = xxxxxxxx || IOM[0e0] = xxxxxxxx
Time = 541.0 ns: DM[0e4] = xxxxxxxx || IOM[0e4] = xxxxxxxx
Time = 541.0 ns: DM[0e8] = xxxxxxxx || IOM[0e8] = xxxxxxxx
Time = 541.0 ns: DM[0ec] = xxxxxxxx || IOM[0ec] = xxxxxxxx
Time = 541.0 ns: DM[0f0] = xxxxxxxx || IOM[0f0] = xxxxxxxx
Time = 541.0 ns: DM[0f4] = xxxxxxxx || IOM[0f4] = xxxxxxxx
Time = 541.0 ns: DM[0f8] = xxxxxxxx || IOM[0f8] = xxxxxxxx
Time = 541.0 ns: DM[0fc] = xxxxxxxx || IOM[0fc] = xxxxxxxx
```

R1 & R2 are loaded and used for logical instructions

and8 $03, $01, $02
and16 $04, $01, $02
or8 $05, $01, $02
or16 $06, $01, $02
xor8 $07, $01, $02
xor16 $08, $01, $02
nor8 $09, $01, $02
nor16 $10, $01, $02

R3 & R4 are results of AND8 & AND16

R5 & R6 are results of OR8 & OR16

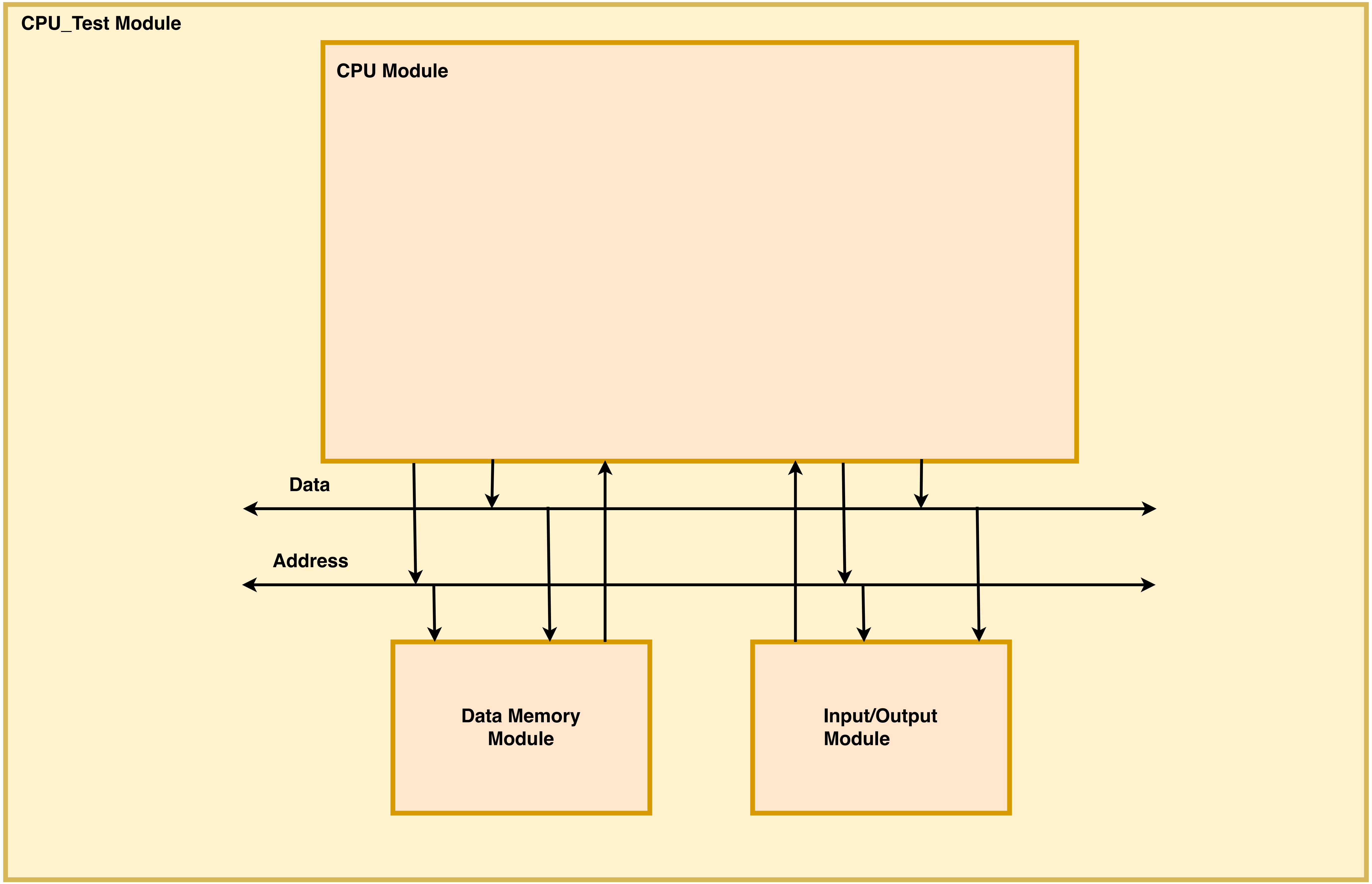
R7 & R8 are results of XOR8 & XOR16

