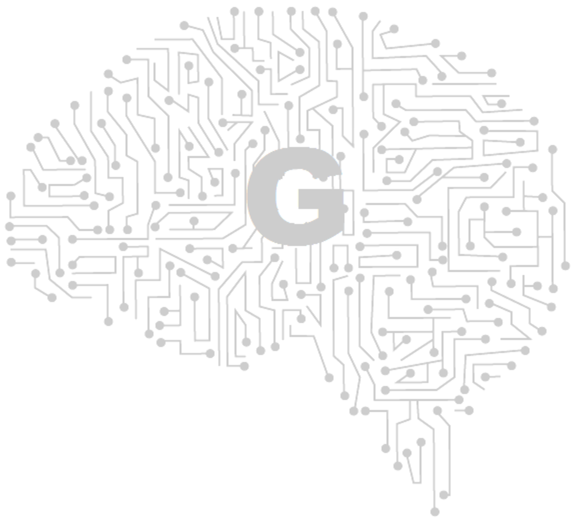

G

GBRAINS PROGRAMMER'S REFERENCE MANUAL

Brian Ortiz Bryan Linares Grace Daliwan

011817687 008236252 014047330

Computer Architecture

Fall 2018- T/TH 11am

Instructor: Robert Allision

Nov.27, 2018

Table of Contents

I. Purpose

"Scientists study the world as it is;
engineers create the world that
has never been."
—Theodore von Karman

"If you had four hours to chop down a tree.
I would spend three hours sharpening my axe."
— Abraham Lincoln

This is a complete Programmer's Reference Manual and user documentation for the first revision instruction implementation of the GBRAINS 32/64-bit Processor.

This CPU Instruction Set Architecture (ISA) project began August 26th, 2018 and was completed by November 27th, 2018. The ISA began with six developmental projects that contributed to the completion of this project, which were verified by instructor Robert Allison. The finale of development on this project was the completion of the added Vector and Floating-point data-path enhancements.

Sections covered:

Harvard Memory Architecture and Organization style governs the instructions and data being transmitted in and received from memory.

Processor Register Set shows the set of registers in the processor available to the user in detail with expected usage.

Data Types define the name, size, and range of each data type expected by the operations.

Addressing Modes defines the various value designation specification for generating a value or effective address of an operand at run-time.

Instruction Set and Binary Instruction Formats contain all instructions recognized by the processor which permit computational tasks.

Verilog Implementation is the design and verification section that contains the source code of the processors design. Annotated log files will verify instruction execution.

Hardware Implementation will show block diagrams of the entire processor from a top-down level including major data and address paths.

Additional Discussions will discuss the enhancement philosophy and implementation journey of this processor.

II. Instruction set Architecture

The Instruction Set Architecture (ISA) of a processor is an abstraction model that serves as an interface between computer hardware and computer software.

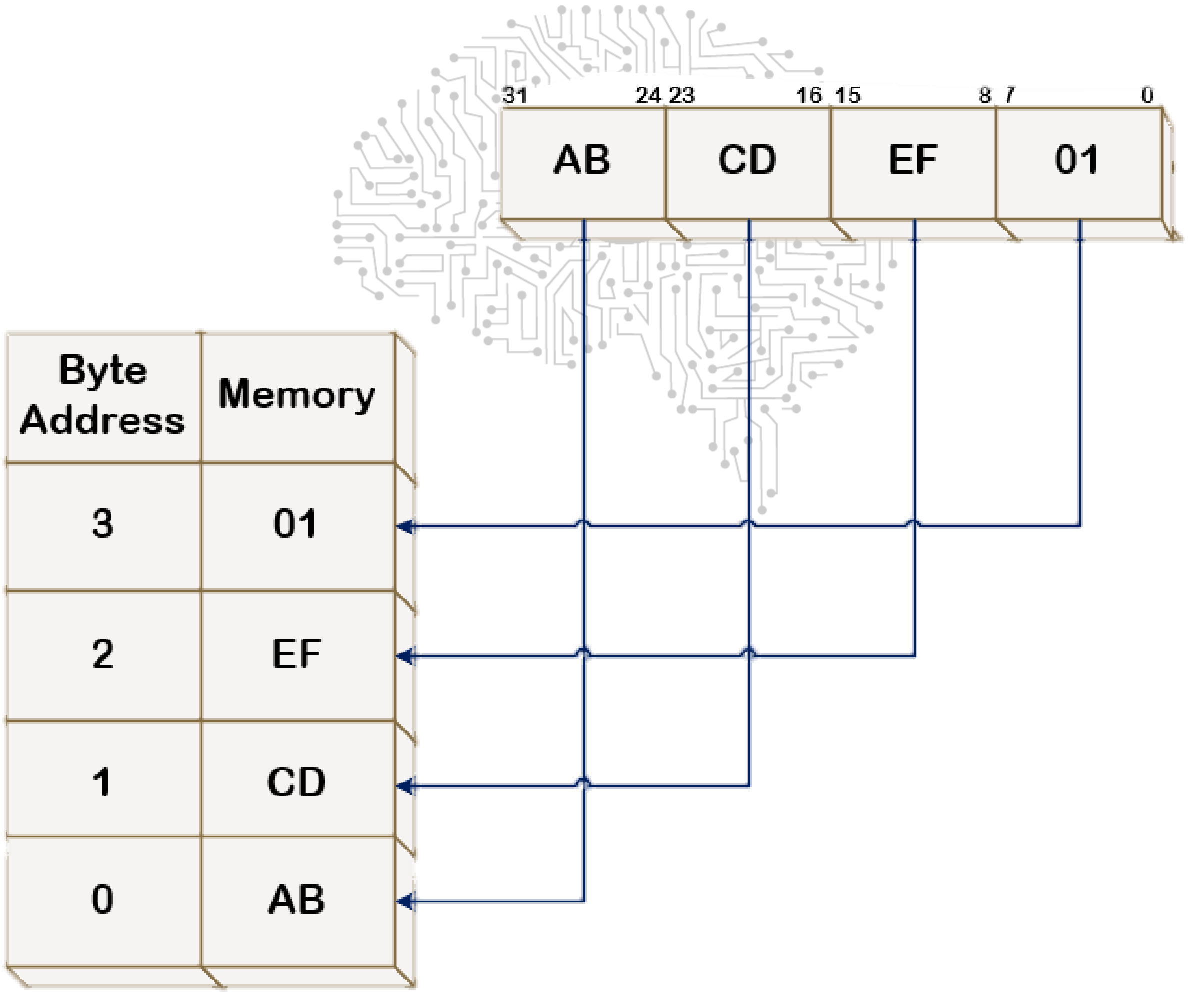
The Instruction Set Architecture is the architecture of the processor for which he/she is writing code. The perspective seen by a programmer are the ***registers*** of the processor available for their use, the ***data types*** that can be operated on, the ***addressing modes*** available for obtaining operands, the different ***operations*** available to process/manipulate data, and the ***binary format***, or machine codes, of the instructions- the actual language of the processor. (Instructor Robert Allison: Instruction Set Architecture PDF)

At the most fundamental level, computers simply "***execute instructions.***"

A. Harvard Memory Architecture and Organization

Harvard Architecture is characterized by separate storage and buses for instructions and data. This makes it possible to fetch instructions and read/write data simultaneously, in the implementation of the MIPS 32-bit Processor.

The memory is a 32-bit address space and is byte-addressable where each memory location contains at most one-byte (8 bits). Therefore, 32-bit memory operands are stored in four consecutive memory locations in big endian format. Which means that the most significant byte of the word is stored in the least significant address.



B. Machine Register Set

The following are 32 bits wide, contained within the Integer datapath register file:

Name	Register Number	Usage	Preserved on call
$zero	0	The constant value 0	n.a.
$at	1	Reserved for the assembler	n.a.
$v0-$v1	2-3	Value for result and expressions	no
$a0-$a3	4-7	Arguments(procedure/function)	yes
$t0-$t7	8-15	temporaries	no
$s0-$s7	16-23	saved	yes
$t8-$t9	24-25	More temporaries	no
$k0-$k1	26-27	Reserved for the operating system	n.a.
$gp	28	global pointer	yes
$sp	29	stack pointer	yes
$fp	30	frame pointer	yes
$ra	31	return address	yes

The following two Register sets are 64-bits wide each, in the Double and Vectored Register Files :

Name	Register Number	Usage	Preserved on call
$f0-$f32	0-32	32 General Purpose 64-bit double precision floating point registers	n.a.
$v0-$v32	0-32	32 General Purpose 64 bit vectored integer registers	n.a.

C. Data Types

32-bit Signed Integer: ranges from -2,147,483,648 to 2,147,483,647 representing a total of 4GB.

32-bit Unsigned Integer: ranges from 0 to 4,294,967,295 or 4GB.

64-bit Double Precision Floating Point: ranges $\pm 2.23 \times 10^{-308}$ to $\pm 1.80 \times 10^{308}$.

Conforms to IEEE754 Floating Point Standard for Double Precision

64-bit total Integer Vector: ranges from 0 to 255 at each byte and 0 to 18,446,744,073,709,551,615 in total range value or 2 EB.

Vector operations expect varying packings of integer sizes (8 bit, 32 bit signed, 64 bit, etc.), please reference the operation details for proper usage.

Flags

FLAGS				
Interrupt Enable	**Carry Flag**	**Overflow Flag**	**Negative Flag**	**Zero Flag**
IE	C	V	N	Z

Carry Flag: is a single bit that indicates that an arithmetic carry or borrow has been generated by an ALU operation.

Overflow Flag: used to indicate that the signed tow's0compliment result would not fit in the number of bits used in the operation.

Negative Flag: indicates that the result of the last operation is a negative.

Zero Flag: indicates that the result is all zeros.

D. Addressing Modes

Addressing modes are a specification for generating the address of an operand at runtime. Addressing modes in this implementation of the GBRAINS processor include:

Immediate Addressing

The operand is a 16-bit constant contained within the instruction.

`Example:` `addi $r1, $r2, 0xABCD`

Register Addressing

The operands are in a specified register.

`Example:` `add $r1. $r2, $r3`

Base-Indexed Addressing

The effective address is the sum of a register and an immediate value.

`Example:` `lw $r1, 4($r2)`

PC-Relative Addressing

The instruction address is the sum of the PC and a 16-bit constant contained within the instruction.

`Example:` `beq $r0, $r1, Label`

Indirect Addressing

The effective address is in a register.

`Example:` `jr $r31`

Pseudo-Direct Addressing

The instruction address is the 26-bit constant within the instruction concatenated with the upper 4 bits of the PC.

`Example:` `j Label`

E. Instruction Set and Binary Instruction Formats

The instruction set is a set of all the machine code that can be recognized and executed by the processor. Each instruction provides commands to the processor. Note that any instruction that is not specified below is classified as an ILLEGAL OP, which will cause a register data dump and end instruction execution.

At the core of the GBRAINS processor, there are four major classes of instructions:
R-Type: all operands and destination register are from the register file.
I-Type: Immediate types include a 16-bit immediate in the instruction.
J-Type: contain the opcode and 26-bit address field.
ER-Type: for the enhanced register the fields mirror those of the R-Type instruction. Except the SHAMNT is a FMT field, used to designate variations of the operations in the FUNCT field. All the Floating Point and Vector type operations in this revision of the CPU are this variation of R-Type.

1. R-Types

31 26	25 21	20 16	15 11	10 6	5 0
opcode 000000	rs	rt	rd	shamt	funct
6	5	5	5	5	6

R-Type Instructions: Main processor instruction that do not require a target address, immediate value, or branch displacement use an R-Type coding format. If bits [31:26] are equal to zero, then the instruction is an R-Type, otherwise it may be a I-Type or J-type.

R-Type Instruction Format:

```
    op rd, rs, rt
```

`op field[31:26]:` contains value zero. (see above diagram)

`rd field[25:21]:` Destination register.

`rs field[20:16]:` Source 1 register.

`rt field[15:11]:` Source 2 register.

`shamt field[10:6]:` Source 2 register.

The instruction is further classified depending on the value of **`funct field[5:0]:`** See next page for list of R-Type instructions sorted by `function` codes and a detailed explanation.

Table of R-Type Instructions

Mnemonic/ Instruction	Purpose	Function IR[5:0]
SLL	Logical Shift Left	000000
SRL	Logical Shift Right	000010
SRA	Shift Right Arithmetic	000011
JR	Jump Register	001000
BREAK	Breakpoint	001101
MFHI	Move from High Register	010000
MFLO	Move from Low Register	010010
MULT	Multiplication	011000
DIV	Division	011010
SETIE	Set Interrupt Enable	011111
ADD	ADD Signed	100000
ADDU	Add Unsigned	100001
SUB	Subtraction Signed	100010
SUBU	Subtraction Unsigned	100011
AND	Logical AND	100100
OR	Logical OR	100101
XOR	Bitwise Exclusive OR	100110
NOR	Bitwise Not OR	100111
SLT	Set Less Than	101010
SLTU	Set Less Than Unsigned	101011

ADD: Addition

31 26	25 21	20 16	15 11	10 6	5 0
opcode 000000	rs	rt	rd	shamt 00000	funct 100000
6	5	5	5	5	6

Format: `ADD rd, rs, rt`

Purpose: Add 32-bit integers.

Description: `rd = rs + rt`

A 32-bit word value in register *`rt`* is **ADDED** with a 32-bit value in register *`rs`*. The 32-bit result is stored into register *`rd`*.
Carry flag, Overflow flag, Negative flag, and Zero flag are set accordingly.

Operation:

```
{C,rd} = rs + rs;

V = (~rs[31] & ~rt[31]) & rd[31] |
     ( rs[31] &  rt[31]) & ~rd[31]};
```

Example:

ADD $r5, $r3, $r4		
$r3, $r4	$r5	Flags

`$r3 = 0x0000_020D`
`$r4 = 0xFFFF_FFE3`
	`$r5 = 0x0000_01F0`	`C=1, V=0, N=0, Z=0`

`$r3 = 0XFFFF_FFC9`
`$r4 = 0x0000_000D`
	`$r5 = 0xFFFF_FFD6`	`C=0, V=0, N=1, Z=0`

`$r3 = 0xFFFF_FF9C`
`$r4 = 0xFFFF_FF9D`
	`$r5 = 0xFFFF_FF39`	`C=1, V=0, N=1, Z=0`

ADDU: Add (Unsigned)

31 26	25 21	20 16	15 11	10 6	5 0
opcode 000000	rs	rt	rd	shamt 00000	funct 100001
6	5	5	5	5	6

Format: `ADDU rd, rs, rt`

Purpose: Add 32-bit integers.

Description: `rd = rs + rt`

A 32-bit word value in register *`rt`* is **ADDED** to the 32-bit value in register *`rs`*. The 32-bit result is stored into register *`rd`*.
Carry flag, Overflow flag, Negative flag, and Zero flag are set accordingly.
Note: if Carry is a one, so is Overflow.

Operation:

```
{C,rd} = rs + rs;

V = (~rs[31] & ~rt[31]) & rd[31] |
     ( rs[31] &  rt[31]) & ~rd[31]};
```

Example:

ADDU $r5, $r3, $r4		
$r3, $r4	$r5	Flags

`$r3 = 0x0000_020D`
`$r4 = 0xFFFF_FFE3`
	`$r5 = 0x0000_01F0`	`C=1, V=1, N=0, Z=0`

`$r3 = 0xFFFF_FFC9`
`$r4 = 0x0000_000D`
	`$r5 = 0xFFFF_FFD6`	`C=0, V=0, N=0, Z=0`

`$r3 = 0xFFFF_FF9C`
`$r4 = 0xFFFF_FF9D`
	`$r5 = 0xFFFF_FF39`	`C=1, V=1, N=0, Z=0`

AND: Bitwise AND

31 26	25 21	20 16	15 11	10 6	5 0
opcode 000000	rs	rt	rd	shamt 00000	funct 100100
6	5	5	5	5	6

Format: `AND rd, rs, rt`

Purpose: Executes a bitwise AND for the 32-bit registers.

Description: rd = rs **and** rt

A 32-bit word value in register *rt* is **AND'D** with a 32-bit value in register *rs*. The 32-bit result is stored into register *rd*.
Negative flag and Zero flag are set accordingly.

Operation: `rd = rs & rt;`

rs	rt	rd
0	0	0
0	1	0
1	0	0
1	1	1

Examples:

AND $r5, $r3, $r4		
$r3, $r4	$r5	Flags
$r3 = 0xF0F0_3C3C $r4 = 0xBF0F_F5F5	$r5 = 0xB000_3434	N=1, Z=0
$r3 = 0x0000_0025 $r4 = 0x0000_001D	$r5 = 0x0000_0005	N=0, Z=0

BREAK: Breakpoint

31 26	25 21	20 16	15 11	10 6	5 0
opcode 000000	rs 00000	Rt 00000	Rd 00000	shamt 00000	funct 001101
6	5	5	5	5	6

Format: `BREAK`

Purpose: Stops the program counter.

Description:

Stops the program counter to signal the end of the program. The break instruction will dump all data registers, data memory, and I/O memory when called.

DIV: Division

31 26	25 21	20 16	15 11	10 6	5 0
opcode 000000	rs	rt	rd	shamt 00000	funct 011010
6	5	5	5	5	6

Format: `DIV rs, rt`

Purpose: To divide 32-bit signed integers.

Description: `(HI,LO) = rs / rt`

A 32-bit word value in register *rs* is **DIVIDED** by a 32-bit value in register *rt.* This result in a 32-bit quotient and 32-bit remainder.
The 32-bit quotient is placed into a special register LO and the 32-bit remainder is placed into special register HI.
Negative flag and Zero flag are set accordingly.

Operation:
```
LO = rs / rt;
HI = rs % rt;
```

Examples:

DIV $r3, $r4		
$r3, $r4	HI, LO	Flags

$r3 = 0x0000_020D
$r4 = 0xFFFF_FFE3
	HI = 0X0000_0003
	LO = 0XFFFF_FFEE
		N=1, Z=0

$r3 = 0x0000_0025
$r4 = 0x0000_001D
	HI = 0X0000_0008
	LO = 0X0000_0001
		N=0, Z=0

JR: Jump Register

31 26	25 21	20 16	15 11	10 6	5 0
opcode 000000	rs	rt 00000	rd 00000	shamt 00000	funct 001000
6	5	5	5	5	6

Format: `JR rs`

Purpose: To jump to an instruction address in a register.

Description: PC = rs

Jumps to the effective address indicated by *rs*.

Operation:
```
PC = rs;
```

Examples:

JR $r3		
$r3	PC	Flags
$r3 = 0x0000_FFE4	PC = 0X0000_FFE4	

MFHI: Move from HI Register

31 26	25 21	20 16	15 11	10 6	5 0
opcode 000000	rs 00000	rt 00000	rd	shamt 00000	funct 001010
6	5	5	5	5	6

Format: `MFHI rd`

Purpose: To copy the special purpose register *`HI`* to register *`rd`*.

Description: `rs = HI`

The contents of the special register *`HI`* are loaded into register *`rd`*.

Operation:

```
rd = HI;
```

Examples:

MFHI $r3		
HI	$r3	Flags
`HI = 0x04C0_F3E1`	`$r3 = 0X04C0_F3E1`	

MFLO: Move from Lo Register

31 26	25 21	20 16	15 11	10 6	5 0
opcode 000000	rs 00000	rt 00000	rd	shamt 00000	funct 001010
6	5	5	5	5	6

Format: `MFLO rd`

Purpose: To copy the special purpose register *`LO`* to register *`rd`*.

Description: `rd = LO`

The contents of the special register *`LO`* are loaded into register *`rd`*.

Operation:

```
    rd = LO;
```

Examples:

MFHI $r3		
LO	$r3	Flags
LO = 0x04C0_F3E1	$r3 = 0X04C0_F3E1	

MULT: Multiplication

31 — 26	25 — 21	20 — 16	15 — 11	10 — 6	5 — 0
opcode 000000	rs	rt	rd	shamt 00000	funct 011000
6	5	5	5	5	6

Format: `MULT rs, rt`

Purpose: To multiply 32-bit signed integers.

Description: `(HI,LO) = rs * rt`

A 32-bit word value in register *rt* is **MULTIPLIED** by a 32-bit value in register *rs.* The result is a 64-bit value. The most significant 32-bit of the result is placed into a register HI and the least significant 32-bit of the remining result are placed in register LO.
Negative flag and Zero flag are set accordingly.

Operation:

```
{HI,LO} = rs * rt;
```

Examples:

MULT $r3, $r4		
$r3, $r4	HI, LO	Flags

`$r3 = 0x0000_020D`
`$r4 = 0xFFFF_FFE3`
	`HI = 0XFFFF_FFFF`
	`LO = 0Xffff_C487`
		`N=1, Z=0`

`$r3 = 0x0000_0025`
`$r4 = 0x0000_001d`
	`HI = 0X0000_0000`
	`LO = 0X0000_0431`
		`N=0, Z=0`

NOR: Bitwise Not OR

31 26	25 21	20 16	15 11	10 6	5 0
opcode 000000	rs	rt	rd	shamt 00000	funct 100111
6	5	5	5	5	6

Format: `NOR rd, rs, rt`

Purpose: Executes a bitwise NOR for the two 32-bit registers.

Description: `rd = rs` **`nor`** `rt`

A 32-bit word value in register *`rt`* is **NOR'D** with a 32-bit value in register *`rs.`* The 32-bit result is stored into register *`rd`*.
Negative flag and Zero flag are set accordingly

Operation: `rd = ~(rs | rt);`

rs	rt	rd
0	0	1
0	1	0
1	0	0
1	1	0

Examples:

NOR $r5, $r3, $r4		
$r3, $r4	$r5	Flags

`$r3 = 0xF0F0_3C3C`
`$r4 = 0xBF0F_F5F5`
	`$r5 = 0x0000_0202`	`N=0, Z=0`

`$r3 = 0x7000_C025`
`$r4 = 0x7001_DD54`
	`$r5 = 0x8FFE_228A`	`N=1, Z=0`

OR

31 26	25 21	20 16	15 11	10 6	5 0
opcode 000000	rs	rt	rd	shamt 00000	funct 100101
6	5	5	5	5	6

Format: `OR rd, rs, rt`

Purpose: Executes a bitwise OR for the two 32-bit registers.

Description: `rd = rs` **`or`** `rt`

A 32-bit word value in register *`rt`* is **OR'D** with a 32-bit value in register *`rs.`* The 32-bit result is stored into register *`rd`*.
Negative flag and Zero flag are set accordingly

Operation: `rd = rs | rt;`

rs	rt	rd
0	0	0
0	1	1
1	0	1
1	1	1

Examples:

OR $r5, $r3, $r4		
$r3, $r4	$r5	Flags

`$r3 = 0xF0F0_3C3C`
`$r4 = 0xBF0F_F5F5`
	`$r5 = 0xB000_3434`	`N=1, Z=0`

`$r3 = 0x0000_0025`
`$r4 = 0x0000_001D`
	`$r5 = 0x0000_003D`	`N=0, Z=0`

SETIE: Set Interrupt Enable

31 26	25 21	20 16	15 11	10 6	5 0
opcode 000000	rs 00000	rt 00000	rd 00000	shamt 00000	Funct 011111
6	5	5	5	5	6

Format: `SETIE`

Purpose: To enable interrupts.

Description:

Sets interrupt flag to 1 and allows the program to jump to the interrupt service routine.

Operation:

```
IE = 1'b1;
```

SLL: Shift Left Logical

31 26	25 21	20 16	15 11	10 6	5 0
opcode 000000	rs 00000	rt	rd	shamt	funct 000000
6	5	5	5	5	6

Format: `SLL rd, rt, shamt`

Purpose: Logical left shift a 32-bit word by a fixed number of bits.
Description: `rd = rt << shamt`

Register *rt* is shifted to the left *shamt* times with a zero fill. The result if stored in *rd.*

Note: Zeros are shifted in.

Carry flag, Negative flag and Zero flag are set accordingly.

Operation:
```
    {C,rd} = {rt[31], rt[30:0], 1'b0};
```

Examples:

SLL $r5, $r4, 1		
$r4, shamt	$r5	Flags
$r4 = 0xF0FF_F5F5 shamt = 1	$r5 = 0x7E1F_EBEA	C=1, N=0, Z=0

Binary Example:
```
     1111_0000_1111_1111  1111_0101_1111_0101
1 ◄— 1110_0001_1111_1111  1110_1011_1110_1010 ◄— 0

C=1, N=0, Z=0
```

SLT: Shift Left Logical (Signed)

31 26	25 21	20 16	15 11	10 6	5 0
opcode 000000	rs	rt	rd	shamt 00000	funct 101010
6	5	5	5	5	6

Format: `SLT rd, rs, rt`

Purpose: To record the result of a less-than comparison.

Description: `rd = (rs < rt)`

Compare the register contents of *`rs`* and *`rt`* as signed integers. Stores the Boolean result in *`rd.`* If *`(rs < rt),`* the result is 1 (true), otherwise the result is 0 (false).
Negative flag and Zero flag are set accordingly.

Operation:

```
    {V,C} = 2'bx;

        rd = (rs < rt);
```

Examples:

SLT $r5, $r3, $r4		
$r3, $r4	$r5	Flags

`$r3 = 0000_020D`
`$r4 = OxFFFF_FFE3`
	`$r5 = 0x0000_0000`	`N=0, Z=1`

`$r3 = 0xFFFF_FFC9`
`$r4 = 0x0000_000D`
	`$r5 = 0x0000_0001`	`N=0, Z=0`

SLTU: Set on Less Than (Unsigned)

31 26	25 21	20 16	15 11	10 6	5 0
opcode 000000	rs	rt	rd	shamt 00000	funct 101011
6	5	5	5	5	6

Format: `SLT rd, rs, rt`

Purpose: To record the result of a less-than comparison.

Description: `rd = (rs < rt)`

Compare the register contents of *`rs`* and *`rt`* as signed integers. Stores the Boolean result in *`rd.`* If *`(rs < rt)`*, the result is 1 (true), otherwise the result is 0 (false).
Negative flag and Zero flag are set accordingly.

Operation:
```
    {V,C} = 2'bx;

        rd = (rs < rt);
```

Examples:

SLTU $r5, $r3, $r4		
$r3, $r4	$r5	Flags

`$r3 = 0x0000_020D`
`$r4 = 0xFFFF_FFE3`
	`$r5 = 0x0000_0001`	`N=0, Z=0`

`$r3 = 0xFFFF_FFC9`
`$r4 = 0x0000_000D`
	`$r5 = 0x0000_0001`	`N=0, Z=1`

SRA: Shift Right Arithmetic

31 26	25 21	20 16	15 11	10 6	5 0
opcode 000000	rs 00000	rt	rd	shamt	funct 000011
6	5	5	5	5	6

Format: `SRA rd, rt, shamt`

Purpose: Arithmetic right shift a 32-bit word by a fixed number of bits.
Description: `rd = rt >> shamt`

Register *`rt`* is shifted to the right *`shamt`* times with a duplicated sign-bit *`(bit-31)`* fill. The result if stored in *`rd`*.
Carry flag, Negative flag and Zero flag are set accordingly.

Operation:

```
{C,rd} = {rt[0], rt[31], rt[31:1]};
```

Examples:

SRA $r5, $r4, 1		
$r4, shamt	$r5	Flags

`$r4   = 0xBF0F_F5F5`
`shamt = 1`
	`$r5 = 0xDF87_FAFA`	`C=1, N=1, Z=0`

Binary Example:

```
      1011_1111_0000_1111  1111_0101_1111_0101
1 ──► 1101_0000_1111_1111  1111_0101_1111_0101 ──►   1

C=1, N=1, Z=0
```

SRL: Shift Right Logical

31 26	25 21	20 16	15 11	10 6	5 0
opcode 000000	rs 00000	rt	rd	shamt	funct 000010
6	5	5	5	5	6

Format: `SRL rd, rt, shamt`

Purpose: Logical right shift a 32-bit word by a fixed number of bits.

Description: `rd = rt >> shamt`

Register *`rt`* is shifted to the right *`shamt`* times with a zero fill. The result if stored in *`rd`*.

Note: Zeros are shifted in.

Carry flag, Negative flag and Zero flag are set accordingly.

Operation:

```
{C,rd} = {rt[31], rt[30:0], 1’b0};
```

Examples:

SRL $r5, $r4, 1		
$r4, shamt	$r5	Flags

`$r4 = 0xBF0F_F5F5`
`shamt = 1`
	`$r5 = 0x5F87_FAFA`	`C=1, N=0, Z=0`

Binary Example:

```
      1011_0000_1111_1111  1111_0101_1111_0101
0 ──► 0111_1000_0111_1111  1111_1010_1111_1010 ──►  0

C=1, N=0, Z=0
```

SUB: Subtraction

31 26	25 21	20 16	15 11	10 6	5 0
opcode 000000	rs	rt	rd	shamt 00000	funct 100010
6	5	5	5	5	6

Format: `SUB rd, rs, rt`

Purpose: Subtract 32-bit integers.

Description: `rd = rs - rt`

A 32-bit word value in register *`rt`* is **SUB'D** with a 32-bit value *`rs`*.
The 32-bit result is stored into register *`rd`*.
Carry flag, Overflow flag, Negative flag, and Zero flag are set accordingly.

Operation:

```
{C,rd} = rs - rs;

V = (~rs[31] & ~rt[31]) & rd[31] |
     ( rs[31] &  rt[31]) & ~rd[31]};
```

Example:

SUB $r5, $r3, $r4		
$r3, $r4	$r5	Flags

`$r3 = 0x0000_020D`
`$r4 = 0xFFFF_FFE3`
	`$r5 = 0x0000_022A`	`C=1, V=0, N=0, Z=0`

`$r3 = 0xFFFF_FFC9`
`$r4 = 0x0000_000D`
	`$r5 = 0xFFFF_FFBC`	`C=0, V=0, N=1, Z=0`

`$r3 = 0xFFFF_FF9C`
`$r4 = 0xFFFF_FF9D`
	`$r5 = 0xFFFF_FFFF`	`C=1, V=0, N=1, Z=0`

SUBU: Subtract (Unsigned)

31 26	25 21	20 16	15 11	10 6	5 0
opcode 000000	rs	rt	rd	shamt 00000	funct 100011
6	5	5	5	5	6

Format: `SUBU rd, rs, rt`

Purpose: Add 32-bit integers.

Description: `rd = rs - rt`

A 32-bit word value in register *`rt`* is **SUBTRACTED** from the 32-bit value in register *`rs`*. The 32-bit result is stored into register *`rd`*.
Carry flag, Overflow flag, Negative flag, and Zero flag are set accordingly.
Note: if Carry is a one, so is Overflow.

Operation:

```
{C,rd} = rs + rs;

V = (~rs[31] & ~rt[31]) & rd[31] |
     ( rs[31] &  rt[31]) & ~rd[31]};
```

Example:

SUBU $r5, $r3, $r4		
$r3, $r4	$r5	Flags

`$r3 = 0x0000_020D`
`$r4 = 0xFFFF_FFE3`
	`$r5 = 0x0000_022A`	`C=1, V=1, N=0, Z=0`

`$r3 = 0xFFFF_FFC9`
`$r4 = 0x0000_000D`
	`$r5 = 0xFFFF_FFBC`	`C=0, V=0, N=0, Z=0`

`$r3 = 0xFFFF_FF9C`
`$r4 = 0xFFFF_FF9D`
	`$r5 = 0xFFFF_FFFF`	`C=1, V=1, N=0, Z=0`

XOR: Bitwise Exclusive OR

31 26	25 21	20 16	15 11	10 6	5 0
opcode 000000	rs	rt	rd	shamt 00000	funct 100110
6	5	5	5	5	6

Format: `XOR rd, rs, rt`

Purpose: Executes a bitwise an EXCLUSIVE OR for two 32-bit registers.
Description: `rd = rs` **`or`** `rt`

A 32-bit word value in register *`rt`* is **EXCLUSIVE OR'D** with a 32-bit value in register *`rs`*. The 32-bit result is stored into register *`rd`*.
Negative flag and Zero flag are set accordingly

Operation: `rd = rs ^ rt;`

rs	rt	rd
0	0	0
0	1	1
1	0	1
1	1	1

Examples:

XOR $r5, $r3, $r4		
$r3, $r4	$r5	Flags

`$r3 = 0xF0F0_3C3C`
`$r4 = 0xBF0F_F5F5`
	`$r5 = 0xB000_3434`	`N=1, Z=0`

`$r3 = 0x0000_0025`
`$r4 = 0x0000_001D`
	`$r5 = 0x0000_003D`	`N=0, Z=0`

2. I-Types

31 26	25 21	20 16	15 0
opcode	rs	rt	immediate
6	5	5	16

I-Type Instructions: Have a 16-bit immediate field that codes one of the following types of information.

I-Type Instruction Format:

```
    op rs, rt, immediate
```

`op field[31:26]:` is the mnemonic for the instruction.

`rs field[25:21]:` Source register.

`rt field[20:16]:` Destination registers.

`immediate field[15:0]:` is the 16-bit immediate value.

`IR[31:26]` ***`opcode`*** will be identified as an I-Type by the processor and then execute the instruction.

Table of I-Type Instructions

Mnemonic/ Instruction	Purpose	Function IR[31:26]
BEQ	Branch is Equal	000100
BNE	Branch if Not Equal	000101
BLEZ	Branch if Less Than or equal Zero	000110
BGTZ	Branch if Greater Than Zero	000111
ADDI	Add Immediate	001000
SLTI	Set Less than Immediate	001010
SLTIU	Set Less than Imm. Unsigned	001011
ANDI	And Immediate	001100
ORI	OR Immediate	001101
XORI	Exclusive OR Immediate	001110
LUI	Load Upper Immediate	001111
INPUT	Input	011100
OUTPUT	Output	011101
RETI	Return from Interrupt	011110
SW	Store Word	101011
LW	Load Word	100011

ADDI: Add Immediate

31 26	25 21	20 16	15 0
opcode 001000	rs	rt	immediate
6	5	5	16

Format: `ADDI rt, rs, immediate`

Purpose: Adds an immediate value to 32-bit integers.

Description: `rt = rs + immediate`

A 32-bit word value in register *`rt`* is **ADDED** with a 16-bit signed *`immediate`* value. The 32-bit result is stored into register *`rd`*. Carry flag, Overflow flag, Negative flag, and Zero flag are set accordingly.

Operation:

```
rt = rs + immediate;
```

Example:

ADDi $r5, $r3, Immediate		
$r3, Immediate	ADDI	Flags

`$r3       = 0x0000_020D`
`immediate = 0x0000_0003`
	`$r5 = 0x0000_03FD`	`C=0, V=0, N=0, Z=0`

`$r3       = 0XFFFF_FFC9`
`immediate = 0x0000_0F0D`
	`$r5 = 0x0000_0ED6`	`C=1, V=0, N=1, Z=0`

ANDI: And Immediate

31 26	25 21	20 16	15 0
opcode 001100	rs	rt	immediate
6	5	5	16

Format: `ANDI rt, rs, immediate`

Purpose: Bitwise ANDS a register *`rs`* and an *`immediate value`*. The result is stored in *`register rt.`*

Description: `rd = rs` **`AND`** `immediate`

A 32-bit word value in register *`rt`* is **AND’D** with a a 16-bit signed *`immediate`* value. The 32-bit result is stored into register *`rd`*. Negative flag and Zero flag are set accordingly

Operation:

```
rt = {rs & (16’h0, immediate[15:0]};
```

Examples:

ANDI $r5, $r4, 0xFAFA		
$r4, Immediate	$r5	Flags

`$r4       = 0xF0F0_3C3C`
`Immediate = 0x0000_FAFA`
	`$r5 = 0xF0F0_3434`	`N=0, Z=0`

BEQ: Branch on Equal

31 26	25 21	20 16	15 0
opcode 000100	rs	rt	offset
6	5	5	16

Format: `BEQ rs, rt, offset`

Purpose: Branches if the register *rs* and *rt* are equal.

Description: `if (rs == rt) then branch`

If the contents in the *rs* and *rt* are equal, branch to the effective target address.

Operation:

```
PC = (rs == rt) ? PC+{(14{offset[15]}},offset,2'b00} : PC;
```

Examples:

BEQ $r5, $r3, offset		
PC, $R5, $R3, OFFSET	PC	Flags

PC = 0x010A_0524
$r5 = 0x02C1_F001
$r3 = 0x02C1_F001
Offset = 0x0002
	PC = 0x010A_0526	

BGTZ: Branch on Greater Than Zero

31 26	25 21	20 16	15 0
opcode 000111	rs	rt 000000	offset
6	5	5	16

Format: `BGTZ rs, offset`

Purpose: Branches if the register `rs` is greater than `zero`

Description: `if (rs > 0) then branch`

If the contents in the `rs` is greater than `zero`, branch to the effective target address.

Operation:

```
PC = (rs > 0) ? PC+{(14{offset[15]}},offset,2’b00} : PC;
```

Examples:

BGTZ $r5, $r3, offset		
PC, $R5, $R3, OFFSET	PC	Flags

PC = 0x010A_0524
$r5 = 0x02C1_F001
$r3 = 0x0000_0000
Offset = 0x0002
	PC = 0x010A_0526	

BLEZ: Branch on Less Than or Equal to Zero

31 26	25 21	20 16	15 0
opcode 000110	rs	rt 000000	offset
6	5	5	16

Format: `BLEZ rs, offset`

Purpose: Branches if the register *rs* is less than or equal to *zero*

Description: `if (rs ≤ 0) then branch`

If the contents in the *rs* is less than or equal to zero, branch to the effective target address.

Operation:

```
PC = (rs <= 0) ? PC+{(14{offset[15]}},offset,2’b00} : PC;
```

Examples:

BLEZ $r5, $r3, offset		
PC, $R5, $R3, OFFSET	PC	Flags
PC = 0x010A_0524 $r5 = 0x02C1_F001 $r3 = 0x0000_0000 Offset = 0x0002	PC = 0x010A_0526	

BNE: Branch on Not Equal

31 26	25 21	20 16	15 0
opcode 000101	rs	rt	offset
6	5	5	16

Format: `BNE rs, rt, offset`

Purpose: Branches if the register *rs* is not equal to register *rt*

Description: `if (rs != 0) then branch`

If the contents in the *rs* is not equal to *rt*, branch to the effective target address.

Operation:

```
PC = (rs != rt) ? PC+{(14{offset[15]}},offset,2'b00} : PC;
If(rs != rt) PC = PC + offset
```

Examples:

BNE $r5, $r3, offset		
PC, $R5, $R3, OFFSET	PC	Flags
PC = 0x010A_0524 $r5 = 0x02C1_F001 $r3 = 0x0200_0341 Offset = 0x0002	PC = 0x010A_0526	

LUI: Load Upper Immediate

31 26	25 21	20 16	15 0
opcode 001111	rs	rt	immediate
6	5	5	16

Format: `LUI rt, immediate`

Purpose: The *`immediate`* value in shifted to the upper 16 bits of a register.

Description: `rt = (immediate << 16)`

The 16-bit *`immediate value`* is concatenated with 16 bits of zeros of a register and stored in register *`rt`*.
Negative flag and Zero flag are set accordingly.

Operation:

```
rt = {immediate[15:0], 16'h0};
```

Examples:

LUI $r5, 0xFAFA		
Immediate	$r5	Flags
Immediate = 0x0000_FAFA	$r5 = 0xFAFA_0000	N=1, Z=0

LW: Load Word

31 26	25 21		20 16	15 0
opcode 100011	rs		rt	offset
6	5		5	16

Format: `LW rt, offset(rs)`

Purpose: Load a word from memory.

Description: `rt = memory[rs+offset]`

The contents of the memory location specified by the effective address are stored in register *rt*. The 16-bit signed *offset* is sign-extended and added to the contents of *rs* to form the effective address.
Note: The least two significant bits of the *offset* must be zero.

Operation:

```
rt = memory[rs+offset];
```

Examples:

LW $r5, offset($r3)		
$r3, offset	$r5	Flags

$r3 = 0x0000_001B
offset = 0x0000_0005
Mem[32] = 0x0B10_4500
	$r5 = 0x0B10_4500	

ORI: OR Immediate

31 26	25 21	20 16	15 0
opcode 001101	rs	rt	immediate
6	5	5	16

Format: `ORI rt, rs, immediate`

Purpose: Executes a bitwise OR with an *`immediate`* value.

Description: `rt = rs` **OR** `immediate`

A 32-bit word value in register *`rt`* is **OR** with an *`immediate value`* and stored in *`rt.`*
Negative flag and Zero flag are set accordingly.

Operation:

```
rt = {rs ^ {}16'h0, immediate[15:0]};
```

Examples:

XORI $r5, $r3, 0xF5F5		
$r3, Immediate	$r5	Flags

`$r3       = 0xF0F0_3C3C`
`Immediate = 0x0000_F5F5`
	`$r5 = 0xF0F0_FDFD`	`N=1, Z=0`

SLTI: Set Less Than Immediate

31 26	25 21	20 16	15 0
opcode 001010	rs	rt	immediate
6	5	5	16

Format: `SLTI rt, rs, immediate`

Purpose: To record the result of a less-than comparison with an immediate value.

Description: `rt = (rs < immediate)`

Compares register `rs` and the *`immediate`*. Records the Boolean result in *`rt.`* If *`(rs < immediate)`*, the result is 1 (true), otherwise the result is 0 (false).
Negative flag and Zero flag are set accordingly.

Operation:

```
rd = (rs < immediate) ? 32'b1 : 32'b0;
```

Examples:

SLTI $r5, $r3, Immediate		
$r3, immediate	$r5	Flags

`$r3       = 0x0BCF_020D`
`immediate = 0x4000_0001`
	`$r5 = 0x0000_0000`	`N=0, Z=0`

`$r3       = 0xFFFF_FFC9`
`immediate = 0x0000_000D`
	`$r5 = 0x0000_0001`	`N=1, Z=0`

SLTIU: Set on Less Than Immediate (Unsigned)

31 26	25 21	20 16	15 0
opcode 001011	rs	rt	immediate
6	5	5	16

Format: `SLTIU rt, rs, immediate`

Purpose: Performs an unsigned less than comparison with an *`immediate value.`*

Description: `rd = (rs < immediate)`

Compare the contents of *`rs`* and *`rt`* as unsigned integers. Records the Boolean result in *`rt.`* If *`(rs < immediate)`*, the result is 1 (true), otherwise the result is 0 (false).
Negative flag and Zero flag are set accordingly.