R9 & R10 are results of NOR8 & NOR16

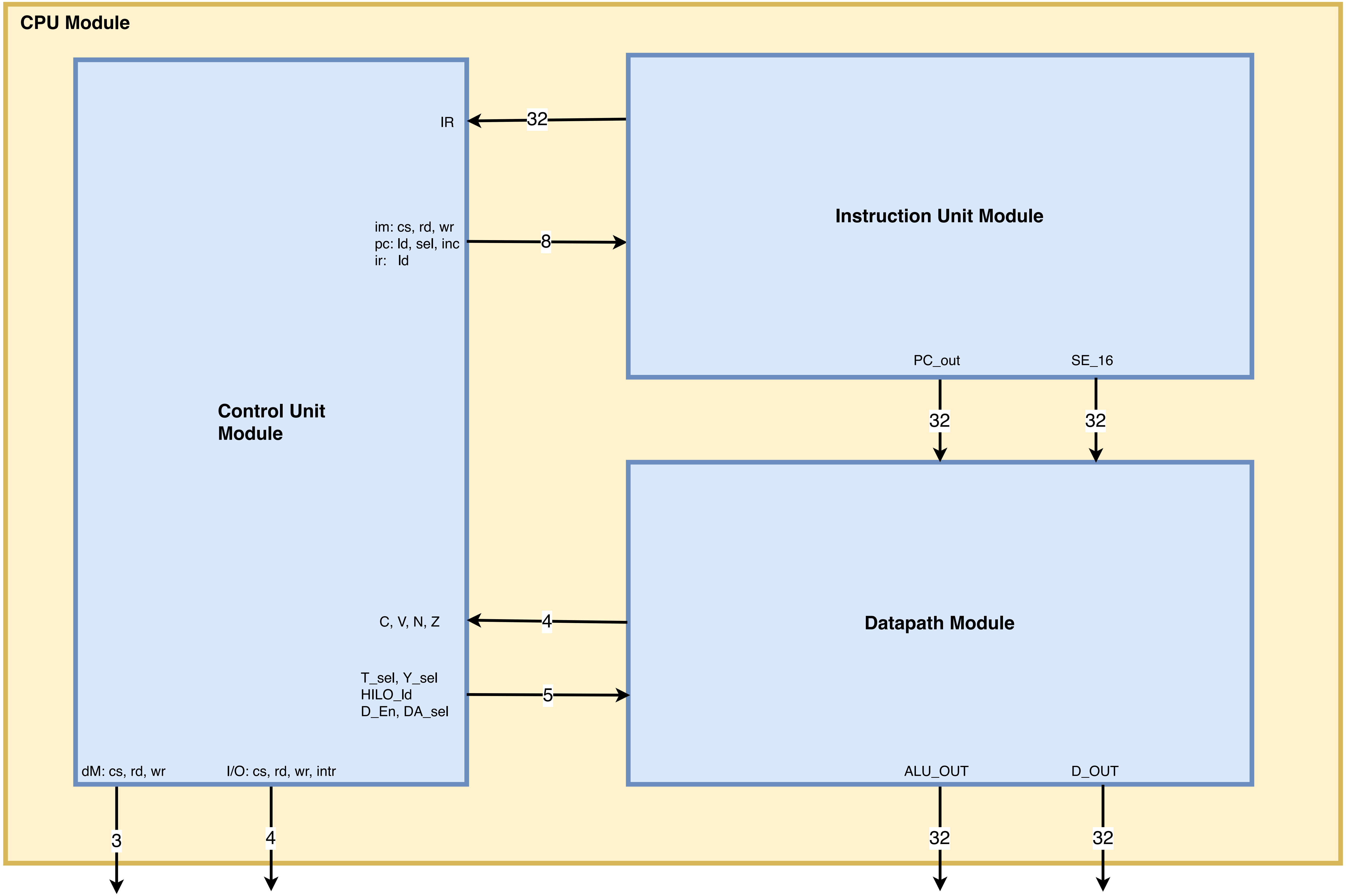
Vector bitwise logical results look the same.

IV. Hardware Implementation

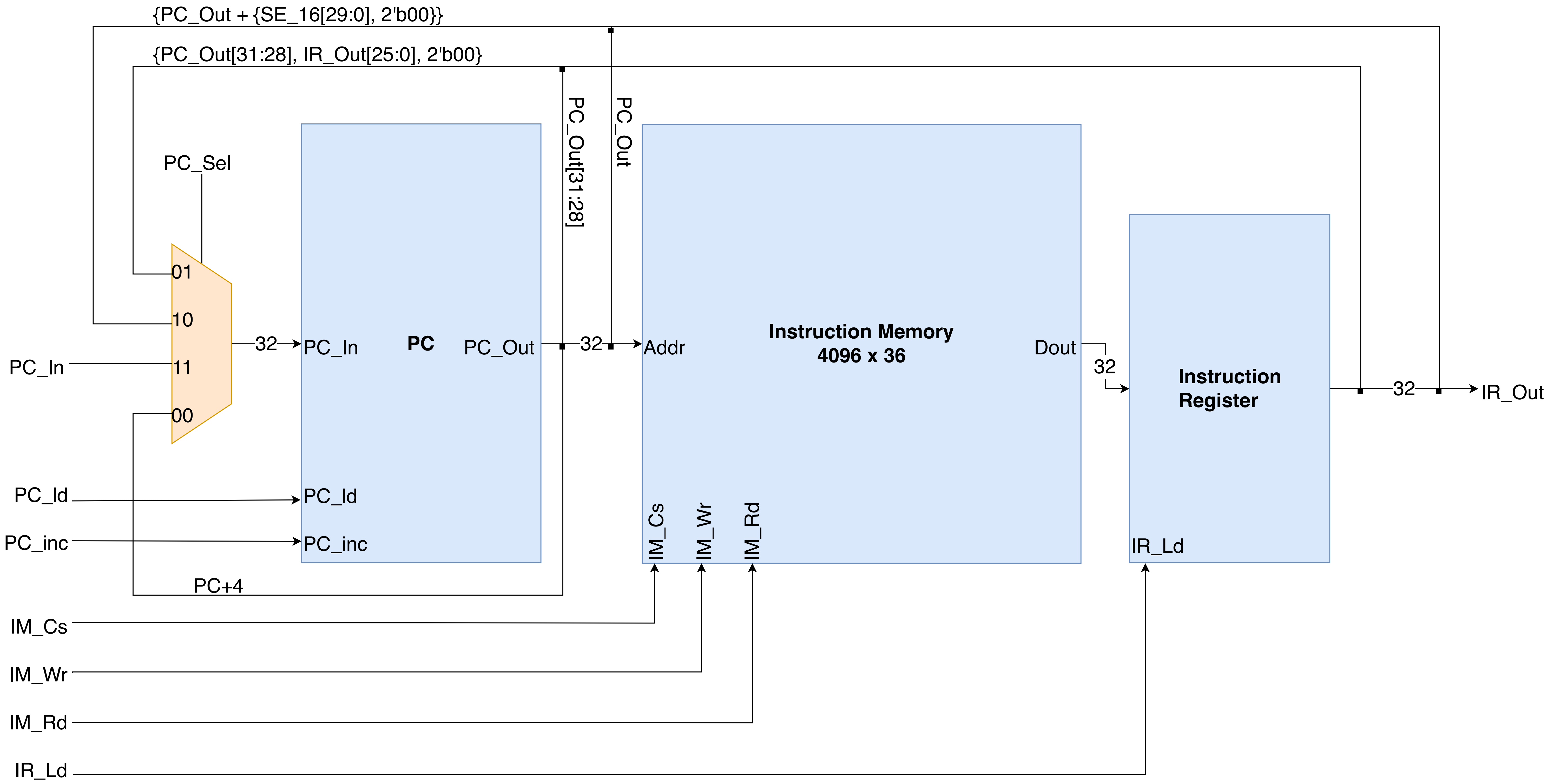
CPU Top Module Block

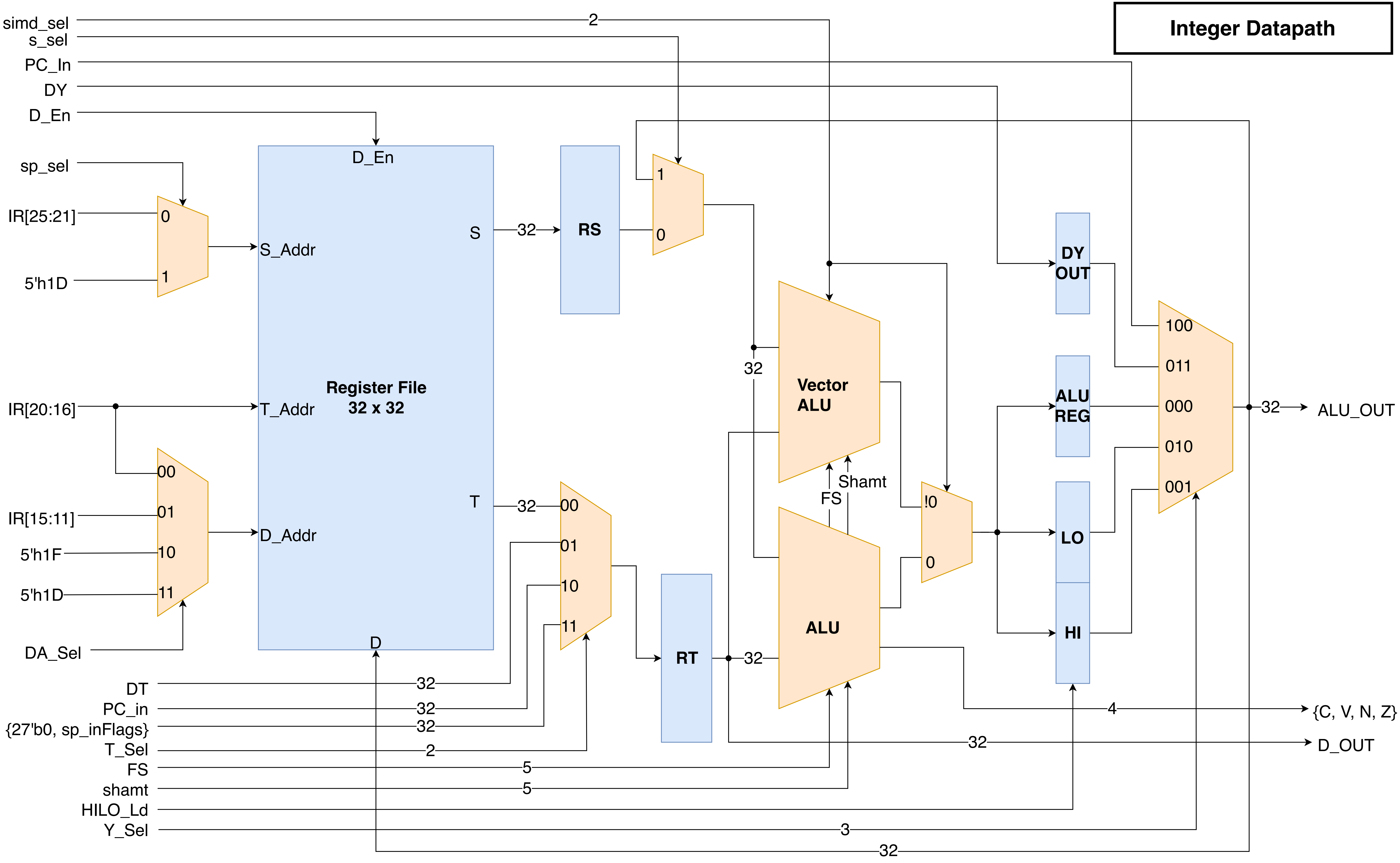


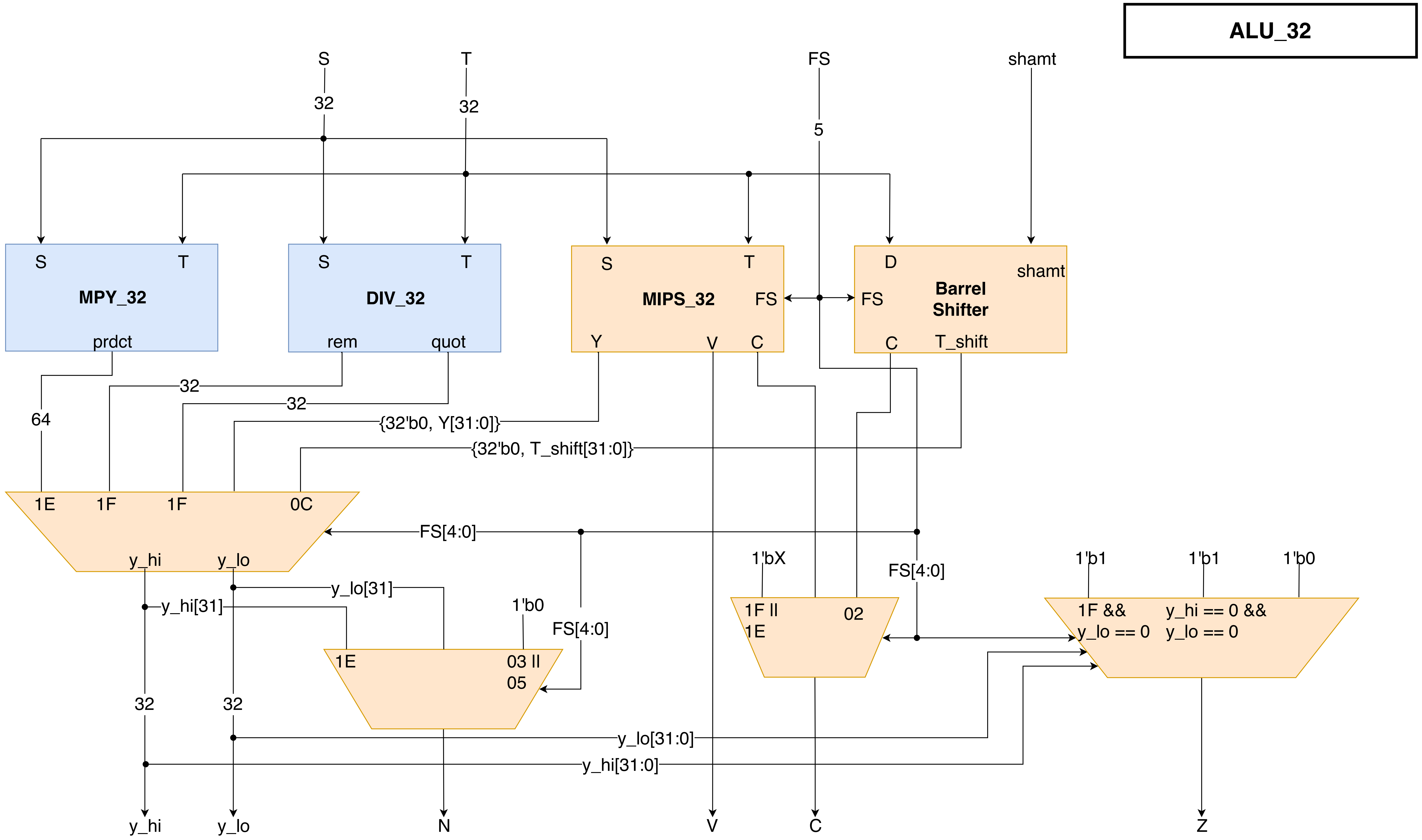
CPU Module Block

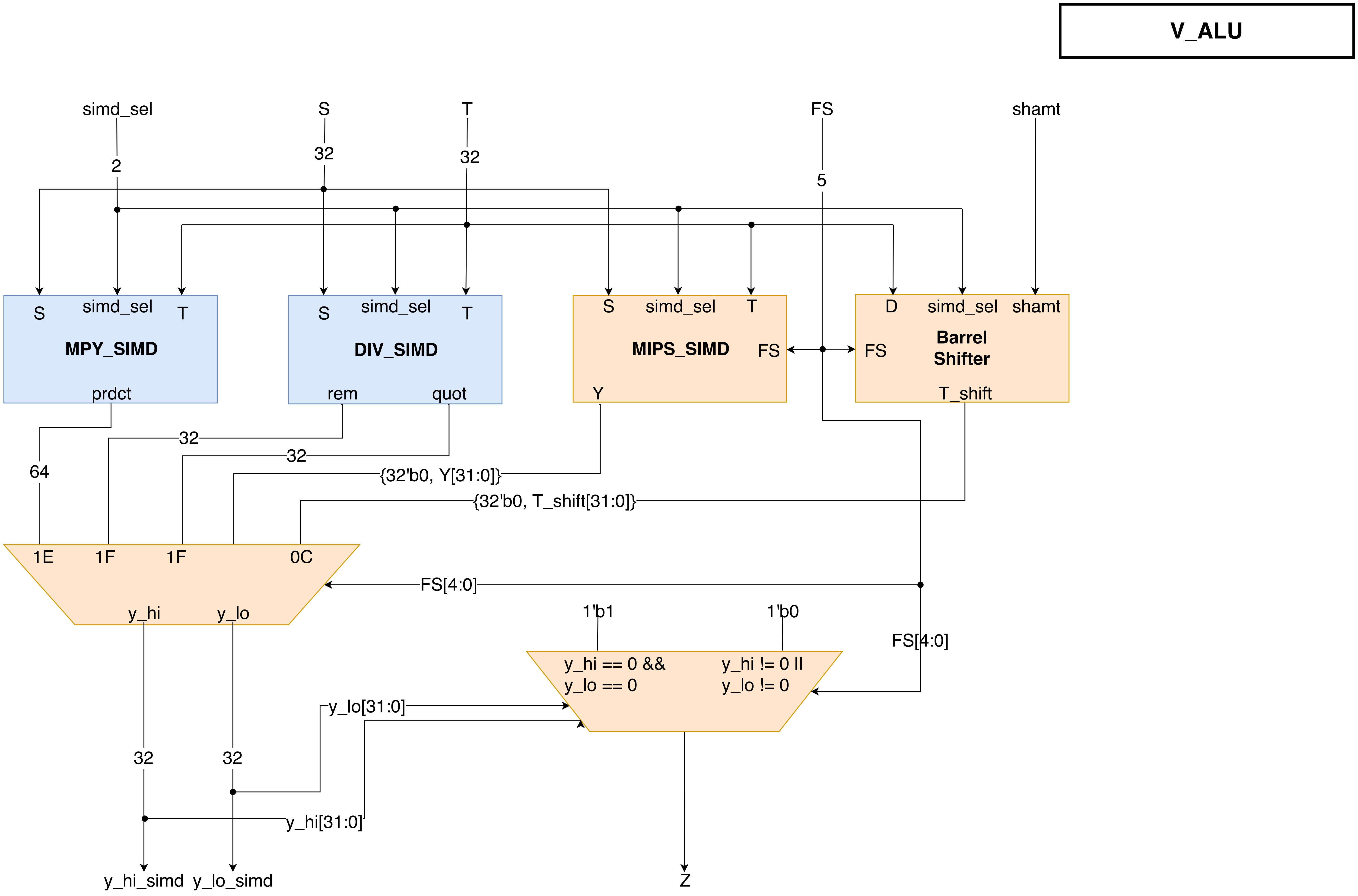


Instruction Unit




Integer Datapath