Operation:
```
rt = (rs < {{16{immediate[15}}},immediate}} ? 32’b1 : 32’b0;
```

Examples:

SLTIU $r5, $r3, immediate		
$r3, immediate	$r5	Flags

`$r3       = 0x0BCF_020D`
`immediate = 0x4000_0001`
	`$r5 = 0x0000_0000`	`N=0, Z=1`

`$r3       = 0xFFFF_FFC9`
`immediate = 0x0000_000D`
	`$r5 = 0x0000_0001`	`N=0, Z=0`

SW: Store Word

31 26	25 21	20 16	15 0
opcode 101011	rs	rt	offset
6	5	5	16

Format: `SW rt, offset(rs)`

Purpose: Store a word from memory.

Description: `memory[rs+offset] = rt`

Register *`rt`* is stored in memory at the location specified by the effective address. The 16-bit signed *`offset`* is sign-extended and added to the contents of *`rs`* to form the effective address.
Note: The least two significant bits of the *`offset`* must be zero.

Operation: `memory[rs+offset] = rt;`

Examples:

SW $r5, 2($r3)		
$r3, offset	$r5	Flags

`$r3 = 0x0000_001B`
`offset = 0x0000_0005`
`Mem[32] = 0x0B10_4500`
	`$r5 = 0x0B10_4500`	

XORI: Exclusive OR Immediate

31 26	25 21	20 16	15 0
opcode 001110	rs	rt	Immediate
6	5	5	16

Format: `XORI rt, rs, immediate`

Purpose: Executes an exclusive OR with an *`immediate`* value.

Description: `rt = rs` **`XOR`** `immediate`

A bitwise **EXCLUSIVE OR** is executed on the registers *`rs`* and *`immediate value`* and stored in *`rt`*.
Negative flag and Zero flag are set accordingly.

Operation:

```
rt = {rs ^ {}16’h0, immediate[15:0]};
```

Examples:

XORI $r5, $r3, 0xF5F5		
$r3, Immediate	$r5	Flags

`$r3       = 0xF0F0_3C3C`
`Immediate = 0x0000_F5F5`
	`$r5 = 0xF0F0_C9C9`	`N=1, Z=0`

3. J-Type Instructions

31 26	25 0
opcode	target address
6	26

J-Type Instructions: The 6-bit opcode corresponds to a jump instruction is followed by 26-bits that calculate the target address and will be loaded into Program Counter (PC).

J-Type Instruction Format:

```
     Op target
```

`op:` is the mnemonic for the instruction

`jump address:` is the target address that will be jumped to.

Table of J-Type Instructions

Mnemonic/ Instruction	**Purpose**	**Function IR[31:26]**
J	Jump	000010
JAL	Jump and Link	000011

J: Jump

31 26	25 0
opcode 000010	target address
6	26

Format: `J target`

Purpose: To jump to another address

Description: `PC = target address`

Note: The four most significant bits of the *`offset`* are removed and least two significant bits of the *`offset`* must be zero.

Operation:

```
PC = {PC[31:28], target, 2'b0};
```

Examples:

J 0x0060_AEFC		
Target address	PC	Flags
Target = 0x0060_AEFC	PC = 0x0818_2BBF	

Binary Example:

```
J 0x0060_AEFC = 000010 | 0000_0000_0110_0000 1010_1110_1111_1100
```

~~0000~~_0000_0110_0000 1010_1110_1111_11~~00~~

```
                         0000_1000_0001_1000  0010_1011_1011_1111
PC = 0x0812_2BBF
```

JAL: Jump and Link

31 26	25 0
opcode 000011	target address
6	26

Format: `JAL target`

Purpose: To jump to another address

Description: `PC = target address`

Similar to the jump instruction, except that it stores the address of the next instruction (the one immediately after the jump) in the return address ($ra) register. This allows a subroutine to return to the main body routine after completion.

Note: The four most significant bits of the *`offset`* are removed and the least two significant bits of the *`offset`* must be zero.

Operation:
```
$ra = PC + 4;
PC = {PC[31:28], target, 2’b0};
```

Examples:

JAL 0x0060_AEFC		
Target address	PC	Flags
Target = 0x008F_2648 $ra = PC + 4	PC = 0x0C23_C992	

4. Enhanced Instructions

Enhanced instructions that are derivative of R, J and I-Type instructions. Not included as part of the MIPS instruction set.

Enhanced instructions that are

Table of Enhanced Instructions

Mnemonic/ Instruction	Purpose	Function IR[31:26]
INPUT	Input	011100
OUTPUT	Output	011101
RETI	Return from Interrupt	011110

INPUT

31 26	25 21	20 16	15 0
opcode 011100	rs	rt	offset
6	5	5	16

Format: `INPUT rt, offset(rs)`

Purpose: To load a word from memory.

Description: `rt = memory[rs+offset]`

The 16-bit signed *`offset`* is and added to the register *`rs`* to form the effective address. The effective address is than loaded into register *`rt`*.

Programming Note:

Accesses memory from external I/O module.
Similar to a Load Word instruction.

Examples:

INPUT $r5, 2($r3)		
$r3, offset	$r5	Flags

`$r3 = 0x0000_001B`
`offset = 0x0000_0005`
`Mem[32] = 0x0B10_4500`
	`$r5 = 0x0B10_4500`	

Example sequence:

```
$r3 = 0x0000_001B;

     02 + 0x1B = 32

Mem[32] = 0x0B10_4500;
```

OUTPUT

31 26	25 21	20 16	15 0
opcode 011101	rs	rt	offset
6	5	5	16

Format: `OUTPUT rt, offset(rs)`

Purpose: To store a word in memory.

Description: `memory[rs+offset] = rt`

The 16-bit signed *`offset`* is sign-extended and added to the register *`rs`* to form the effective address. Register *`rt`* is store in external memory at the location specified by the effective address .

Programming Note:

Accesses memory from external I/O module.
Similar to a Load Word instruction.

Examples:

OUTPUT $r5, 2($r3)		
$r5, offset	$r3	Flags

`$r5    = 0x05D3_05Fd2`
`offset = 0x0000_0005`
`$r3    = 0x0000_001B`
	`Mem[32] = 05D3_05Fd2`	

Example sequence:

```
$r3 = 0x0000_001B;

     02 + 0x1B = 32

Mem[32] = 05D3_05Fd2;
```

RETI: Return from Interrupt

31 26	25 21	20 16	15 0
opcode 011110	rs	rt	offset
6	5	5	16

Format: `RETI`

Purpose: Return from interrupt service routine after it returns the Program Counter and Flag registers to their values before the ISR was executed.

Description:

```
PC    = memory[$sp]
Flags = memory[$sp + 4]
```

The value pointed to by the stack pointer is loaded/popped into the program counter. This will return the status Flags and the PC. Then saves the new $sp value into $sp.

Examples:

RETI $r5, 2($r3)		
Stack Pointer	Address	Memory
(TOS)	0x3F4	Flags
	0x3F8	PC
	0X3FC	SP

5. Floating Point Double Precision-Types

31 26	25 21	20 16	15 11	10 6	5 0
opcode	fs	ft	fd	fmt	funct
6	5	5	5	5	6

FPD-Type Instructions: Main processor instruction that do not require a target address, immediate value, or branch displacement use an R-Type coding format. If bits [31:26] are equal to zero, then the instruction is an R-Type or FP-Type, otherwise it may be a I-Type or J-type.

FPD-Type Instruction Format:

```
    Op fs, ft, fd, fmt, funct
```

`op field[31:26]:` contains value 1F.

`fs field[25:21]:` Source register.

`ft field[20:16]:` Source register.

`fd field[15:11]:` Destination register.

`fmt field[10:6] :` future choice for single or double precision.

`funct[5:0] :` Choice of function.

The instruction is further classified depending on the value of **`funct field[5:0]:`** See next page for list of FPD-Type instructions sorted by `function` codes and a detailed explanation.

$bitstoreal: is used for both input S and T. This function converts the bit value to a real number.

Table of FP-R-Type Instructions

Mnemonic/ Instruction	Purpose	Function IR[5:0]
MVFR	Move from Int. reg to FP reg	000000
FMULT	FP Multiplication	000001
FDIV	FP Division	000010
FADD	FP Addition	000011
FSUB	FP Subtraction	000100
FZERO	Zeros	000101

MVFR: Move To Floating Point Register

31 26	25 21	20 16	15 11	10 6	5 0
opcode 011111	fs	ft	fd	fmt	funct 000000
6	5	5	5	5	6

Format: `MVFR fd, fs, ft`

Purpose: Store a 64-bit value into Floating Point Register.

Description: `F[fd] = {R[fs],R[ft]}`

A 32-bit value in register `rs` is concatenated with a 32-bit value in register *rt* and is placed as a 64-bit value into `rd`.

Operation:

```
fpY = {rS,rT};
```

Examples:

MVFR $f4, $f2, $f3,	
$f2, $f3	$f4
\$r2 = 12345678 \$r3 = DDDDDDDD	\$r4 = 12345678_DDDDDDDD
\$r2 = FFFFFFFF \$r3 = ABCDEF12	\$r4 = FFFFFFFF_ABCDEF12

FMULT: FP Multiply

31 26	25 21	20 16	15 11	10 6	5 0
opcode 011111	fs	ft	fd	fmt	funct 000001
6	5	5	5	5	6

Format: `FMULT rd, rs, rt`

Purpose: To multiply 64-bit integers.

Description: `F[fd] = F[fs] * F[ft]`

A 64-bit word value in register *`ft`* is **MULTIPLIED** by a 64-bit value in register *`fs.`* The result is a 64-bit value stored in *fd*.

Operation:

```
        fpY = fpS * fpT;
```

Examples:

FMULT $f7 $f3, $f4	
$3, $f4	$f7

`$r3 = 3.141593`
`$r4 = 2.000000`
	`$r7 = 6.283185`

`$r3 = 5.123395`
`$r4 = 8.452956`
	`$r7 = 43.307832`

FDIV: FP Divide

31 26	25 21	20 16	15 11	10 6	5 0
opcode 011111	fs	ft	fd	fmt	funct 000010
6	5	5	5	5	6

Format: FDIV `rd, rs, rt`

Purpose: To divide 64-bit integers.

Description: `F[fd] = F[fs] / F[ft]`

A 64-bit word value in register *`fs`* is **DIVIDED** by a 64-bit value in register *`ft.`* This results in a 64-bit quotient.
The 64-bit quotient is placed in register *fd*.

Operation:

```
        fpY = fpS / fpT;
```

Examples:

FDIV $f7 $f3, $f4	
$f3, $f4	$f7

`$r3 = 6.283185`
`$r4 = 3.141593`
	`$r7 = 2.000000`

`$r3 = 43.307832`
`$r4 = 5.123395`
	`$r7 = 8.452956`

FADD: FP Add

31 26	25 21	20 16	15 11	10 6	5 0
opcode 011111	fs	ft	fd	fmt	funct 000011
6	5	5	5	5	6

Format: `FADD fd, fs, ft`

Purpose: Add 64-bit integers.

Description: F[f`d] = F[fs] + F[ft]`

A 64-bit word value in register *ft* is **ADDED** with a 64-bit value in register *fs*. The 64-bit result is stored into register *fd*.

Operation:

```
        fpY = fpS + fpT;
```

Example:

FADD $f5, $f3, $f4	
$f3, $f4	$f5
$r3 = 6.283185 $r4 = 3.141593	$r5 = 9.424778
$r3 = 8.452956 $r4 = 5.123395	$r5 = 13.576351

FSUB: FP Subtract

31 26	25 21	20 16	15 11	10 6	5 0
opcode 011111	fs	ft	fd	fmt	funct 000100
6	5	5	5	5	6

Format: FSUB `fd, fs, ft`

Purpose: Sub 64-bit real.

Description: `F[fd] = F[fs] - F[ft];`

A 64-bit word value in register *ft* is **SUBTRACTED** with a 64-bit value in register *fs*. The 64-bit result is stored into register *fd*.

Operation:

```
fpY = fpS - fpT;
```

Example:

FSUB $f5, $f3 $f4	
$f3 $f4	**$f5**
$r3 = 2.000000 $r4 = 3.141593	$r5 = -1.141593
$r3 = 8.452956 $r4 = 5.123395	$r5 = 3.329561

FZERO: FP Zero

31 26	25 21	20 16	15 11	10 6	5 0
opcode 011111	fs	ft	fd	fmt	funct 000101
6	5	5	5	5	6

Format: ZERO `fd, fs, ft`

Purpose: Load 64-bit real of 0.0.

Description: f`d = 0.0`

A 64-bit word value 0.0 is **Load** into register *fd*. Regardless of fs and ft input.

Operation:

```
F[fd] = 0.0;
```

Example:

FZERO $f5, $f3 $f4	
	$f5
0.00	$r5 = 0.000000

6. Vector SIMD: Single Instruction Multiple Data Instructions

31 26	25 21	20 16	15 11	10 6	5 0
opcode 011111	vs	vt	vd	fmt	funct 000100
6	5	5	5	5	6

SIMD-Type Instructions: Have two or more values used as operands.

Vector-Type Instruction Format:

```
Ekey_op(0x1F) vs, vt, vd, fmt(5'b0), funct
```

op field[31:26]: is the mnemonic for the Enhanced Instruction.

rs field[25:21]: Source register.

rt field[20:16]: Source register.

rd field[15:11]: Destination Register.

Fmt field [6:10]: for setting packed size

IR[5:0] ***funct*** will be identify the ER type instruction to execute

Table of SIMD-Type Instructions

Mnemonic/ Instruction	Purpose	Function IR[5:0]
MVVR	Move to Vector register	000110
VADDS	Vector Add(Saturated,8b)	000111
VMULADD	Vector Multiply Add	001000
VANDEI	Vector logical and(8b)	001001
VCEQ	Vector compare if equal(8b)	001010
VCLT	Vector compare if less than(8b)	001011

MVVR: Move to Vector Register

31 26	25 21	20 16	15 11	10 6	5 0
opcode 011111	vs	vt	vd	fmt	funct 000110
6	5	5	5	5	6

Format: `MVVR vd, vs, vt`

Purpose: Stores concatenated rs and rt values from integer registers into the rd addressed vector register.

Description: `V[rd[63:0]] = {IntR[rs[31:0]],IntR[rt[31:0]]};`

:

The 32-bit value in *rs* in the integer register file is concatenated with the 32-bit value in *rt* in the integer register file and is stored in the 64-bit vector register at *rd*.

VADDS: Vector Add Saturated

31 26	25 21	20 16	15 11	10 6	5 0
opcode 011111	vs	vt	vd	fmt	funct 000111
6	5	5	5	5	6

Format: `VADDS vd, vs, vt`

Purpose: Adds vector value at rs with vector value at rt which can become saturated.

Description:

```
{carry[0],Y[7:0]} = V[rs[7:0]] + V[rt[7:0]];
                          .
                          .
{carry[7],Y[63:56]} = V[rs[63:56]] + V[rt[63:56]];

V[rd[7:0]] = carry[0]?8'hFF:Y[7:0];
                          .
                          .
V[rd[63:56]] = carry[7]?8'hFF:Y[63:56];
```

The 64-bit value in *rs* in the vector register file is added to the 64-bit value in *rt* in the vector register file byte-wise. If any 8-bit result saturates, the 8-bit value for that byte is kept at saturation point, 8'hFF, and is stored in the corresponding byte in the vector register at *rd*.

Operation:

```
{carry,vY} = vS + vT;
vD = carry?(8'hFF):vY;
```

Example:

VADDS $r5, $r3, $r4	
$r3, $r4	$r5

`$r3 = EE11EE11_FF11FF11`
`$r4 = 00FF00FF_00EE00EE`
	`$r5 = EEFFEEFF_FFFFFFFF`

`$r3 = A5A5A5A5_B4B4B4B4`
`$r4 = 5A5A5A5A_15151515`
	`$r5 = FFFFFFFF_C9C9C9C9`

VMULADD: Vector Multiply and Add

31 26	25 21	20 16	15 11	10 6	5 0
opcode 011111	vs	vt	vd	fmt	funct 001000
6	5	5	5	5	6

Format: `VMULADD vd, vs, vt`

Purpose: Multiplies values at *rs* and *rt* then adds the value at *rd* and stores them in *rd*.

Description:
```
V[rd[31:0]] = V[rs[31:0]] * V[rt[31:0]] + V[rd[31:0]];
                          .
                          .
V[rd[63:32]] = V[rs[63:32]] * V[rt[63:32] + V[rd[63:32]];
```

The 64-bit value in *rs* in the vector register file is multiplied the 64-bit value in *rt* in the vector register file and then added to the 64-bit value in rd in a byte-wise fashion. The results are stored in the corresponding byte in the vector register at *rd*.

Operation:
```
vYhi = vShi * vThi + vD;
vYlo = vSlo * vTlo + vD;
```

Example:

VMULADD $r5, $r3, $r4	
$r3, $r4, $r5	$r5
$r3 = 00000001_00000002 $r4 = 00000000_00000004 $r5 = 00000000_00000008	$r5 = 00000000_00000010
$r3 = 00110011_00450456 $r4 = 00000000_00000008 $r5 = 22222222_22222222	$r5 = 22222222_244A44D2

VANDEI: Vector AND Eight Integers

31 26	25 21	20 16	15 11	10 6	5 0
opcode 011111	vs	vt	vd	fmt	funct 001001
6	5	5	5	5	6

Format: `VANDEI vd, vs, vt`

Purpose: Bitwise "and"s the values at *rs* and *rt* and stores them in *rd*.

Description:

```
V[rd[7:0]] = V[rs[7:0]] & V[rt[7:0]];
                  .
                  .
V[rd[63:56]] = V[rs[63:56]] & V[rt[63:56];
```

The 64-bit value in *rs* in the vector register file is "and"ed the 64-bit value in *rt* in the vector register file in a byte-wise fashion. The results are stored in the corresponding byte in the vector register at *rd*.

Operation:

```
vY = vS & vT;
```

Example:

VANDEI $r5, $r3, $r4	
$r3, $r4	$r5
$r3 = F6F6F6F6_F6F6F6F6 $r4 = FFFFFFFF_FFFFFFFF	$r5 = F6F6F6F6_F6F6F6F6
$r3 = 12345678_23456789 $r4 = 12345678_87654321	$r5 = 12345678_23454321

VCEQ: Vector Compare if Equal

31 26	25 21	20 16	15 11	10 6	5 0
opcode 011111	vs	vt	vd	fmt	funct 001010
6	5	5	5	5	6

Format: `VCEQ vd, vs, vt`

Purpose: Compares vector value at rs with vector value at rt and assigns rd with ones if they are equal and zeros if they are not.

Description:

```
V[vd[7:0]] = V[vs[7:0]] == V[vt[7:0]]?8'hFF:8'h0;
                   .
                   .
V[vd[63:56]]=(V[vs[63:56]] == V[vt[63:56]])?8'hFF:8'h0;
```

The 64-bit value in *rs* in the vector register file is compared to the 64-bit value in *rt* in the vector register file byte-wise. If the 8-bits are equal, the resulting value is set to ones, 8'hFF, and is stored in the corresponding byte in the vector register at *rd*. If not, the resulting value s set to zeros and is stored in the vector register at *rd*.

Operation:

```
vY = (vS == vT)?8'hFF:8'h0;
```

Example:

VCEQ $r5, $r3, $r4	
$r3, $r4	$r5

`$r3 = 12345678_98765432`
`$r4 = 02345078_12345678`
	`$r5 = 00FF00FF_00000000`

`$r3 = FFFFFFFF_FFFFFFFF`
`$r4 = FF99FF99_56FF56FF`
	`$r5 = FF00FF00_00FF00FF`

VCLT: Vector Compare if Less Than

31 26	25 21	20 16	15 11	10 6	5 0
opcode 011111	vs	vt	vd	fmt	funct 001011
6	5	5	5	5	6

Format: `VCLT vd, vs, vt`

Purpose: Compares vector value at rs with vector value at rt and assigns rd with ones if rs is less than rt and zeros if they are not.

Description:
```
V[vd[7:0]] = V[vs[7:0]] < V[vt[7:0]]?8'hFF:8'h0;
                    .
                    .
V[vd[63:56]] = V[vs[63:56]] < V[vt[63:56]]?8'hFF:8'h0;
```

The 64-bit value in *rs* in the vector register file is compared to the 64-bit value in *rt* in the vector register file byte-wise. If the 8-bits *rs* are less than the 8-bits *rt*, the resulting value is set to ones, 8'hFF, and is stored in the corresponding byte in the vector register at *rd*. If not, the resulting value is set to zeros and is stored in the vector register at *rd*.