simd_sel
s_sel
PC_In
DY
D_En
sp_sel
IR[25:21]
5'h1D
IR[20:16]
IR[15:11]
5'h1F
5'h1D
DA_Sel
0
1
00
01
10
11
D_En
S_Addr
T_Addr
D_Addr
Register File
32 x 32
S
T
D
32
RS
1
0
32
00
01
10
11
RT
DT
PC_in
{27'b0, sp_inFlags}
T_Sel
FS
shamt
HILO_Ld
Y_Sel
32
32
32
2
5
5
3
32
2
32
Vector
ALU
Shamt
FS
ALU
32
!0
0
DY
OUT
ALU
REG
LO
HI
100
011
000
010
001
32
ALU_OUT
4
{C, V, N, Z}
32
D_OUT


ALU_32
S
T
FS
shamt
32
32
5
S
T
MPY_32
prdct
S
T
DIV_32
rem
quot
S
T
MIPS_32
FS
Y
V
C
D
shamt
Barrel
Shifter
FS
C
T_shift
64
32
32
{32'b0, Y[31:0]}
{32'b0, T_shift[31:0]}
1E
1F
1F
0C
FS[4:0]
y_hi
y_lo
y_lo[31]
y_hi[31]
1'b0
FS[4:0]
1E
03 ||
05
1'bX
1F ||
1E
02
FS[4:0]
1'b1
1'b1
1'b0
1F &&
y_lo == 0
y_hi == 0 &&
y_lo == 0
32
32
y_lo[31:0]
y_hi[31:0]
y_hi
y_lo
N
V
C
Z



V_ALU
simd_sel
S
T
FS
shamt
2
32
32
5
S
simd_sel
T
MPY_SIMD
prdct
S
simd_sel
T
DIV_SIMD
rem
quot
S
simd_sel
T
MIPS_SIMD
FS
Y
D
simd_sel
shamt
Barrel Shifter
FS
T_shift
64
32
32
{32'b0, Y[31:0]}
{32'b0, T_shift[31:0]}
1E
1F
1F
0C
FS[4:0]
y_hi
y_lo
1'b1
1'b0
FS[4:0]
y_hi == 0 &&
y_lo == 0
y_hi != 0 ll
y_lo != 0
y_lo[31:0]
32
32
y_hi[31:0]
y_hi_simd
y_lo_simd
Z

V. Additional Discussion and Comments

At the beginning, our discussed enhancement to the MIPS32 processor was to do a pipeline, but that proved to take up too much time and had to be discarded to make sure to meet deadline requirements. Our thinking was to implement stage registers for each pipeline state and carry the control words through each of the registers for the basic step but that was where our problem remained before we ended up changing our minds.

Given more time, we may revisit this option.