Operation:
```
vY = (vS < vT)?8'hFF:8'h0;
```

Example:

VCLT $r5, $r3, $r4	
$r3, $r4	$r5
$r3 = 12345678_ABCDEF98 $r4 = 87654321_FFFFFFFF	$r5 = FFFF0000_FFFFFFFF
$r3 = 87654321_98765432 $r4 = 00990099_99009900	$r5 = 00FF00FF_FF00FF00

III. Verilog Implementation/Design/Verification

A. Source Code Top Level

CPU_Test

```
`timescale 1ns / 1ps
/*********************************************************************************
* Author(s): Brian Ortiz
*            Bryan Linares
*            Grace Daliwan
* Filename: CPU_Test.v
* Date:     Nov. 27, 2018
* Project:  CECS 440 Senior Project 'GBRAINS'
* Version:  1.0
*
* Notes:    GBRAINS ENHANCED MIPS CPU Testbench. Instantiates the MIPS CPU,
*           I/O Memory Module, and Data Memory modules. and initializes
*           the IO, Instruction and Data memories. Reset is asserted and deasserted
*           and the MCU state machine runs without other input other than the
*           generated interrupt.
*********************************************************************************/
module CPU_Test;

   reg  clk,   reset;

   wire intr,  int_ack;          //Intr out from IO to CPU, Int_ack from CPU to IO
   wire dm_cs, dm_wr, dm_rd;     //Data memory access controls, IO memory controls
   wire io_cs, io_wr, io_rd;
   wire [31:0] D_OUT, DY;        //Data_Out and memory data value in
   wire [31:0] ALU_OUT;          //IDP outputs, ALU_Out is computed data
                                 //used as Address.

   //Instantiate the CPU
   CPU cpu (
    .clk(clk), .reset(reset), .intr(intr),      //inputs, interrupt from io
    .int_ack(int_ack),                          //outputs
    .dm_cs(dm_cs),     .dm_wr(dm_wr), .dm_rd(dm_rd),
    .io_cs(io_cs),     .io_wr(io_wr), .io_rd(io_rd),
    .ALU_OUT(ALU_OUT), .D_OUT(D_OUT), .DY(DY));

   //Instantiate the Data Memory
   DATA_MEMORY dm (
    .clk(clk), .dm_cs(dm_cs), .dm_wr(dm_wr), .dm_rd(dm_rd),  //inputs
    .Address(ALU_OUT[11:0]),  .D_in(D_OUT),                  //outputs
    .D_Out(DY) );

   //Instantiate the I/O Memory Space with Interrupt
   IO_Module io (
    .clock(clk),
    .int_ack(int_ack),       .io_wr(io_wr),
    .io_cs(io_cs),           .io_rd(io_rd),
    .Address(ALU_OUT[11:0]), .IO_in(D_OUT), //inputs
    .intr(intr),.IO_out(DY));               //outputs

   //create a 10ps clock
    always #5 clk = ~clk;

  ////////////////////////////////////////////////////////////////////////////////
  //Welcome to the enhancements testbench for the 'GBRAINS' CPU!
  //
  // The project is setup and ready to run the custom Enhanced Instruction Memory
```

```
   // Module. The procedure includes loading values into the Enhanced register files
   // in their respective datapaths, operating on the values using the implemented
   // operations and ending on a break statement which dumps the final status
   // of the register files in all three data paths, with some conversions done
   // on the floating point values for easier verification.
   //
   // Make sure that iSim 'runs all' the simulation! Enjoy!
   ////////////////////////////////////////////////////////////////////////////////////

   initial begin
       $display("             C E C S 4 4 0   S E N I O R P R O J E C T   GBRAINS");
       $timeformat(-9, 1, " ps", 9);
       clk = 1'b0;
       @(negedge clk)
       reset = 1'b1;
       @(negedge clk)
       reset = 1'b0;

       $readmemh("dMemEN1_Fa18.dat", dm.memory);
       $readmemh("iMemEN1_Fa18_commented.dat", cpu.iu.IMemReg.memory);
       $readmemh("dMemEN1_Fa18.dat", io.memory);

   end

endmodule
```

B. Source Code

CPU

```
`timescale 1ns / 1ps
/*******************************************************************************
* Author(s): Brian Ortiz
*            Bryan Linares
*            Grace Daliwan
* Filename: CPU.v
* Date:     Nov. 27, 2018
* Project:  CECS 440 Senior Project
* Version:  1.0
*
* Notes:    MIPS CPU module. Instantiates the MIPS Control Unit,
*           Instruction Unit and Datapath modules.
*           the MCU state machine runs without other input other than the
*           generated interrupt.
*******************************************************************************/
module CPU(clk, reset, intr, DY,
           int_ack, dm_cs, dm_wr, dm_rd, io_cs, io_wr, io_rd, ALU_OUT, D_OUT);

    // Inputs from Test module, io interrupt
    input clk, reset, intr;
    // DataMem/IO data input
   input [31:0] DY;

    // output to acknowledge io interrupt
   output int_ack;
    // DataPath outputs, also Address input to memories
   output [31:0] ALU_OUT;
    // Data_Out, muxed depending on data type set
   output [31:0] D_OUT;
    // data memory control
    output dm_cs, dm_wr, dm_rd;
    // io module control
   output io_cs, io_wr, io_rd;

   // MCU Outputs
    //Current Instruction Register value, the current instruction
    wire[31:0] IR;
    //carry, negative, zero, and overflow arithmetic flags
    wire c,n,z,v;
    //PC_sel,chooses input to pc, calculated branch or jump, D_sel
    wire [1:0] pc_sel, D_Sel;
    //PC load, PC_inc, and S_Sel controls (S_Sel, sets $29 on idp regfile in)
    wire pc_ld, pc_inc, S_Sel;
    //IO module memory chip select, write, read,and instr. reg load control
    wire im_cs, im_wr, im_rd, ir_ld;
    //IDP Regfile data write enable, Immediate into ALU, and HILO register loads
    wire D_En, T_Sel, HILO_ld;
    //DA_Sel, regfile Destination: D_Address (default), $29($sp) and $31($ra)
    wire [1:0] DA_Sel;
    //ALU_Out select, changes source of Data_Out in IDP
    wire [2:0] Y_Sel;
    //Function Select for integer ALU, flags input and output from MCU
    //flags register
    wire [4:0] FS, flags, flagsin;
    //Program Counter out
    wire [31:0] PC_out;
    //Sign Extended 16 bit immediate
    wire [31:0] SE_16;
    //integer, floating point, and vector data outs
   wire [31:0] iD_OUT, fD_OUT, vD_OUT;

    //Floating point and Vector SIMD Function Select
   wire [4:0] fpFS, vpFS;
    // Floating point Datapath Controls
   wire fD_En,fT_Sel,fDA_Sel,fDIN_Sel,fDOut_Sel,fY_Sel;
    //Vector Datapath controls
   wire vD_En,vT_Sel,vDA_Sel,vDIN_Sel,vDOut_Sel,vY_Sel;
    // {RS,RT} 64 bit concatenation lead to T input of Enhanced ALUs, for
    //immediate loads
   wire [63:0] LONG_OUT;
    //Type Select chooses alternate datapath output to external memories
   wire Type_Sel;

   //Instantiate the Control Unit
   MCU mcu (
   .sys_clk(clk),    .reset(reset),    .intr(intr),      .FLAGSIN(flagsin),
   .n(n),            .z(z),            .v(v),             .c(c),           //inputs
   .IR(IR),          .D_Sel(D_Sel),    .S_Sel(S_Sel),    .FLAGS(flags),  //outputs
   .pc_sel(pc_sel), .pc_ld(pc_ld),     .pc_inc(pc_inc), .int_ack(int_ack),
   .im_cs(im_cs),    .im_wr(im_wr),    .im_rd(im_rd),    .ir_ld(ir_ld),
   .D_En(D_En),      .DA_Sel(DA_Sel), .T_Sel(T_Sel),     .HILO_ld(HILO_ld),
   .dm_cs(dm_cs),    .dm_wr(dm_wr),    .dm_rd(dm_rd),
   .io_cs(io_cs),    .io_wr(io_wr),    .io_rd(io_rd),
   .Y_Sel(Y_Sel),    .FS(FS),
   //MCU Floating Point control signals
   .fD_En(fD_En),        .fpFS(fpFS),          .fT_Sel(fT_Sel), .fDA_Sel(fDA_Sel),
   .fDIN_Sel(fDIN_Sel), .fDOut_Sel(fDOut_Sel), .fY_Sel(fY_Sel),
   //MCU SIMD Vector control signals
   .vD_En(vD_En),    .vpFS(vpFS), .vT_Sel(vT_Sel), .vDA_Sel(vDA_Sel),
   .vDOut_Sel(vDOut_Sel),         .vY_Sel(vY_Sel), .vDIN_Sel(vDIN_Sel),
   .Type_Sel(Type_Sel));

//Instantiate the Instruction Unit
   INSTRUCTION_UNIT iu (
    .CLK(clk),        .RESET(reset),
    .im_cs(im_cs),    .im_wr(im_wr),     .im_rd(im_rd),
    .pc_ld(pc_ld),    .pc_inc(pc_inc),   .ir_ld(ir_ld),
    .PC_in(ALU_OUT), .pc_sel(pc_sel),                         //ins
    .PC_out(PC_out), .IR_out(IR),        .SE_16(SE_16));     //outs

   //Instantiate the Datapath Modules.
    INTEGER_DATAPATH idp (
    .CLK(clk),           .RESET(reset),
    .FS(FS),             .HILO_ld(HILO_ld),    .FLAGS(flags),       .S_Sel(S_Sel),
    .D_Sel(D_Sel),       .D_En(D_En),          .DY(DY),             .Y_Sel(Y_Sel),
    .D_Addr(IR[15:11]),.S_Addr(IR[25:21]),.T_Addr(IR[20:16]), .SHAMT(IR[10:6]),
    .DT(SE_16),          .T_Sel(T_Sel),        .PC_in(PC_out),    .DA_sel(DA_Sel),
    .C(c),               .V(v),                .N(n),               .Z(z),
    .ALU_OUT(ALU_OUT), .D_OUT(iD_OUT),.FLAGS_OUT(flagsin),.LONG_OUT(LONG_OUT));

    //Floating Point Datapath
    FLOATINGPOINT_DATAPATH fdp (
    .CLK(clk),           .RESET(reset), //inputs
    .D_EN(fD_En),        .FS(fpFS),
    .D_Addr(IR[15:11]), .S_Addr(IR[25:21]), .T_Addr(IR[20:16]), .FMT(IR[10:6]),
     //fields
    .DT(LONG_OUT), .DY(DY),    .T_Sel(fT_Sel), .DIN_Sel(fDIN_Sel),
    .DOut_Sel(fDOut_Sel),      .Y_Sel(fY_Sel),
    .D_OUT(fD_OUT));

    //SIMD Vector Operations Datapath
    VECTOR_DATAPATH vdp (
    .CLK(clk),          .RESET(reset), //inputs
    .D_EN(vD_En),       .FS(vpFS),
    .D_Addr(IR[15:11]), .S_Addr(IR[25:21]), .T_Addr(IR[20:16]), .FMT(IR[10:6]),
     //fields
    .DT(LONG_OUT), .DY(DY), .T_Sel(vT_Sel), .DIN_Sel(vDIN_Sel),
    .DOut_Sel(vDOut_Sel),   .Y_Sel(vY_Sel),
    .D_OUT(vD_OUT));

    //Data Type Mux
    assign D_OUT = (Type_Sel == 1'b1)? fD_OUT:
                   (Type_Sel == 1'bZ)? vD_OUT:
                                       iD_OUT;

endmodule
```

MCU

```verilog
/****************************************************************************
* Author(s): Brian Ortiz
*            Bryan Linares
*            Grace Daliwan
* Filename: MCU.v
* Date:     Nov. 27, 2018
* Project:  CECS 440 Senior Project GBRAINS
* Version:  2.15
* Credit:   Based on and extending from a design provided by R.W. Allison.
*
* A state machine implementing the MIPS Control Unit (MCU) for the major cycles
* of fetch, execute and some MIPS instructions from memory, including checking
* for interrupts.
*
*-------------------------------------------------------------------------------
*                 MCU   C O N T R O L   W O R D
*-------------------------------------------------------------------------------
*
   {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} = 8'b0_00_0_0_000;
   FS = 5'h0;
   {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;                 int_ack = 1'b0;
   #1{ns_i,ns_c,ns_v, ns_n, ns_z} = {ps_i, ps_c, ps_v, ps_n, ps_z};
   {io_cs,  io_rd,  io_wr} = 3'b0_0_0;
   {S_Sel, D_Sel} = 3'b 0_00;
   {fD_En, fT_Sel, fDIN_Sel, fDOut_Sel, fY_Sel,  fDA_Sel } =
                                                 6'b0_0_0_0_0_0;
   fpFS = 5'h0;

   {vD_En, vT_Sel, vDIN_Sel, vDOut_Sel, vY_Sel,  vDA_Sel } =
                                                 6'b0_0_0_0_0_0;
   vpFS = 5'h0;

   Type_Sel = 1'b0;
*
****************************************************************************/

//***************************************************************************
module MCU (sys_clk, reset, intr,   // system inputs
            c, n, z, v,             // ALU status inputs
            IR,                   // Instruction Register input
            int_ack,                   // output to I/O subsystem
            pc_sel, pc_ld, pc_inc, im_cs, im_wr, im_rd, ir_ld,
            D_En, DA_Sel, T_Sel, HILO_ld, Y_Sel, FS,
            dm_cs, dm_wr, dm_rd,
            io_cs, io_rd, io_wr,
            FLAGS, FLAGSIN, D_Sel, S_Sel, // interrupt paths
            fD_En, fT_Sel, fDIN_Sel, fDOut_Sel, fY_Sel,  fDA_Sel, fpFS,
                //fdp control
            vD_En, vT_Sel, vDIN_Sel, vDOut_Sel, vY_Sel,  vDA_Sel, vpFS,
                //vdp control
            Type_Sel
            );
//***************************************************************************
input   sys_clk, reset, intr;      // system clock, reset, and interrupt request
input   c, n, z, v;                // Integer ALU status inputs
input [31:0] IR;                   // Instruction Register input from IU
input [4:0] FLAGSIN;       //input to restore flags from stack when appropriate
                                 //includes IE,C,V,N,Z
output reg int_ack;              //interrupt acknowledge
output reg Type_Sel;
output reg [1:0] pc_sel;         // all the controlword fields
output reg pc_ld, pc_inc, ir_ld; // needed by the IU, DP and Data Memory
output reg im_cs, im_wr, im_rd;
output reg dm_cs, dm_rd, dm_wr;
output reg io_cs, io_rd, io_wr;
output reg fD_En, fT_Sel, fDIN_Sel, fDOut_Sel, fY_Sel,  fDA_Sel;
output reg vD_En, vT_Sel, vDIN_Sel, vDOut_Sel, vY_Sel,  vDA_Sel;
output reg [4:0] fpFS, vpFS;

output reg D_En, T_Sel, HILO_ld, S_Sel;
output reg [2:0] Y_Sel;
output reg [4:0] FS, FLAGS;
output reg [1:0] DA_Sel, D_Sel;
integer i,j;                    //iterators for simulation breaks

//*****************************
// Flag registers             *
//*****************************
reg   ps_i, ps_c, ps_v, ps_n, ps_z;   // present state registers flags
reg   ns_i, ns_c, ns_v, ns_n, ns_z;   // next state registers flags


always@(*)
   assign FLAGS = {ps_i, ps_c, ps_v, ps_n, ps_z};  // saved on interrupt

//*****************************
//  internal data structures
//*****************************
// state assignments
parameter
    //starting states
   RESET  =  00, FETCH  =  01, DECODE =  02, SETIE = 03,
   //arithmatic states
    ADD    =  10, ADDU   =  11, SUB    =  12, SUBU  =  13, MULT  = 14,
    DIV    =  15,
    //non-immediate logic states
    AND    =  20, OR     =  21, NOR    =  22, XOR   =  23, SRL   = 24,
    SRA    =  25, SLL    =  26, SLT    =  27, SLTU  =  28,
    //immediate logic states
    ORI    =  30, LUI    =  31, SLTI   =  32, XORI  =  33, ANDI  = 34,
    SLTIU  =  35, ADDI   =  36,
    //write back states
    WB_alu =  40, WB_imm =  41, WB_Din =  42, WB_hi =  43, WB_lo = 44,
    WB_mem =  45, WB_reg =  46,
    //load store states
    LW     =  50, LW_2   =  51, SW     =  52, MFLO  =  53, MFHI  = 54,
    //jump states
    JR     =  60, JR_2   =  61, JAL    =  62, JAL_2 =  63, J     = 64,
    //branch states
    BEQ    =  70, BEQ_2  =  71, BNE    =  72, BNE_2 =  73, BLEZ  = 74,
    BLEZ_2 =  75, BGTZ   =  76, BGTZ_2 =  77,
    //interrupt states
   INTR_1 = 501,  INTR_2 = 502,  INTR_3  = 503,  INTR_4 =504,  INTR_5 = 505,
    INTR_6 = 506, INTR_7 = 507, INTR_8 = 508, INTR_9 = 509,
    //break states
   BREAK = 510, ILLEGAL_OP = 511,
    //I/O states
   INPUT = 80,  INPUT_2 = 81, OUTPUT = 82, OUTPUT_2 = 83,
    //return from interrupt states
   RETI = 90, RETI_2 = 91,  RETI_3 = 92, RETI_4 = 93, RETI_5 = 94, RETI_6 = 95,
    //floating point states
   MVFR = 100,  WBF_imm = 101,  FMULT = 102, FDIV = 103, FADD = 104,
     FSUB = 105, FZERO = 106,
     //vector states
    MVVR = 110,  WBV_reg = 111,  VADDS = 112, VMULADD = 113, VANDEI = 114,
     VMULOI = 115, VCEQ = 116, VCLT = 117   ;


//FS values
parameter
    pass_s = 5'h00,  pass_t  = 5'h01,  add    = 5'h02, sub    = 5'h03,
     addu   = 5'h04,  subu    = 5'h05,  slt    = 5'h06, sltu   = 5'h07,
     fs_and = 5'h08,  fs_or   = 5'h09,  fs_xor = 5'h0a, fs_nor = 5'h0b,
     sll    = 5'h0c,  srl     = 5'h0d,  sra    = 5'h0e, andi   = 5'h16,
     ori    = 5'h17,  lui     = 5'h18,  xori   = 5'h19, inc    = 5'h0f,
     dec    = 5'h10,  inc4    = 5'h11,  dec4   = 5'h12, zeros  = 5'h13,
     ones   = 5'h14,  sp_init = 5'h15,  mult   = 5'h1E, div    = 5'h1F;

//enhanced FS values, some analogous ones are same as above
parameter
   raw_s   = 5'h04,  raw_t   = 5'h05,   adds  = 5'h08,    muladd  = 5'h09,
    andei  = 5'h02,  vcmpe   = 5'h06,   vclti = 5'h07;

//state register (up to 512 states)
reg [8:0] state;

// updating the flags
always @(posedge sys_clk, posedge reset)
   if (reset==1'b1)
      {ps_i, ps_c, ps_v, ps_n, ps_z} = 5'b0;
   else
      {ps_i, ps_c, ps_v, ps_n, ps_z} =  {ns_i, ns_c, ns_v, ns_n, ns_z};

/*************************************************
* 440 MIPS CONTROL UNIT (Finite State Machine) *
*************************************************/
always @(posedge sys_clk, posedge reset)
   if (reset)
      @(negedge sys_clk) begin  //deassert all and send to RESET state
      {pc_sel, pc_ld, pc_inc, ir_ld} = 5'b00_0_0_0;                //PC/IR sigs
      {im_cs, im_rd, im_wr} = 3'b0_0_0;                            //IM
      {D_En, DA_Sel, T_Sel, HILO_ld, Y_Sel} = 8'b0_00_0_0_000; FS = sp_init;
        //IDP load sp_init
      {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;
      #1 {ns_i, ns_c, ns_v,   ns_n,    ns_z}  = {ps_i, ps_c, ps_v, ps_n, ps_z};
      {io_cs,  io_rd,  io_wr} = 3'b0_0_0;
      {S_Sel, D_Sel} = 3'b 0_00;
      int_ack=1'b0;//DM

      {fD_En, fT_Sel, fDIN_Sel, fDOut_Sel, fY_Sel,  fDA_Sel } = 6'b0_0_0_0_0_0;
      fpFS = 5'h0;

      {vD_En, vT_Sel, vDIN_Sel, vDOut_Sel, vY_Sel,  vDA_Sel } = 6'b0_0_0_0_0_0;
      vpFS = 5'h0;
      Type_Sel  = 1'b0;
      state = RESET;
      end
   else
      case (state)
      FETCH:
         @(negedge sys_clk)
         if (int_ack==0 & (intr==1 & ps_i==1))
            begin   //*** new interrupt pending; prepare for ISR ***
            // control word assignments for "deasserting" everything
            {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} = 8'b0_00_0_0_000;
            {dm_cs,  dm_rd,  dm_wr}                  = 3'b0_0_0;
            {io_cs,  io_rd,  io_wr}                  = 3'b0_0_0;
            #1 {ns_i, ns_c, ns_v, ns_n, ns_z} = {ps_i, ps_c, ps_v, ps_n, ps_z};
            FS      = sp_init;
            int_ack = 0;
            {S_Sel, D_Sel} = 3'b 0_00;
            {fD_En, fT_Sel, fDIN_Sel, fDOut_Sel, fY_Sel, fDA_Sel } = 6'b0_0_0_0_0_0;
            fpFS = 5'h0;

            {vD_En, vT_Sel, vDIN_Sel, vDOut_Sel, vY_Sel, vDA_Sel } = 6'b0_0_0_0_0_0;
            vpFS = 5'h0;

            Type_Sel = 1'b0;
            state   = INTR_1;
            end
         else
            begin   //*** no new interrupt pending; fetch and instruction ***
               if (int_ack==1 & intr==0) int_ack=1'b0;
               // control word assignments for IR <- iM[PC]; PC <- PC+4
               {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} = 8'b0_00_0_0_000;
                    FS = sp_init;
               {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;                      int_ack=0;
               #1{ns_i,ns_c,ns_v, ns_n, ns_z} = {ps_i, ps_c, ps_v, ps_n, ps_z};
               {io_cs,  io_rd,  io_wr} = 3'b0_0_0;
               {S_Sel, D_Sel} = 3'b 0_00;
               {fD_En, fT_Sel, fDIN_Sel, fDOut_Sel, fY_Sel,  fDA_Sel } =
                                                         6'b0_0_0_0_0_0;
               fpFS = 5'h0;
               {vD_En, vT_Sel, vDIN_Sel, vDOut_Sel, vY_Sel,  vDA_Sel }=6'b0_0_0_0_0_0;
               vpFS = 5'h0;
               state = DECODE;
            end
      RESET:
         @(negedge sys_clk)
         begin
         // control word assignments for $sp <-- ALU_Out(32'h3FC)
         {pc_sel, pc_ld, pc_inc, ir_ld} = 5'b00_1_0_0;
         {im_cs, im_rd, im_wr} = 3'b0_0_0;
         {D_En, DA_Sel, T_Sel, HILO_ld, Y_Sel} = 8'b1_11_0_0_000; FS = sp_init;
         {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;                      int_ack=0;
         #1 {ns_i, ns_c, ns_v, ns_n, ns_z} = {ps_i, ps_c, ps_v, ps_n, ps_z};
         {io_cs,  io_rd,  io_wr} = 3'b0_0_0;
         {S_Sel, D_Sel} = 3'b 0_00;
         {fD_En, fT_Sel, fDIN_Sel, fDOut_Sel, fY_Sel,  fDA_Sel } = 6'b0_0_0_0_0_0;
         fpFS = 5'h0;

         {vD_En, vT_Sel, vDIN_Sel, vDOut_Sel, vY_Sel,  vDA_Sel } = 6'b0_0_0_0_0_0;
         vpFS = 5'h0;

         state = FETCH;
         end
     DECODE:
         begin
            @(negedge sys_clk)
            if ( IR[31:26] == 6'h1F ) begin
                //check MIPS format for 'e_key': Enhanced Instructions
            //RS <- $rs, RT <- $rt (default), because fs=rs, and ft=rt in
                //Instruction Format
               {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} = 8'b0_00_0_0_000;
             FS = 5'h0;
        {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;                 int_ack = 1'b0;
        #1{ns_i,ns_c,ns_v, ns_n, ns_z} = {ps_i, ps_c, ps_v, ps_n, ps_z};
        {io_cs,  io_rd,  io_wr} = 3'b0_0_0;
        {S_Sel, D_Sel} = 3'b 0_00;
        {fD_En, fT_Sel, fDIN_Sel, fDOut_Sel, fY_Sel,  fDA_Sel } =
                                                    6'b0_0_0_0_0_0;
        fpFS = 5'h0;

        {vD_En, vT_Sel, vDIN_Sel, vDOut_Sel, vY_Sel, vDA_Sel } = 'b0_0_0_0_0_0;
        vpFS = 5'h0;

        Type_Sel = 1'b1;
        //$display("IR %h, es %h, et %h, ed %h, fmt %h, funct %h ", IR,
             //IR[25:21],IR[20:16],IR[15:11],IR[10:6],IR[5:0]  );
        case ( IR[5:0] )
          // 6'OPCODE(1F) | 5'ES  |  5'ET  |  5'ED  |  5'FMT   | 6'FUNCT
           6'h00 :  state = MVFR;       // F(ed) = {R(rs),R(rt)}
           6'h01 :  state = FMULT;      // F(ed) = F(es) * F(et)
           6'h02 :  state = FDIV;       // F(ed) = F(es) / F(et)
           6'h03 :  state = FADD;       // F[ed] = F(es) + F(et)
           6'h04 :  state = FSUB;       // F[ed] = F(es) - F(et)
           6'h05 :  state = FZERO;      // F[ed] = 0.0

           6'h06 :  state = MVVR;       // V(ed) = {R(rs),R(rt)}
           6'h07 :  state = VADDS;      // V(ed) = {V(es) + V(et)}
                                                // Saturated 8 bit
           6'h08 :  state = VMULADD;    // V(ed) = {V(es)*V(et) + V(ed)}
           6'h09 :  state = VANDEI;     // V(ed) = {V(rs)&V(rt)}
                                                // even 8 bit ints
           6'h0A :  state = VCEQ;       // V(ed) = {V(rs)==V(rt)} 8 bit
                                                // Equals compare
           6'h0B :  state = VCLT;       // V(ed) = {V(rs)<V(rt)} 8 bit
                                                // less than compare
           default: state = ILLEGAL_OP;
        endcase

    end ///end enhanced instructions

    else if ( IR[31:26] == 6'h0 )     // check for MIPS format
        begin    // it is an R-type format
              // control word assignments: RS <-- $rs    RT <-- $rt
        {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} = 8'b0_00_0_0_000;
             FS = 5'h0;
        {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;                 int_ack = 1'b0;
        #1{ns_i,ns_c, ns_v,ns_n, ns_z} = {ps_i, ps_c, ps_v, ps_n, ps_z};
        {io_cs,  io_rd,  io_wr} = 3'b0_0_0;
        {S_Sel, D_Sel} = 3'b 0_00;
        {fD_En, fT_Sel, fDIN_Sel, fDOut_Sel, fY_Sel,  fDA_Sel } =
                                                    6'b0_0_0_0_0_0;
        fpFS = 5'h0;
        {vD_En, vT_Sel, vDIN_Sel, vDOut_Sel, vY_Sel,  vDA_Sel } =
                                                    6'b0_0_0_0_0_0;
        vpFS = 5'h0;
        case ( IR[5:0] )
           6'h00 :  state = SLL;
           6'h02 :  state = SRL;
           6'h03 :  state = SRA;
           6'h08 :  state = JR;
```

```
               6'h0D :  state = BREAK;
               6'h10 :  state = MFHI;
               6'h12 :  state = MFLO;
               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 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} = 8'b0_00_1_0_000;
                     FS = 5'h0;
               {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;           int_ack = 1'b0;
               #1{ns_i,ns_c,ns_v,ns_n,ns_z}= {ps_i, ps_c, ps_v, ps_n, ps_z};
               {io_cs,  io_rd,  io_wr} = 3'b0_0_0;
               {S_Sel, D_Sel} = 3'b 0_00;
               {fD_En, fT_Sel, fDIN_Sel, fDOut_Sel, fY_Sel,  fDA_Sel } =
                                                                6'b0_0_0_0_0_0;
               fpFS = 5'h0;
               {vD_En, vT_Sel, vDIN_Sel, vDOut_Sel, vY_Sel,  vDA_Sel } =
                                                                6'b0_0_0_0_0_0;
               vpFS = 5'h0;
               case ( IR[31:26] )
                  6'h02 : state = J;
                  6'h03 : state = JAL; //R[31]=PC+8;PC=JumpAddr
                  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'h2B : state = SW;
                  6'h23 : state = LW;
                  default: state = ILLEGAL_OP;
               endcase
         end // end of else for I-type or J-type formats
      end    // end of DECODE

MVVR:
   @(negedge sys_clk) begin
   // control word assignments for
```

```
        //V[ed[63:0]] <-- {IntR[rs[31:0]], IntR[rt[31:0]]};
      {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} = 8'b0_00_0_0_000; FS = 5'h0;
      {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;                        int_ack=0;
      #1 {ns_i, ns_c, ns_v, ns_n, ns_z} = {ps_i, ps_c, ps_v, ps_n, ps_z};
      {fD_En, fT_Sel, fDIN_Sel, fDOut_Sel, fY_Sel,  fDA_Sel } = 6'b0_0_0_0_0_0;
      fpFS = 5'h0;
      {vD_En, vT_Sel, vDIN_Sel, vDOut_Sel, vY_Sel,  vDA_Sel } = 6'b1_1_0_0_0_0;
      vpFS = pass_t;
      state = FETCH;
      end

   VADDS:
      @(negedge sys_clk) begin
      // control word assignments for
        // {carry,V[ed]} <-- V[eS] + V[eT];
      // V[ed] <-- carry ? (8'hFF) : V[ed];
      {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} = 8'b0_00_0_0_000; FS = 5'h0;
      {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;                        int_ack=0;
      #1 {ns_i, ns_c, ns_v, ns_n, ns_z} = {ps_i, ps_c, ps_v, ps_n, ps_z};
      {fD_En, fT_Sel, fDIN_Sel, fDOut_Sel, fY_Sel,  fDA_Sel } = 6'b0_0_0_0_0_0;
      fpFS = 5'h0;
      {vD_En, vT_Sel, vDIN_Sel, vDOut_Sel, vY_Sel,  vDA_Sel } = 6'b1_0_0_0_0_0;
      vpFS = adds;
      state = FETCH;
      end

   VMULADD:
      @(negedge sys_clk) begin
      // control word assignments for V[ed] <-- V[eS] * V[eT] + V[ed];
      {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} = 8'b0_00_0_0_000; FS = 5'h0;
      {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;                        int_ack=0;
      #1 {ns_i, ns_c, ns_v, ns_n, ns_z} = {ps_i, ps_c, ps_v, ps_n, ps_z};
      {fD_En, fT_Sel, fDIN_Sel, fDOut_Sel, fY_Sel,  fDA_Sel } = 6'b0_0_0_0_0_0;
      fpFS = 5'h0;
      {vD_En, vT_Sel, vDIN_Sel, vDOut_Sel, vY_Sel,  vDA_Sel } = 6'b1_0_0_0_0_0;
      vpFS = muladd;
      state = FETCH;
      end
   VANDEI:
      @(negedge sys_clk) begin
      // control word assignments for V[ed] <-- V[eS] & V[eT];
      {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} = 8'b0_00_0_0_000; FS = 5'h0;
      {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;                        int_ack=0;
      #1 {ns_i, ns_c, ns_v, ns_n, ns_z} = {ps_i, ps_c, ps_v, ps_n, ps_z};
      {fD_En, fT_Sel, fDIN_Sel, fDOut_Sel, fY_Sel,  fDA_Sel } = 6'b0_0_0_0_0_0;
      fpFS = 5'h0;
      {vD_En, vT_Sel, vDIN_Sel, vDOut_Sel, vY_Sel,  vDA_Sel } = 6'b1_0_0_0_0_0;
      vpFS = andei;
      state = FETCH;
      end

   VCEQ:
      @(negedge sys_clk) begin
      // control word assignments for
        // V[ed] <-- (V[eS] == V[eT]) ? 8'hFF : 8'h0;
      {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} = 8'b0_00_0_0_000; FS = 5'h0;
      {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;                        int_ack=0;
      #1 {ns_i, ns_c, ns_v, ns_n, ns_z} = {ps_i, ps_c, ps_v, ps_n, ps_z};
      {fD_En, fT_Sel, fDIN_Sel, fDOut_Sel, fY_Sel,  fDA_Sel } = 6'b0_0_0_0_0_0;
      fpFS = 5'h0;
      {vD_En, vT_Sel, vDIN_Sel, vDOut_Sel, vY_Sel,  vDA_Sel } = 6'b1_0_0_0_0_0;
      vpFS = vcmpe;
      state = FETCH;
      end

   VCLT:
      @(negedge sys_clk) begin
      // control word assignments for
        // V[ed] = (V[eS] < V[eT]) ? 8'hFF : 8'h0;
      {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} = 8'b0_00_0_0_000; FS = 5'h0;
      {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;                        int_ack=0;
      #1 {ns_i, ns_c, ns_v, ns_n, ns_z} = {ps_i, ps_c, ps_v, ps_n, ps_z};
      {fD_En, fT_Sel, fDIN_Sel, fDOut_Sel, fY_Sel,  fDA_Sel } = 6'b0_0_0_0_0_0;
      fpFS = 5'h0;
      {vD_En, vT_Sel, vDIN_Sel, vDOut_Sel, vY_Sel,  vDA_Sel } = 6'b1_0_0_0_0_0;
      vpFS = vclti;
      state = FETCH;
      end

   MVFR:
      @(negedge sys_clk) begin
      // control word assignments for  F[ed] = {rS,rT};
      {pc_sel, pc_ld, pc_inc, ir_ld} = 5'bxx_0_0_0;
      {im_cs, im_rd, im_wr} = 3'b0_0_0;
      {D_En, DA_Sel, T_Sel, HILO_ld, Y_Sel} = 8'b0_00_0_0_000; FS = 5'h0;
      {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;                        int_ack=0;
      #1 {ns_i, ns_c, ns_v, ns_n, ns_z} = {ps_i, ps_c, ps_v, ps_n, ps_z};
      {fD_En, fT_Sel, fDIN_Sel, fDOut_Sel, fY_Sel,  fDA_Sel } = 6'b1_1_0_0_0_0;
      fpFS = pass_t;
      state = WBF_imm;
      end

   FMULT:
      @(negedge sys_clk) begin
      // control word assignments for F[ed] <-- F[es] * F[et]
      {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} = 8'b0_00_0_0_000; FS = 5'h0;
      {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;                        int_ack=0;
      #1 {ns_i, ns_c, ns_v, ns_n, ns_z} = {ps_i, ps_c, ps_v, ps_n, ps_z};
      {fD_En, fT_Sel, fDIN_Sel, fDOut_Sel, fY_Sel,  fDA_Sel } = 6'b1_0_0_0_0_0;
      fpFS = mult;
      state = FETCH;
      end

   FDIV:
      @(negedge sys_clk) begin
      // control word assignments for F[ed] <-- F[es] / F[et]
      {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} = 8'b0_00_0_0_000; FS = 5'h0;
      {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;                        int_ack=0;
      #1 {ns_i, ns_c, ns_v, ns_n, ns_z} = {ps_i, ps_c, ps_v, ps_n, ps_z};
      {fD_En, fT_Sel, fDIN_Sel, fDOut_Sel, fY_Sel,  fDA_Sel } = 6'b1_0_0_0_0_0;
      fpFS = div;
      state = FETCH;
      end
   FADD:
      @(negedge sys_clk) begin
      // control word assignments for F[ed] <-- F[es] + F[et]
      {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} = 8'b0_00_0_0_000; FS = 5'h0;
      {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;                       int_ack=0;
      #1 {ns_i, ns_c, ns_v, ns_n, ns_z} = {ps_i, ps_c, ps_v, ps_n, ps_z};
      {fD_En, fT_Sel, fDIN_Sel, fDOut_Sel, fY_Sel,  fDA_Sel } = 6'b1_0_0_0_0_0;
      fpFS = add;
      state = FETCH;
      end
   FSUB:
      @(negedge sys_clk) begin
      // control word assignments for F[ed] <-- F[es] - F[et]
      {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} = 8'b0_00_0_0_000; FS = 5'h0;
      {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;                       int_ack=0;
      #1 {ns_i, ns_c, ns_v, ns_n, ns_z} = {ps_i, ps_c, ps_v, ps_n, ps_z};
      {fD_En, fT_Sel, fDIN_Sel, fDOut_Sel, fY_Sel,  fDA_Sel } = 6'b1_0_0_0_0_0;
      fpFS = sub;
      state = FETCH;
      end
   FZERO:
      @(negedge sys_clk) begin
      // control word assignments for F[ed] <-- 0.0
      {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} = 8'b0_00_0_0_000; FS = 5'h0;
      {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;                       int_ack=0;
      #1 {ns_i, ns_c, ns_v, ns_n, ns_z} = {ps_i, ps_c, ps_v, ps_n, ps_z};
      {fD_En, fT_Sel, fDIN_Sel, fDOut_Sel, fY_Sel,  fDA_Sel } = 6'b1_0_0_0_0_0;
      fpFS = zeros;
      state = FETCH;
      end

  WBF_imm:
      @(negedge sys_clk) begin
      // control word assignments for F[rd] <-- {RS,RT} on Long_Out wire
      {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} = 8'b0_00_0_0_000; FS = 5'h0;
      {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;                       int_ack=1'b0;
      #1 {ns_i, ns_c, ns_v, ns_n, ns_z} = {ps_i, ps_c, ps_v, ps_n, ps_z};
      {fD_En, fT_Sel, fDIN_Sel, fDOut_Sel, fY_Sel,  fDA_Sel } = 6'b1_1_0_0_0_0;
      fpFS = pass_t;
      state = FETCH;
      end

   SETIE:
      @(negedge sys_clk) begin
      // control word assignments:  IE <-- 1'B1
      {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} = 8'b0_00_0_0_000; FS = 5'h0;
      {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;                       int_ack=0;
      {io_cs, io_rd, io_wr} = 3'b0_0_0;
      #1 {ns_i, ns_c, ns_v, ns_n, ns_z} = {ps_i, ps_c, ps_v, ps_n, ps_z};
      state   = FETCH;
```

```
         end

      INPUT:
         @(negedge sys_clk) begin
         // control word assignment for ALU_OUT <-- RS($rs) + RT(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} = 8'b0_00_0_0_000; FS = add;
         {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;                        int_ack = 0;
         {io_cs, io_rd, io_wr} = 3'b0_0_0;
         #1 {ns_i, ns_c, ns_v, ns_n, ns_z} = {ps_i, c, v, n, z};
         {S_Sel, D_Sel} = 3'b 0_00;
         state = INPUT_2;
         end

      INPUT_2:
         @(negedge sys_clk) begin
         // control word assignments for D_in <-- IOM[ ALU_Out($rs+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} = 8'b0_00_0_0_000; FS = 5'h0;
         {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;                        int_ack=1'b0;
         {io_cs, io_rd, io_wr} = 3'b1_1_0;
         #1 {ns_i, ns_c, ns_v, ns_n, ns_z} = {ps_i, ps_c, ps_v, ps_n, ps_z};
         {S_Sel, D_Sel} = 3'b 0_00;
         state = WB_reg;
         end

      OUTPUT:
         @(negedge sys_clk) begin
         // control word assignments for ALU_Out <-- RS($rs) + RT(se_16),
           // RT <-- $rt
         {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} = 8'b0_00_0_0_000; FS = add;
         {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;                        int_ack=1'b0;
         {io_cs, io_rd, io_wr} = 3'b0_0_0;
         #1 {ns_i, ns_c, ns_v, ns_n, ns_z} = {ps_i, c, v, n, z};
         {S_Sel, D_Sel} = 3'b 0_00;
         state = OUTPUT_2;
         end

      OUTPUT_2:
         @(negedge sys_clk) begin
         // control word assignments for IOM[ ALU_Out($rs+se_16) ] <-- RT($rt)
         {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} = 8'b0_00_0_0_000; FS = 5'h0;
         {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;                        int_ack=1'b0;
         {io_cs,  io_rd, io_wr} = 3'b1_0_1;
         #1 {ns_i, ns_c, ns_v, ns_n, ns_z} = {ps_i, ps_c, ps_v, ps_n, ps_z};
         {S_Sel, D_Sel} = 3'b 0_00;
         state = FETCH;
         end

      ADD:
         @(negedge sys_clk) begin
         // control word assignments:  ALU_Out <-- RS($rs) + RT($rt)
         {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} = 8'b0_00_0_0_000; FS = add;
         {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;                        int_ack=0;
         {io_cs, io_rd, io_wr} = 3'b0_0_0;
```

```
  #1 {ns_i, ns_c, ns_v, ns_n, ns_z} = {ps_i, c, v, n, z};
  {S_Sel, D_Sel} = 3'b 0_00;
  state = WB_alu;
  end

SUB:
    @(negedge sys_clk) begin
  // control word assignments:  ALU_Out <-- RS($rs) + RT($rt)
  {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} = 8'b0_00_0_0_000; FS = sub;
  {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;                       int_ack=0;
  #1 {ns_i, ns_c, ns_v, ns_n, ns_z} = {ps_i, c, v, n, z};
  state = WB_alu;
  end

AND:
    @(negedge sys_clk) begin
  // control word assignments:  ALU_Out <-- RS($rs) + RT($rt)
  {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} = 8'b0_00_0_0_000; FS = fs_and;
  {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;                       int_ack=0;
  #1 {ns_i, ns_c, ns_v, ns_n, ns_z} = {ps_i, c, v, ps_n, ps_z};
  state = WB_alu;
  end

XOR:
     @(negedge sys_clk) begin
  // control word assignments:  ALU_Out <-- RS($rs) ^ RT($rt)
  {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} = 8'b0_00_0_0_000; FS = fs_xor;
  {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;                       int_ack=0;
  #1 {ns_i, ns_c, ns_v, ns_n, ns_z} = {ps_i, c, v, ps_n, ps_z};
  state = WB_alu;
  end

OR:
 @(negedge sys_clk) begin
  // control word assignments:  ALU_Out <-- RS($rs) | RT($rt)
  {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} = 8'b0_00_0_0_000; FS = fs_or;
  {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;                       int_ack=0;
  #1 {ns_i, ns_c, ns_v, ns_n, ns_z} = {ps_i, c, v, ps_n, ps_z};
  state = WB_alu;
  end

NOR:
 @(negedge sys_clk) begin
  // control word assignments:  ALU_Out <-- ~(RS($rs) | RT($rt))
  {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} = 8'b0_00_0_0_000; FS = fs_nor;
  {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;                       int_ack=0;
  #1 {ns_i, ns_c, ns_v, ns_n, ns_z} = {ps_i, c, v, ps_n, ps_z};
  state = WB_alu;
  end

SLTU:
 @(negedge sys_clk) begin
  // control word assignments:ALU_Out <-- RS($rs) < RT($rt) ? 1:0 unsigned
  {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} = 8'b0_00_0_0_000; FS = sltu;
  {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;                          int_ack=0;
  #1 {ns_i, ns_c, ns_v, ns_n, ns_z} = {ps_i, c, v, ps_n, ps_z};
  state = WB_alu;
  end

SLTIU:
     @(negedge sys_clk) begin
  // control word assignments:ALU_Out <-- RS($rs) < RT($rt) ? 1:0 unsigned
  {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} = 8'b0_00_1_0_000; FS = sltu;
  {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;                          int_ack=0;
  #1 {ns_i, ns_c, ns_v, ns_n, ns_z} = {ps_i, c, v, ps_n, ps_z};
  state = WB_imm;
  end

MFHI:
    @(negedge sys_clk) begin
    // control word assignments: RegFile(rd) <-- HI
    {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} = 8'b1_00_0_0_100; FS = 5'b0;
  {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;
    #1 {ns_i, ns_c, ns_v, ns_n, ns_z} = {ps_i, ps_c, ps_v, ps_n, ps_z};
  state = FETCH;
    end

MFLO:
    @(negedge sys_clk) begin
    // control word assignments: RegFile(rd) <-- LO
    {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} = 8'b1_00_0_0_011; FS = 5'b0;
  {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;
    #1 {ns_i, ns_c, ns_v, ns_n, ns_z} = {ps_i, ps_c, ps_v, ps_n, ps_z};
  state = FETCH;
    end

MULT:
    @(negedge sys_clk) begin
    // control word assignments: {HI,LO} <-- RS($rs) * RT($rt)
    {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} = 8'b0_00_0_1_000; FS = mult;
  {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;
    #1 {ns_i, ns_c, ns_v, ns_n, ns_z} = {ps_i, c, ps_v, n, z};
  state = FETCH;
    end

DIV:
    @(negedge sys_clk) begin
    // ctrl word assignments: HI <-- RS($rs) % RT($rt),
    // LO <-- RS($rs) / RT($rt)
    {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} = 8'b0_00_0_1_000; FS = div;
  {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;
    #1 {ns_i, ns_c, ns_v, ns_n, ns_z} = {ps_i, c, ps_v, n, z};
  state = FETCH;
    end
```

```
 XORI:
     @(negedge sys_clk) begin
   // ctrl word assignments for ALU_Out <-- RS($rs) | {16'h0, RT[15:0]}
   {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} = 8'b0_00_1_0_000; FS = xori;
   {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;                        int_ack=0;
   #1 {ns_i, ns_c, ns_v, ns_n, ns_z} = {ps_i, ps_c, ps_v, ps_n, ps_z};
   state = WB_imm;
   end

 ANDI:
     @(negedge sys_clk) begin
   // ctrl word assignments for ALU_Out <-- RS($rs) | {16'h0, RT[15:0]}
   {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} = 8'b0_00_0_0_000; FS = andi;
   {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;                        int_ack=0;
   #1 {ns_i, ns_c, ns_v, ns_n, ns_z} = {ps_i, ps_c, ps_v, ps_n, ps_z};
   state = WB_imm;
   end

ORI:
   @(negedge sys_clk) begin
   // ctrl word assignments for ALU_Out <-- RS($rs) | {16'h0, RT[15:0]}
   {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} = 8'b0_00_0_0_000; FS = ori;
   {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;                        int_ack=0;
   #1 {ns_i, ns_c, ns_v, ns_n, ns_z} = {ps_i, ps_c, ps_v, ps_n, ps_z};
   state = WB_imm;
   end

LUI:
   @(negedge sys_clk) begin
   // control word assignments for ALU_Out <-- { RT[15:0], 16'h0}
   {pc_sel, pc_ld, pc_inc, ir_ld} = 5'bxx_0_0_0;
   {im_cs, im_rd, im_wr} = 3'b0_0_0;
   {D_En, DA_Sel, T_Sel, HILO_ld, Y_Sel} = 8'b0_00_0_0_000; FS = lui;
   {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;                        int_ack=0;
   #1 {ns_i, ns_c, ns_v, ns_n, ns_z} = {ps_i, ps_c, ps_v, ps_n, ps_z};
   state = WB_imm;
   end

SW:
   @(negedge sys_clk) begin
   // control word assignments for ALU_Out <-- RS($rs) + RT(se_16),
     // RT <-- $rt
   {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} = 8'b0_00_0_0_000; FS = add;
   {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;                        int_ack=0;
   {io_cs, io_rd, io_wr} = 3'b0_0_0;
   #1 {ns_i, ns_c, ns_v, ns_n, ns_z} = {ps_i, ps_c, ps_v, ps_n, ps_z};
   state = WB_mem;
   end

BEQ:
   @(negedge sys_clk) begin
   // control word assignments for ALU_Out <-- RS($rs) - RT($rt)
     // (affects zero flag)
   {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} = 8'b0_00_0_0_000; FS = sub;
      {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;                       int_ack=0;
      #1 {ns_i, ns_c, ns_v, ns_n, ns_z} = {ps_i, c, v, ps_n, ps_z};
      state = BEQ_2;
      end
   BEQ_2:
      @(negedge sys_clk) begin
      // control word assignments for if(zero==1) PC <-- (PC+4) + {SE_16[29:0],
      // 2'b00} (Branch Addr.)
      {pc_sel, pc_ld, pc_inc, ir_ld} = (z == 1'b1)? 5'b00_1_0_0 : 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} = 8'b0_00_1_0_000; FS = sub;
      {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;                       int_ack=0;
      #1 {ns_i, ns_c, ns_v, ns_n, ns_z} = {ps_i, ps_c, ps_v, ps_n, ps_z};
      state = FETCH;
      end

   BNE:
      @(negedge sys_clk) begin
      // control word assignments for ALU_Out <-- RS($rs) - RT($rt)
        //(affects zero flag)
      {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} = 8'b0_00_0_0_000; FS = sub;
      {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;                       int_ack=0;
      #1 {ns_i, ns_c, ns_v, ns_n, ns_z} = {ps_i, c, v, n, z};
      state = BNE_2;
      end
   BNE_2:
      @(negedge sys_clk) begin
      // control word assignments for if
        //(zero flag==0) PC <-- (PC+4) + {SE_16[29:0], 2'b00} (Branch Addr.)
      {pc_sel, pc_ld, pc_inc, ir_ld} = (z == 1'b0)? 5'b00_1_0_0 : 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} = 8'b0_00_1_0_000; FS = sub;
      {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;                       int_ack=0;
      #1 {ns_i,   ns_c,   ns_v,   ns_n, ns_z} = {ps_i, ps_c, ps_v, ps_n, ps_z};
      state = FETCH;
      end
   BLEZ:
      @(negedge sys_clk) begin
      // control word assignments for ALU_Out <-- RS($rs) - RT($rt)
        // (affects zero flag)
      {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} = 8'b0_00_0_0_000; FS = sub;
      {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;                       int_ack=0;
      #1 {ns_i, ns_c, ns_v, ns_n, ns_z} = {ps_i, ps_c, ps_v, ps_n, ps_z};
      state = BLEZ_2;
      end
   BLEZ_2:
      @(negedge sys_clk) begin
      // control word assignments for if
        //(zero flag==0) PC <-- (PC+4) + {SE_16[29:0], 2'b00} (Branch Addr.)
      {pc_sel, pc_ld, pc_inc, ir_ld} = (n == 1'b1||z==1'b1)? 5'b00_1_0_0 :
        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} = 8'b0_00_1_0_000; FS = sub;
      {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;                       int_ack=0;
      #1 {ns_i, ns_c, ns_v, ns_n, ns_z} = {ps_i, ps_c, ps_v, ps_n, ps_z};
      state = FETCH;
      end
```

```
 BGTZ:
   @(negedge sys_clk) begin
   // control word assignments for ALU_Out <-- RS($rs) - RT($rt)
     //(affects zero flag)
   {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} = 8'b0_00_0_0_000; FS = sub;
   {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;                        int_ack=0;
   #1 {ns_i, ns_c, ns_v, ns_n, ns_z} = {ps_i, ps_c, ps_v, ps_n, ps_z};
   state = BGTZ_2;
   end

BGTZ_2:
   @(negedge sys_clk) begin
   // control word assignments for if
     //(zero flag==0) PC <-- (PC+4) + {SE_16[29:0], 2'b00} (Branch Addr.)
   {pc_sel, pc_ld, pc_inc, ir_ld} = (n == 1'b0&&z==1'b0)? 5'b00_1_0_0 :
     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} = 8'b0_00_1_0_000; FS = sub;
   {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;                        int_ack=0;
   #1 {ns_i, ns_c, ns_v, ns_n, ns_z} = {ps_i, ps_c, ps_v, ps_n, ps_z};
   state = FETCH;
   end

ADDI:
   @(negedge sys_clk) begin
   // ctrl word assignments for ALU_Out <-- RS($rs) + RT[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} = 8'b0_00_0_0_000; FS = add;
   {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;                        int_ack=0;
   {io_cs,  io_rd,  io_wr}                 = 3'b0_0_0;
   #1 {ns_i,   ns_c,   ns_v,   ns_n,    ns_z}  = {ps_i, c, v, n, z};
   state = WB_imm;
   end

SRL:
   @(negedge sys_clk) begin
   // ctrl word assignments for ALU_Out <-- RT($rt) >> (IR[10:6]) shamnt
   {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} = 8'b0_00_0_0_000; FS = srl;
   {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;                        int_ack=0;
   #1 {ns_i, ns_c, ns_v, ns_n, ns_z} = {ps_i, ps_c, ps_v, ps_n, ps_z};
   state = WB_alu;
   end

J:
   @(negedge sys_clk) begin
   // ctrl word assignments for PC <-- {PC_out[31:28], IR_out[25:0],
     //2'b00}(Jump Addr.)
   {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} = 8'b0_00_0_0_000; FS = 5'h0;
   {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;                        int_ack=0;
   #1 {ns_i, ns_c, ns_v, ns_n, ns_z} = {ps_i, ps_c, ps_v, ps_n, ps_z};
   state = FETCH;
   end

JR:
   @(negedge sys_clk) begin
   // ctrl word assignments for ALU_Out <-- RS($rs)
   {pc_sel, pc_ld, pc_inc, ir_ld} = 5'b10_0_0_0;
   {im_cs, im_rd, im_wr} = 3'b0_0_0;
   {D_En, DA_Sel, T_Sel, HILO_ld, Y_Sel} = 8'b0_00_0_0_000; FS = 5'h0;
   {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;                         int_ack=0;
   {io_cs,  io_rd,  io_wr}                    = 3'b0_0_0;
   #1 {ns_i,   ns_c,   ns_v, ns_n, ns_z}  = {ps_i, ps_c, ps_v, ps_n, ps_z};
   state = JR_2;
   end

JR_2:
   @(negedge sys_clk) begin
   // ctrl word assignments for PC <- ALU_Out($rs)
   {pc_sel, pc_ld, pc_inc, ir_ld} = 5'b10_1_0_0;
   {im_cs, im_rd, im_wr} = 3'b0_0_0;
   {D_En, DA_Sel, T_Sel, HILO_ld, Y_Sel} = 8'b0_00_0_0_000; FS = 5'h0;
   {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;                         int_ack=0;
   {io_cs,  io_rd,  io_wr}                    = 3'b0_0_0;
   #1 {ns_i,   ns_c,   ns_v, ns_n, ns_z}  = {ps_i, ps_c, ps_v, ps_n, ps_z};
   state = FETCH;
   end

 JAL:
     @(negedge sys_clk) begin
     //ctrl word assignments RegFile($31) <-- PC
     {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} = 8'b1_10_0_0_001; FS = 5'b0;
   {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;
     #1 {ns_i, ns_c, ns_v, ns_n, ns_z} = {ps_i, ps_c, ps_v, ps_n, ps_z};
   state = J;
     end

SRA:
   @(negedge sys_clk) begin
   // ctrl word assignments for ALU_Out <-- RT($rt) >> (IR[10:6])
     // shamnt (arith.)
   {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} = 8'b0_00_0_0_000; FS = sra;
   {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;                         int_ack=0;
   {io_cs,  io_rd,  io_wr}                    = 3'b0_0_0;
   #1 {ns_i,   ns_c,   ns_v,   ns_n,    ns_z}  = {ps_i, c, v, n, z};
   state = WB_alu;
   end

SLL:
   @(negedge sys_clk) begin
   // ctrl word assignments for ALU_Out <-- RT($rt) << (IR[10:6]) shamnt
   {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} = 8'b0_00_0_0_000; FS = sll;
   {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;                         int_ack=0;
   #1 {ns_i, ns_c, ns_v, ns_n, ns_z} = {ps_i, ps_c, ps_v, ps_n, ps_z};
   state = WB_alu;
   end

SLT:
   @(negedge sys_clk) begin
   // ctrl word assignments for ALU_Out <-- RS($rs) < RT($rt) ? 1:0
   {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} = 8'b0_00_0_0_000; FS = slt;
```

```
     {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;                       int_ack=0;
     #1 {ns_i, ns_c, ns_v, ns_n, ns_z} = {ps_i, ps_c, ps_v, ps_n, ps_z};
     state = WB_alu;
     end

   SLTI:
     @(negedge sys_clk) begin
     // ctrl word assignments for ALU_Out <-- RS($rs) < RT[se_16] ? 1:0
     {pc_sel, pc_ld, pc_inc, ir_ld} = 5'b00_0_0_0;
     {im_cs, im_rd, im_wr} = 3'b0_0_0;
     {D_En, DA_Sel, T_Sel, HILO_ld, Y_Sel} = 8'b0_00_0_0_000; FS = slt;
     {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;                       int_ack=0;
     #1 {ns_i, ns_c, ns_v, ns_n, ns_z} = {ps_i, ps_c, ps_v, ps_n, ps_z};
     state = WB_imm;
     end
   LW:
     @(negedge sys_clk) begin
     // crtl word assignments ALU_Out <-- RS($rs) + RT($rt)
     {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} = 8'b0_00_0_0_000; FS = add;
     {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;
     #1 {ns_i, ns_c, ns_v, ns_n, ns_z} = {ps_i, ps_c, ps_v, ps_n, ps_z};
     state = LW_2;
       end

   LW_2:
     @(negedge sys_clk) begin
       // crtl word assignments ALU_Out <-- Dmem((RS($rs)+ RT($rt))
     {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} = 8'b0_00_0_0_000; FS = add;
     {dm_cs, dm_rd, dm_wr} = 3'b1_1_0;
     #1 {ns_i, ns_c, ns_v, ns_n, ns_z} = {ps_i, ps_c, ps_v, ps_n, ps_z};
     state = WB_reg;
       end

   WB_reg:
     @(negedge sys_clk) begin
     // crtl word assignments R[rt] <-- ALU_Out
     {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} = 8'b1_01_0_0_010; FS = 5'h0;
     {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;
     #1 {ns_i, ns_c, ns_v, ns_n, ns_z} = {ps_i, ps_c, ps_v, ps_n, ps_z};
     state = FETCH;
     end
  WB_alu:
     @(negedge sys_clk) begin
     // control word assignments for R[rd] <-- ALU_Out
     {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} = 8'b1_00_0_0_000; FS = 5'h0;
     {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;                       int_ack=0;
     {io_cs, io_rd,  io_wr} = 3'b0_0_0;
     #1 {ns_i, ns_c, ns_v, ns_n, ns_z} = {ps_i, ps_c, ps_v, ps_n, ps_z};
     state = FETCH;
     end

  WB_imm:
     @(negedge sys_clk) begin
     // control word assignments for R[rt] <-- ALU_Out
     {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} = 8'b1_01_0_0_000; FS = 5'h0;
     {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;                        int_ack=1'b0;
     #1 {ns_i, ns_c, ns_v, ns_n, ns_z} = {ps_i, ps_c, ps_v, ps_n, ps_z};
     state = FETCH;
     end

  WB_mem:
     @(negedge sys_clk) begin
     // control word assignments for M[ ALU_Out($rs+se_16) ] <-- RT($rt)
     {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} = 8'b0_00_0_0_000; FS = 5'h0;
     {dm_cs, dm_rd, dm_wr} = 3'b1_0_1;                        int_ack=1'b0;
     {io_cs, io_rd,  io_wr} = 3'b0_0_0;
     #1 {ns_i, ns_c, ns_v, ns_n, ns_z} = {ps_i, ps_c, ps_v, ps_n, ps_z};
     state = FETCH;
     end

  BREAK:
  @(negedge sys_clk) begin
     $display("BREAK INSTRUCTION FETCHED %t",$time);
     // control word assignments for "deasserting" everything
     @(negedge sys_clk) begin
     {pc_sel, pc_ld, pc_inc, ir_ld} = 5'b00_0_0_0;
     {im_cs, im_rd, im_wr} = 3'b0_0_0;
     {D_En, DA_Sel, T_Sel, HILO_ld, Y_Sel} = 8'b0_00_0_0_000; FS = 5'h0;
     {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;                        int_ack=0;
     #1 {ns_i, ns_c, ns_v, ns_n, ns_z} = {ps_i, ps_c, ps_v, ps_n, ps_z};
     end
     $display(" R E G I S T E R S  A F T E R   B R E A K");
     $display(" ");
     Dump_Registers;   // task to output MIPS RegFiles
     $display(" ");
     //Dump_Data_Memory;
      $display(" ");
     //Dump_IO_Memory;
     $finish;

  end

  ILLEGAL_OP:
     @(negedge sys_clk) begin
     $display("ILLEGAL OPCODE FETCHED %t",$time);
     // control word assignments for "deasserting" everything
     {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} = 8'b0_00_0_0_000; FS = 5'h0;
     {dm_cs, dm_rd, dm_wr} = 3'b0_0_0;                        int_ack=1'b0;
     #1 {ns_i, ns_c, ns_v, ns_n, ns_z} = {ps_i, ps_c, ps_v, ps_n, ps_z};
     Dump_Registers;
     Dump_PC_and_IR;
     $finish;
     end

  INTR_1:  // steps to Save PC in dM[$sp-4] and Flags in dM[$sp-8],
               //then PC loads address of interrupt vector PC <-dM[0x3FC];
     @(negedge sys_clk) begin
     // control word assignments for ALU_Out <-- (($sp)-4)
       //--reads $sp directly from regfile
     $display("INTERRUPT REQUESTED");
     {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} = 8'b0_11_0_0_000;
     {dm_cs,  dm_rd,  dm_wr}                 = 3'b0_0_0;
     {io_cs,  io_rd,  io_wr}                 = 3'b0_0_0;
     {S_Sel, D_Sel} = 3'b 1_00;
     FS       = dec4;
     int_ack = 1'b0;
     state    = INTR_2;
     end

  INTR_2:
     @(negedge sys_clk) begin
     // dM[ALU_Out($sp)] <-- PC, $sp <-- ALU_Out($sp-4)
     {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} = 8'b1_11_0_0_000;
     {dm_cs,  dm_rd,  dm_wr}                 = 3'b1_0_1;
     {io_cs,  io_rd,  io_wr}                 = 3'b0_0_0;
     {S_Sel, D_Sel} = 3'b 1_01;
     FLAGS    = 5'b0;
     FS       = dec4;
     int_ack = 1'b0;
     state    = INTR_3;
     end

  INTR_3:
     @(negedge sys_clk) begin
     {pc_sel, pc_ld,  pc_inc, ir_ld}         = 5'b00_0_0_0;
     {im_cs,  im_rd,  im_wr}                 = 3'b0_0_0;
     {D_En,   DA_Sel, T_Sel, HILO_ld, Y_Sel} = 8'b1_11_0_0_000;
     {dm_cs,  dm_rd,  dm_wr}                 = 3'b1_0_1;
     {io_cs,  io_rd,  io_wr}                 = 3'b0_0_0;
     {S_Sel, D_Sel} = 3'b 1_01;
     FLAGS    = 5'b0;
     FS       = dec4;
     int_ack = 1'b0;
     state    = INTR_4;
    end

  INTR_4:
     @(negedge sys_clk) begin
     // R[ALU_Out($sp)] <-- {27'b0, FLAGS}, $sp <-- ALU_Out($sp-4)
     {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} = 8'b1_11_0_0_000;
     {dm_cs,  dm_rd,  dm_wr}                 = 3'b1_0_1;
     {io_cs,  io_rd,  io_wr}                 = 3'b0_0_0;
     {S_Sel, D_Sel} = 3'b 1_01;
     FLAGS    = 5'b0;
     FS       = dec4;
     int_ack = 1'b0;
     state    = INTR_5;
    end

  INTR_5:
     @(negedge sys_clk) begin
     {pc_sel, pc_ld,  pc_inc, ir_ld}         = 5'b00_0_0_0;
     {im_cs,  im_rd,  im_wr}                 = 3'b0_0_0;
     {D_En,   DA_Sel, T_Sel, HILO_ld, Y_Sel} = 8'b1_11_0_0_000;
     {dm_cs,  dm_rd,  dm_wr}                 = 3'b1_0_1;
     {io_cs,  io_rd,  io_wr}                 = 3'b0_0_0;
     {S_Sel, D_Sel} = 3'b 1_01;
     FLAGS    = 5'b0;
     FS       = dec4;
```

```
     int_ack = 1'b0;
     state   = INTR_6;
   end

 INTR_6:  // pc and flags saved at this point
     @(negedge sys_clk) begin
     {pc_sel, pc_ld,  pc_inc, ir_ld}          = 5'b00_0_0_0;
     {im_cs,  im_rd,  im_wr}                  = 3'b0_0_0;
     {D_En,   DA_Sel, T_Sel, HILO_ld, Y_Sel} = 8'b1_11_0_0_000;
     {dm_cs,  dm_rd,  dm_wr}                  = 3'b1_0_1;
     {io_cs,  io_rd,  io_wr}                  = 3'b0_0_0;
     {S_Sel, D_Sel} = 3'b 1_10;
     FS      = dec4;
     int_ack = 1'b0;
     state   = INTR_7;
   end

 INTR_7:    ///now steps to load PC with dM(3FC): PC <- dM[3FC]
     @(negedge sys_clk) begin
      //ALU_Out <- 0x3FC
     {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} = 8'b0_00_0_0_000;
     {dm_cs,  dm_rd,  dm_wr}                  = 3'b0_0_0;
     {io_cs,  io_rd,  io_wr}                  = 3'b0_0_0;
     {S_Sel, D_Sel} = 3'b 0_00;
     FLAGS   = 5'b0;
     FS      = sp_init;
     int_ack = 1'b0;
     state   = INTR_8;
   end

 INTR_8:
     @(negedge sys_clk) begin
      //control word assignments for D_in <- dM[ALU_Out(0x3FC)]
     {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} = 8'b0_11_0_0_000;
     {dm_cs,  dm_rd,  dm_wr}                  = 3'b1_1_0;
     {io_cs,  io_rd,  io_wr}                  = 3'b0_0_0;
     {S_Sel, D_Sel} = 3'b 0_00;
     FLAGS   = 5'b0;
     FS      = 5'b0;
     int_ack = 1'b0;
     state   = INTR_9;
   end

 INTR_9:
     @(negedge sys_clk) begin
      // PC <- D_in( dM[0x3FC] )
     {pc_sel, pc_ld,  pc_inc, ir_ld}          = 5'b10_1_0_0;
     {im_cs,  im_rd,  im_wr}                  = 3'b0_0_0;
     {D_En,   DA_Sel, T_Sel, HILO_ld, Y_Sel} = 8'b0_01_0_0_010;
     {dm_cs,  dm_rd,  dm_wr}                  = 3'b0_0_0;
     {io_cs,  io_rd,  io_wr}                  = 3'b0_0_0;
     {S_Sel, D_Sel} = 3'b 0_00; //S_Sel puts $29
     FLAGS   = 5'b0;
     FS      = 5'h0;
     int_ack = 1'b1;
     state   = FETCH;
   end

 RETI:  //Pops the Flags, then the PC from the Stack
```

```
      @(negedge sys_clk) begin
         //Flags <- M[$sp]
       //ALU_Out <- passS($sp)
      {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} = 8'b0_00_0_0_000;
      {dm_cs,  dm_rd,  dm_wr}                  = 3'b0_0_0;
      {io_cs,  io_rd,  io_wr}                  = 3'b0_0_0;
      {S_Sel, D_Sel} = 3'b 0_00;
      FLAGS    = 5'b0;
      FS       = 5'h0;
      int_ack = 1'b0;
      state    = RETI_2;
    end

   RETI_2:  //Pops the Flags, then the PC from the Stack
      @(negedge sys_clk) begin
      // Flags <- dM(Alu_Out(sp)) , ALUOut<- (sp+4)
      {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} = 8'b0_00_0_0_000;
      {dm_cs,  dm_rd,  dm_wr}                  = 3'b1_1_0;
      {io_cs,  io_rd,  io_wr}                  = 3'b0_0_0;
      {S_Sel, D_Sel} = 3'b 1_00;
      #1 {ns_i,   ns_c,   ns_v,   ns_n,    ns_z}   = {FLAGSIN};
        //the S_sel should be having (IE,cvnz) from the [$sp] in idp
      FS       = inc4;
      int_ack = 1'b0;
      state    = RETI_3;
    end

   RETI_3:  //Pops the Flags, then the PC from the Stack
      @(negedge sys_clk) begin
       // $sp <- ALU_Out ($sp+4)
      {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} = 8'b1_11_0_0_000;
      {dm_cs,  dm_rd,  dm_wr}                  = 3'b0_0_0;
      {io_cs,  io_rd,  io_wr}                  = 3'b0_0_0;
      {S_Sel, D_Sel} = 3'b 0_00;
      FS       = inc4;
      int_ack = 1'b0;
      state    = RETI_4;
    end
   RETI_4:  //Pops the Flags, then the PC from the Stack
      @(negedge sys_clk) begin
       // AluOut<- passS($sp)
      {pc_sel, pc_ld,  pc_inc, ir_ld}          = 5'b00_0_0_0;
      {im_cs,  im_rd,  im_wr}                  = 3'b0_0_0;
      {D_En,   DA_Sel, T_Sel, HILO_ld, Y_Sel} = 8'b1_11_0_0_000;
      {dm_cs,  dm_rd,  dm_wr}                  = 3'b1_1_0;
      {io_cs,  io_rd,  io_wr}                  = 3'b0_0_0;
      {S_Sel, D_Sel} = 3'b 0_00;
      FLAGS    = 5'b0;
      FS       = inc4;
      int_ack = 1'b0;
      state    = RETI_5;
    end
   RETI_5:  //Pops the Flags, then the PC from the Stack
      @(negedge sys_clk) begin
       // PC <- D_in ( dM[$sp] )
      {pc_sel, pc_ld,  pc_inc, ir_ld}          = 5'b10_1_0_0;
      {im_cs,  im_rd,  im_wr}                  = 3'b0_0_0;
      {D_En,   DA_Sel, T_Sel, HILO_ld, Y_Sel} = 8'b0_11_0_0_010;
      {dm_cs,  dm_rd,  dm_wr}                 = 3'b1_1_0;
      {io_cs,  io_rd,  io_wr}                 = 3'b0_0_0;
      {S_Sel, D_Sel} = 3'b 0_00;
      FLAGS   = 5'b0;
      FS      = inc4;
      int_ack = 1'b0;
      state   = RETI_6;
     end
   RETI_6:  //Pops the Flags, then the PC from the Stack
      @(negedge sys_clk) begin
       // $sp <- %sp+4
      {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} = 8'b1_11_0_0_000;
      {dm_cs,  dm_rd,  dm_wr}                 = 3'b0_0_0;
      {io_cs,  io_rd,  io_wr}                 = 3'b0_0_0;
      {S_Sel, D_Sel} = 3'b 0_00;
      FLAGS   = 5'b0;
      FS      = inc4;
      int_ack = 1'b0;
      state   = FETCH;
     end

   endcase  //end of FSM logic

   task Dump_Registers;
   begin
      $display("                    GBRAINS   R e g i s t e r   D u m p    ");
      $display("************************************************************");
      $display("   TIME        ||              Regfile T ($ri)    ||         ");
      $display("************************************************************");
      for(i = 0, j=0; i < 16; i = i + 1) begin
         j = i+16;
         @(negedge sys_clk) begin
         #1 $write("time: %t \t $r[%1d]: %h",
                         $time, i[4:0], CPU_Test.cpu.idp.regfile.registers[i]);
         #1 $display("    $r[%2d]: %h",
                         j[4:0], CPU_Test.cpu.idp.regfile.registers[i+16]);

         end
      end

      $display("                  GBRAINS  D o u b l e R e g i s t e r s    ");
      for(i = 0, j=0; i < 16; i = i + 1) begin
         j = i+16;
         @(negedge sys_clk) begin
         #1 $write("time: %t \t $f[%1d]: %h",
                         $time, i[4:0], CPU_Test.cpu.fdp.regfile.registers[i]);
         #1 $display("    $f[%2d]: %h",
                             j[4:0], CPU_Test.cpu.fdp.regfile.registers[i+16]);
         end
      end

      $display("The double at F[$01] is %f",
        $bitstoreal(CPU_Test.cpu.fdp.regfile.registers[1]) );
      $display("The double at F[$02] is %f",
        $bitstoreal(CPU_Test.cpu.fdp.regfile.registers[2]) );
      $display("The double at F[$03] is %f",
        $bitstoreal(CPU_Test.cpu.fdp.regfile.registers[3]) );
      $display("The double at F[$04] is %f",
        $bitstoreal(CPU_Test.cpu.fdp.regfile.registers[4]) );
      $display("The double at F[$05] is %f",
        $bitstoreal(CPU_Test.cpu.fdp.regfile.registers[5]) );
      $display("The double at F[$06] is %f",
        $bitstoreal(CPU_Test.cpu.fdp.regfile.registers[6]) );
      $display("The double at F[$07] is %f",
        $bitstoreal(CPU_Test.cpu.fdp.regfile.registers[7]) );

      $display("                  GBRAINS  V e c t o r R e g i s t e r s    ");
      for(i = 0, j=0; i < 16; i = i + 1) begin
         j = i+16;
         @(negedge sys_clk) begin
         #1 $write("time: %t \t $f[%1d]: %h",
                         $time, i[4:0], CPU_Test.cpu.vdp.regfile.registers[i]);
         #1 $display("    $f[%2d]: %h",
                             j[4:0], CPU_Test.cpu.vdp.regfile.registers[i+16]);
         end
      end


   end
   endtask

   task Dump_PC_and_IR;
      begin
         $display("time: %t \t PC: %h", $time, CPU_Test.cpu.iu.PCreg.PC_out);
         $display("time: %t \t IR: %h", $time, CPU_Test.cpu.iu.IRReg.Q);
      end
   endtask

   task Dump_Data_Memory;
      begin
      $display("                CECS 440   D a t a M e m o r y   D u m p    ");
      for(i = 8'hC0; i < 8'hFF; i=i+4) begin
      $display("time=%t DM[%1h]=%h", $time, i[8:0],
        {CPU_Test.dm.memory[i],
      CPU_Test.dm.memory[i+1],
        CPU_Test.dm.memory[i+2],
        CPU_Test.dm.memory[i+3]} );
      end

      $display("            CECS 440   W r i t t e n T o O n R e t u r n    ");
      $display("time=%t DM[%1h]=%h", $time, 12'h3F0,
        {CPU_Test.dm.memory[12'h3F0],
      CPU_Test.dm.memory[12'h3F1],
        CPU_Test.dm.memory[12'h3F2],
        CPU_Test.dm.memory[12'h3F3]} );

      $display("              CECS 440   S t a c k M e m o r y   D u m p    ");
      $display("time=%t DM[%1h]=%h", $time, 12'h3F4,
        {CPU_Test.dm.memory[12'h3F4],
      CPU_Test.dm.memory[12'h3F5],
        CPU_Test.dm.memory[12'h3F6],
        CPU_Test.dm.memory[12'h3F7]} );

      $display("time=%t DM[%1h]=%h", $time, 12'h3F8,
        {CPU_Test.dm.memory[12'h3F8],
      CPU_Test.dm.memory[12'h3F9],
        CPU_Test.dm.memory[12'h3FA],
        CPU_Test.dm.memory[12'h3FB]} );

      $display("time=%t DM[%1h]=%h", $time, 12'h3FC,
        {CPU_Test.dm.memory[12'h3FC],
      CPU_Test.dm.memory[12'h3FD],
        CPU_Test.dm.memory[12'h3FE],
         CPU_Test.dm.memory[12'h3FF]} );

      end
   endtask

    task Dump_IO_Memory;
      begin
      $display("                    CECS 440   IO M e m o r y   D u m p    ");
         for(i = 8'hC0; i < 8'hFF; i=i+4) begin
         $display("time=%t  IOM[%1h]=%h", $time, i[8:0],
         {CPU_Test.io.memory[i],
         CPU_Test.io.memory[i+1],
         CPU_Test.io.memory[i+2],
         CPU_Test.io.memory[i+3]} );
         end
      end
   endtask
endmodule
```

INSTRUCTION_UNIT

```
`timescale 1ns / 1ps
/*************************************************************************************
*
* Author(s): Brian Ortiz
*            Bryan Linares
*            Grace Daliwan
* Filename: INSTRUCTION_UNIT.v
* Date:     Oct. 25, 2018
* Project:  CECS 440 Lab 6
* Version:  1.1
*
* Notes:    Instruction Unit module, register file that holds the data
*           Writing to is synchronous, reading asynchronous.
*           Chip select (dm_cs) must asserted with dm_rd or dm_wr
*           simultaneously to read or write
*           Revision 10/25- Added PC_Mux
*
*************************************************************************************
module INSTRUCTION_UNIT( CLK, RESET, im_cs, im_wr, im_rd, pc_ld, pc_inc, ir_ld, PC_in,
                        pc_sel, PC_out, IR_out, SE_16);

    input         CLK, RESET;
    input         im_cs, im_wr, im_rd;
    input         pc_ld, pc_inc, ir_ld;
    input [ 1:0] pc_sel;
    input [31:0] PC_in;

    output [31:0] PC_out;
    output [31:0] IR_out;
    output [31:0] SE_16;

    wire [31:0] D_Out; //Instruction memory data out to IR register
    wire [31:0] PC_MUX;// for Instr. that adjust PC value

    PROGRAM_COUNTER PCreg   (CLK, RESET, pc_ld, pc_inc, PC_MUX, PC_out );
    DATA_MEMORY     IMemReg (CLK, im_cs, im_wr, im_rd, PC_out[11:0], 32'h0, D_Out);
    REG32           IRReg   (CLK, RESET, ir_ld, D_Out, IR_out );

    assign SE_16 = {{16{IR_out[15]}},IR_out[15:0]}; //sign ext. imm from instruction

    assign PC_MUX = (pc_sel == 2'h2)?  PC_in:
                    (pc_sel == 2'h1)? {PC_out[31:28], IR_out[25:0], 2'b00}://jump addr
                                       PC_out + {SE_16[29:0], 2'b00};      //branch
endmodule
```

INTEGER_DATAPATH

```
`timescale 1ns / 1ps
/*************************************************************************************
* Author(s):Bryan Linares
*           Grace Daliwan
*           Brian Ortiz
* Filename: INTEGER_DATAPATH.v
* Date:     Nov. 27, 2018
* Project:  CECS 440 Senior Project
* Version:  1.14
*
* Notes:    Integer Datapath module, routes data from outside sources to
*           destinations in execution phase of instruction cycle.
*           Instantiates the 32x32 Register file and ALU that performs operations.
*           Revision 10/9 - Added pipeline Registers for S,T,ALU_Out,D_in
*           Revision 10/16- Added DA_sel for selecting D_Addr from S_Addr field
*           Revision 10/25- Added additional DA mux selections, expanded DA_Sel
*           Revision 11/18- Added Shifting Amount input to Shifter in ALU
*           Revision 11/20- Added FLAGS input and FLAGS_Out for receiving and
*                           outputting flags register on interrupt states Also added
*                           S_Sel to set rs=$sp and D_Sel, for loading PC and flags
*                           in interrupt
*
*************************************************************************************/
module INTEGER_DATAPATH( CLK, RESET, FS, HILO_ld, D_En, D_Addr, S_Addr, T_Addr, DT,
               T_Sel, PC_in, SHAMT,DA_sel, C, V, N, Z, DY, Y_Sel, ALU_OUT, D_OUT,
               S_Sel, D_Sel, FLAGS, FLAGS_OUT, LONG_OUT);

   input CLK;
   input RESET;
   input [4:0] FS;     //ALU Function Select
   input HILO_ld;      //Load Mul/Dev result registers
   input D_En;         //D Register write enable
   input T_Sel;        //Select T input for ALU, either T from RegFile or DT
   input S_Sel;        //S_Sel alt for  $sp
   input [31:0] DT;    //External T value for ALU
   input [31:0] DY;    //External Y value for Register D input
   input [31:0] PC_in; //External PC value In

   input [4:0] FLAGS;  //present flags register from MCU
   input [4:0] D_Addr; //D Reg Address
   input [4:0] S_Addr; //S Reg Address
   input [4:0] T_Addr; //T Reg Address
   input [4:0] SHAMT;  //Shifting amount

   input [2:0] Y_Sel;  //ALU_Out select 1-5: HI,LO,Y_lo, DY, PC_in
   input [1:0] DA_sel, D_Sel; //Select alternate Destination Address/DataOut
                              //for alt. IR format and Data.
   output        C,V,N,Z;
   output [31:0] D_OUT;
   output [31:0] ALU_OUT;
   output [4:0]  FLAGS_OUT; //to Flags register in MCU
   output [63:0] LONG_OUT;   //64 bit out to Enhanced datpaths

   wire [4:0]  Y_Mux, D_Mux, S_Mux;
   wire [31:0] S, T, T_Reg, T_Out, Y_hi, Y_lo, ALU_lo, HI_out, LO_out, T_Mux;
   wire [31:0] HI, LO, RS, RT, ALU_OutReg, D_in;
   reg  [31:0] Y;

   REG32 HIReg  (.CLK(CLK), .RESET(RESET), .ld(HILO_ld), .D(Y_hi),  .Q(HI) );
   REG32 LOReg  (.CLK(CLK), .RESET(RESET), .ld(HILO_ld), .D(Y_lo),  .Q(LO) );
   REG32 RSReg  (.CLK(CLK), .RESET(RESET), .ld(1'b1),    .D(S),     .Q(RS) );
   REG32 RTReg  (.CLK(CLK), .RESET(RESET), .ld(1'b1),    .D(T_Mux), .Q(RT) );
   REG32 ALUReg (.CLK(CLK), .RESET(RESET), .ld(1'b1),    .D(Y_lo),  .Q(ALU_OutReg));
   REG32 DinReg (.CLK(CLK), .RESET(RESET), .ld(1'b1),    .D(DY),    .Q(D_in) );

   REGFILE32 regfile (
      .CLK(CLK), .RESET(RESET), .D_Addr(D_Mux),
      .S_Addr(S_Mux), .T_Addr(T_Addr), .D_EN(D_En), .D(ALU_OUT),  //inputs
      .S(S), .T(T)                                               //outputs
   );

   ALU_32 alu_ver1 (
    .S(RS), .T(RT), .SHAMT(SHAMT),.FS(FS),                        // inputs
    .Y_hi(Y_hi),    .Y_lo(Y_lo), .C(C), .V(V), .N(N), .Z(Z)      // outputs
   );

   assign LONG_OUT = {RS,RT}; ///64 bit output to enhanced datapaths

   //DA-Mux, destination address mux
   assign D_Mux = (DA_sel == 2'h3)? 5'h1D:  //29 sp
                  (DA_sel == 2'h2)? 5'h1F:  //31 ra
                  (DA_sel == 2'h1)? T_Addr: //IR[20:16]
                                    D_Addr; //IR[15:11]

   // Y-Mux, decides which register is output on ALU_Out/Address
   assign ALU_OUT = (Y_Sel == 3'h4) ? HI:
                    (Y_Sel == 3'h3) ? LO:
                    (Y_Sel == 3'h0) ? ALU_OutReg:
                    (Y_Sel == 3'h2) ? D_in:
                    (Y_Sel == 3'h1) ? PC_in: ALU_OutReg; //defaults to ALU

   //T-Mux, decides whether T is loaded from external immediate or T from regfile
   assign T_Mux = T_Sel ? DT : T;

   //S-Mux, when asserted, sets the regfile S_Addr input to $sp. Used in interrupt
   //This is used in INTR but not needed in RETI since rs has 1D
   assign S_Mux = S_Sel ? 5'h1D : S_Addr;

   //Flags to be read from bottom 5 bits of dM[$sp] input in interrupt return
   assign FLAGS_OUT = S_Sel? DY[4:0] : 5'hX;

                                     //'F1A9S' here as marker
   assign D_OUT = (D_Sel == 2'h2)? {24'hF1A950,3'b000, FLAGS}: //flag output from MCU
                  (D_Sel == 2'h1)? PC_in:                     //data output is raw PC
                                   RT;                       //default data from RT
endmodule
```

FLOATINGPOINT_DATAPATH

```
`timescale 1ns / 1ps
/**************************************************************************************
* Author(s): Brian Ortiz
*            Bryan Linares
*            Grace Daliwan
* Filename:  FLOATINGPOINT_DATAPATH.v
* Project:   CECS 440 Senior Project GBRAINS
*
**************************************************************************************/
module FLOATINGPOINT_DATAPATH( CLK, RESET, D_EN, FS, D_Addr, S_Addr, T_Addr, FMT, DT,
        DY, T_Sel, DIN_Sel, DOut_Sel, Y_Sel, D_OUT  );

   input CLK, RESET;
   input [4:0] FS;      //ALU Function Select

   input D_EN;          //D Register write enable
   input [4:0] D_Addr;  //D Reg Address
   input [4:0] S_Addr;  //S Reg Address
   input [4:0] T_Addr;  //T Reg Address
   input [4:0] FMT;     //*Format specifier, for future single/double precision select
   input [63:0] DT;     //64 bit External Tvalue for ALU, gets Long Imm from IDPR
   input T_Sel,         //Select T input for ALU, either T from RegFile or DT,
         DIN_Sel,       //select which Din reg to load HI or LO, LO is 0
         DOut_Sel;      // choose hi or lo 32bit half of data to come out
   input [31:0] DY;     //External Y value for Register File D input
   input Y_Sel;         //Select Y that goes into regfile from ALU or Memory

   output [31:0] D_OUT;

   wire [63:0] Y, S, T, T_Mux, Y_Mux;
   wire [31:0] Din_HI, Din_LO, Y_hi, Y_lo;
   wire [4:0] DA_Mux;


   //two REG32 Y_lo, Y_hi always loaded, but muxed on output  depending on mcu input
   REG32 Y_HIREG(.CLK(CLK), .RESET(RESET), .ld(1'b1),    .D(Y[63:32]),     .Q(Y_hi));
   REG32 Y_LOREG(.CLK(CLK), .RESET(RESET), .ld(1'b1),    .D(Y[31:0]),      .Q(Y_lo));

   //DIN_Sel selects hi or lo to data in buffers to load, gets from memory
   REG32 DIN_HIREG(.CLK(CLK), .RESET(RESET), .ld(DIN_Sel),  D(DY), .Q(Din_HI));
   REG32 DIN_LOREG(.CLK(CLK), .RESET(RESET), .ld(~DIN_Sel),.D(DY), .Q(Din_LO));

   //Regfile64 32 registers, 64 bits wide
   REGFILE64 regfile (.CLK(CLK), .RESET(RESET),
                     .D_Addr(D_Addr), .S_Addr(S_Addr), .T_Addr(T_Addr),
                     .D_EN(D_EN),     .D(Y_Mux),       .S(S),    .T(T));

//FPALU
   FPALU_64 fpalu ( .S(S), .T(T_Mux), .FS(FS), .Y(Y) );

//TMux, select ALU T from regfile or external IDP Long IDP Regfile Immediate, 64bit,
//used for immediates from IDP regfile
   assign T_Mux = T_Sel? DT : T;

   //Y_Mux, for ALU or external data buffers into RegFile data in
   assign Y_Mux = Y_Sel? {Din_HI, Din_LO} : Y;

   //D_OutMux, select hi or lo reg to come out on 32 bit data line
   assign D_OUT = DOut_Sel ? Y_hi : Y_lo;

endmodule
```

VECTOR_DATAPATH

```
`timescale 1ns / 1ps
/*******************************************************************************
* Author(s): Brian Ortiz
*            Bryan Linares
*            Grace Daliwan
* Filename:  VECTOR_DATAPTH.v
* Project:   CECS 440 Senior Project GBRAINS
*
* Notes:     Integer Datapath module, routes data from outside sources to
*            destinations in execution phase of instruction cycle.
*            Instantiates the 64x64 Register file and ALU that performs operations.
*
*******************************************************************************/

module VECTOR_DATAPATH(CLK, RESET, D_EN, FS, D_Addr, S_Addr, T_Addr, FMT, DT,
        DY, T_Sel, DIN_Sel, DOut_Sel, Y_Sel, D_OUT  );

   input CLK, RESET;
   input [4:0] FS;          //ALU Function Select

   input D_EN;              //D Register write enable
   input [4:0] D_Addr;      //D Reg Address
   input [4:0] S_Addr;      //S Reg Address
   input [4:0] T_Addr;      //T Reg Address
   input [4:0] FMT;         //**Format specifier, mostly unused in this edition VDP,
                            //for future packed data size select, now just for LW
   input [63:0] DT;         //32 bit External Tvalue for ALU, gets data out value
                            //from IDP
   input T_Sel,             //Select T input for ALU, either T from RegFile or DT,
         DIN_Sel,           //select which Din reg to load HI or LO, LO is 0
         DOut_Sel;          //choose hi or lo 32bit half of data to come out
   input [31:0] DY;         //External Y value for Register File D input
   input Y_Sel;             //Select Y that goes into regfile from ALU or Memory

   output [31:0] D_OUT;

   wire [63:0] Y, S, T, T_Mux, Y_Mux, Dreg;
   wire [31:0] Din_HI, Din_LO, Y_hi, Y_lo;
   wire [4:0] DA_Mux;

   //two REG32 Y_lo, Y_hi always loaded, but muxed on output  depending on mcu input
   REG32 Y_HIREG(.CLK(CLK), .RESET(RESET), .ld(1'b1), .D(Y[63:32]), .Q(Y_hi));
   REG32 Y_LOREG(.CLK(CLK), .RESET(RESET), .ld(1'b1), .D(Y[31:0]),  .Q(Y_lo));

   //DIN_Sel selects hi or lo to data in buffers to load,
   REG32 DIN_HIREG(.CLK(CLK), .RESET(RESET), .ld(DIN_Sel),  .D(DY),  .Q(Din_HI));
   REG32 DIN_LOREG(.CLK(CLK), .RESET(RESET), .ld(~DIN_Sel), .D(DY),  .Q(Din_LO));

   //Regfile64 32 registers, 64 bits wide, VREGFILE64, three outputs
   VREGFILE64 regfile (.CLK(CLK), .RESET(RESET),
   .D_Addr(D_Addr), .S_Addr(S_Addr), .T_Addr(T_Addr), .D_EN(D_EN),
   .D(Y_Mux),       .S(S),           .T(T),           .DOUT(Dreg));


  //VALU, takes in three  data inputs, one of which is D, which will get overwritten
   VALU_64 valu (.S(S), .T(T_Mux), .D(Dreg), .FS(FS), .FMT(FMT), .Y(Y) );

   //TMux, select ALU T from regfile or external IDP Reg Immediate,
   //64bit, used for immediate stores from IDP regfile
   assign T_Mux = T_Sel? DT : T;
   //Y_Mux, for ALU or external data buffers into RegFile data in
   assign Y_Mux = Y_Sel? {Din_HI, Din_LO} : Y;

   //D_OutMux, select hi or lo reg to come out on 32 bit data line
   assign D_OUT = DOut_Sel ? Y_hi : Y_lo;

endmodule
```

DATA_MEMORY

```
`timescale 1ns / 1ps
/*****************************************************************************
* Author(s):Bryan Linares
*           Brian Ortiz
*           Grace Daliwan
* Filename: DATA_MEMORY.v
* Date:     Oct. 9, 2018
* Project:  CECS 440 Lab 4
* Version:  1.0
*
* Notes:    Data Memory module, register file that holds the data
*           Writing to is synchronous, reading asynchronous.
*           Chip select (dm_cs) must asserted with dm_rd or dm_wr
*           simultaneously to read or write
*
*****************************************************************************/
module DATA_MEMORY( clk, dm_cs, dm_wr, dm_rd, Address, D_in, D_Out );

   input  clk, dm_cs, dm_wr, dm_rd; //Enables: chip select, write, read
   input  [11:0] Address;
   input  [31:0] D_in;
   output [31:0] D_Out;

   reg [7:0] memory[0:4095]; //big endian 4096x8 byte addressable

   //synchronous writes
   always@(posedge clk)
      if(dm_cs & dm_wr)
                {memory[Address + 0],memory[Address + 1],
                 memory[Address + 2],memory[Address + 3]} <= D_in;

   //asynchronous reading
   assign D_Out = (dm_cs & dm_rd)?
                   {memory[Address + 0],memory[Address + 1],
                    memory[Address + 2],memory[Address + 3]}
                   : 32'hZ;
endmodule
```

IO_MODULE

```
`timescale 1ns / 1ps
/****************************************************************************************
* Author(s):Bryan Linares
*           Grace Daliwan
*           Brian Ortiz
* Filename: IO_Module.v
* Date:     Nov. 17, 2018
* Project:  CECS 440 Lab 4
*
* Notes:    IO Memory module, register file that holds the data.
*           Writing to is synchronous, reading asynchronous.
*           Chip select (iom_cs) must be asserted with iom_rd or iom_wr
*           simultaneously to read or write
*
****************************************************************************************/
module IO_Module( clock, io_cs, io_rd, io_wr, Address, int_ack, intr, IO_in, IO_out);

   input         clock, int_ack;
   input         io_cs, io_wr, io_rd; //Enables: chip select, write, read
   input  [11:0] Address;
   input  [31:0] IO_in;
   output reg     intr;
   output [31:0] IO_out;

   reg [7:0] memory[0:4095]; //big endian 4096x8 byte addressable

   initial begin
      intr = 0;
      #1000 intr = 1;
      @(posedge int_ack) intr = 0;
   end

   //synchronous writes
   always@(posedge clock)
      if(io_cs & io_wr)
               {memory[Address + 0], memory[Address + 1],
                memory[Address + 2], memory[Address + 3]} <= IO_in;

     else begin
         memory[Address+0] = memory[Address+0];
         memory[Address+1] = memory[Address+1];
         memory[Address+2] = memory[Address+2];
         memory[Address+3] = memory[Address+3];
         end

   //asynchronous reading
   assign IO_out = (io_cs & io_rd)?
                  {memory[Address + 0], memory[Address + 1],
                   memory[Address + 2], memory[Address + 3]}
                  : 32'hZ;


endmodule
```

INTEGER_ALU

```
`timescale 1ns / 1ps
/****************************************************************
* Author:   Bryan Linares
*           Brian Ortiz
*           Grace Daliwan
* Filename: ALU_32.v
* Date:     Sep. 11, 2018
* Project:  CECS 440 Lab 1
* Version:  1.0
*
* Notes:    32 bit ALU wrapper module for MIPS ISA.
*           FS is 5 bit function select input. Input
*           operands S and T . C,V,N,Z, Y_hi,Y_lo outputs
***************************************************************/
module ALU_32( S, T, SHAMT, FS, Y_hi, Y_lo, C, V, N, Z);

   input [4:0]  FS, SHAMT;
   input [31:0] S;
   input [31:0] T;

   output        C, V, N, Z;
   output [31:0] Y_hi;
   output [31:0] Y_lo;

   wire mips_c, bs_c;
   wire [31:0] Y, mpy_hi, mpy_lo, div_rem, div_quot, mips_y, bs_y;
   wire [63:0] mpy_product;

   MIPS_32 mips ( S, T, FS, V, mips_c, mips_y);

   MPY_32 mul   ( S, T, mpy_product);

   DIV_32  div  ( S, T, div_rem, div_quot);

   Barrel_Shifter bs ( .FS(FS), .SHAMT(SHAMT), .T(T), .SHFT_OUT(bs_y), .C(bs_c));

   assign mpy_hi = mpy_product[63:32];
   assign mpy_lo = mpy_product[31:0];

   assign {Y_hi, Y_lo, N} = (FS == 5'h1E) ? {mpy_hi, mpy_lo, mpy_hi[31]}:
                            (FS == 5'h1F) ? {div_rem, div_quot, div_quot[31]}:
                            (FS == 5'h0C ||
                             FS == 5'h0D ||
                             FS == 5'h0E) ? {32'b0, bs_y, bs_y[31]}:
                                            {32'b0, mips_y, mips_y[31]};

   assign C = (FS == 5'h1e) ? 1'bx:
              (FS == 5'h1f) ? 1'bx:
              (FS == 5'h0C ||
               FS == 5'h0D ||
               FS == 5'h0E) ? bs_c:
                             mips_c;

   assign Z = ((Y_hi == 16'h0) && (Y_lo == 16'h0)) ? 1'b1 : 1'b0;


endmodule
```

FLOATINGPOINT_ALU

```
`timescale 1ns / 1ps
/****************************************************************************
*Author(s): Brian Ortiz
*           Bryan Linares
*           Grace Daliwan
* Filename: FPALU.v
* Project:  CECS 440 Senior Project GBRAINS
*
****************************************************************************/
module FPALU_64( S, T, FS, Y);

input [63:0] S, T;
input [4:0] FS;

output reg [63:0] Y;

parameter PASS_S = 5'h00, PASS_T = 5'h01, ADD   = 5'h02,  SUB = 5'h03,
          MULT   = 5'h1E, DIV  = 5'h1F,   ZERO  = 5'h13;

real fpY, fpS, fpT;

   always @(*) begin
   fpS = $bitstoreal(S);
   fpT = $bitstoreal(T);

   case (FS)
   PASS_S: fpY = fpS;       // pass S
   PASS_T: fpY = fpT;       // pass T
   ADD:    fpY = fpS + fpT; // Addition
   SUB:    fpY = fpS - fpT; // Subtraction R-S
   MULT:   fpY = fpS * fpT; // Multiply
   DIV:    fpY = fpS / fpT; // Division S/T
   ZERO:   fpY = 0.0;       // zero
   default: fpY = 64'hx;
   endcase

  Y = $realtobits(fpY);
   end
endmodule
```

VECTOR_ALU

```
`timescale 1ns / 1ps
/**************************************************************************************
* Author(s): Brian Ortiz
*            Bryan Linares
*            Grace Daliwan
*
* Filename: VALU_64.v
* Project:  CECS 440 Senior Project GBRAINS
*
* Credit: Based on functions found in the AltiVec Technology Programming
*         Interface Manual
* Notes:  Performs vectored integer operations on combined 32-bit values passed in on
*         three 64bit inputs, outputting to one 64-bit output.
*
**************************************************************************************/
module VALU_64(S, T, D, FS, FMT, Y);

   input  [63:0] S, T, D;
   input  [ 4:0] FS, FMT;

   output reg [63:0] Y;

   reg    [ 7:0] carry;

   parameter
   ADDS   = 5'h08,    MULADD  = 5'h09,   ANDEI  = 5'h02,
   VCMPE   = 5'h06,   VCLT    = 5'h07,   PASS_S = 5'h00,
   PASS_T = 5'h01;

   //bits to integer, so verilog math operators work
    integer S_hi, S_lo,D_hi;
    integer T_hi, T_lo,D_lo;
    integer int_d;

   always @ (*) begin

      //Split inputs for easier indexing
      S_hi = S[63:32]; S_lo = S[31: 0];
      T_hi = T[63:32]; T_lo = T[31: 0];
      D_hi = D[63:32]; D_lo = D[31: 0];
      int_d = D;
      carry = 8'b0;

      case(FS)

         ADDS: //ADD SATURATED 8 Bit Signed
         begin
            {carry[0], Y[ 7: 0]} = S[ 7: 0] + T[ 7: 0];
            {carry[1], Y[15: 8]} = S[15: 8] + T[15: 8];
            {carry[2], Y[23:16]} = S[23:16] + T[23:16];
            {carry[3], Y[31:24]} = S[31:24] + T[31:24];
            {carry[4], Y[39:32]} = S[39:32] + T[39:32];
            {carry[5], Y[47:40]} = S[47:40] + T[47:40];
            {carry[6], Y[55:48]} = S[55:48] + T[55:48];
            {carry[7], Y[63:56]} = S[63:56] + T[63:56];

            Y[ 7: 0] = (carry[0]) ? 8'hFF: Y[ 7: 0]; //if the sum had a carry,
            Y[15: 8] = (carry[1]) ? 8'hFF: Y[15: 8]; //this clamps the value to
            Y[23:16] = (carry[2]) ? 8'hFF: Y[23:16]; //the max within the 8 bits: 8'FF
            Y[31:24] = (carry[3]) ? 8'hFF: Y[31:24];
            Y[39:32] = (carry[4]) ? 8'hFF: Y[39:32];
            Y[47:40] = (carry[5]) ? 8'hFF: Y[47:40];
            Y[55:48] = (carry[6]) ? 8'hFF: Y[55:48];
            Y[63:56] = (carry[7]) ? 8'hFF: Y[63:56]; //over carry drops off

         end

         MULADD: //Multiply and Add 32 BIT Signed, Multiplies the 32 bit
                 //integers in S and T, then adds 64 bit D
         begin
            Y[ 31: 0] = (S_lo * T_lo ) + int_d;
            Y[ 63:32] = (S_hi * T_hi ) + int_d;
         end

         ANDEI:   //AND Unsigned 8 bit Integers, ands every 8 bit in the operands
         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];
            Y[39:32] = S[39:32] & T[39:32];
            Y[47:40] = S[47:40] & T[47:40];
            Y[55:48] = S[55:48] & T[55:48];
            Y[63:56] = S[63:56] & T[63:56];
         end

         VCMPE:   //Vectored 8 BIT Compare if Equal,8 bit element
                  //in S is equal to  parallel element in T
         begin
            Y[ 7: 0] = (S[ 7: 0]==T[ 7: 0]) ? 8'hFF: 8'b0;
            Y[15: 8] = (S[15: 8]==T[15: 8]) ? 8'hFF: 8'b0;
            Y[23:16] = (S[23:16]==T[23:16]) ? 8'hFF: 8'b0;
            Y[31:24] = (S[31:24]==T[31:24]) ? 8'hFF: 8'b0;
            Y[39:32] = (S[39:32]==T[39:32]) ? 8'hFF: 8'b0;
            Y[47:40] = (S[47:40]==T[47:40]) ? 8'hFF: 8'b0;
            Y[55:48] = (S[55:48]==T[55:48]) ? 8'hFF: 8'b0;
            Y[63:56] = (S[63:56]==T[63:56]) ? 8'hFF: 8'b0;
         end

         VCLT:   //compare if less than, Compares if 8 bit element in S is
                 //less than parallel element in T
         begin
            Y[ 7: 0] = (S[ 7: 0]<T[ 7: 0]) ? 8'hFF: 8'b0;
            Y[15: 8] = (S[15: 8]<T[15: 8]) ? 8'hFF: 8'b0;
            Y[23:16] = (S[23:16]<T[23:16]) ? 8'hFF: 8'b0;
            Y[31:24] = (S[31:24]<T[31:24]) ? 8'hFF: 8'b0;
            Y[39:32] = (S[39:32]<T[39:32]) ? 8'hFF: 8'b0;
            Y[47:40] = (S[47:40]<T[47:40]) ? 8'hFF: 8'b0;
            Y[55:48] = (S[55:48]<T[55:48]) ? 8'hFF: 8'b0;
            Y[63:56] = (S[63:56]<T[63:56]) ? 8'hFF: 8'b0;
         end

         PASS_S:  Y = S;   //PASS S
         PASS_T:  Y = T;   //PASS T used when receiving val from IDP,
                           //to direct into Regfile
         default: Y = T;

     endcase
   end

endmodule
```

INTEGER_REGISTER_FILE

```
`timescale 1ns / 1ps
/****************************************************************************
* Author:   Bryan Linares
*           Grace Daliwan
*           Brian Ortiz
* Filename: REGFILE32.v
* Date:     Sep. 20, 2018
* Project:  CECS 440 Lab 2
* Version:  1.0
*
* Notes:    32 bit wide, 32 bits deep register file module for MIPS ISA.
*           Contains the user registers for a given processor.
*           Contains 32 general registers, each 32 bits wide.
*           registers[0] (mips $r0) is read only. Always has value 0 (Zero)
*
*****************************************************************************/
module REGFILE32(CLK, RESET, D_Addr, S_Addr, T_Addr, D_EN, D, S, T);

   input CLK;
   input RESET;

   input       D_EN;
   input [4:0] D_Addr;
   input [4:0] S_Addr;
   input [4:0] T_Addr;
   input [31:0] D;

   output [31:0] S;
   output [31:0] T;

   reg [31:0] registers [31:0];

//Write Section - synchronous on posedge clock and reset signals
   always@(posedge CLK, posedge RESET)
      if(RESET)
         registers[0] <= 32'b0;
      else
         if(D_EN)
            registers[D_Addr] <= (D_Addr == 5'b0) ? registers[D_Addr] : D;
         //otherwise, registers doesn't change

//Read Section  - asynchronous, continuous assign statements
   assign S = registers[S_Addr];
   assign T = registers[T_Addr];

endmodule
```

FLOATINGPOINT_REGISTER_FILE

```
`timescale 1ns / 1ps
/******************************************************************************
* Author(s): Brian Ortiz
*            Bryan Linares
*            Grace Daliwan
* Filename:  REGFILE64.v
* Project:   CECS 440 Senior Project GBRAINS
*
******************************************************************************/
module REGFILE64(CLK, RESET, D_Addr, S_Addr, T_Addr, D_EN, D, S, T);

   input CLK;
   input RESET;

   input       D_EN;
   input [4:0] D_Addr;
   input [4:0] S_Addr;
   input [4:0] T_Addr;
   input [63:0] D;

   output [63:0] S;
   output [63:0] T;

   reg [63:0] registers [31:0];

//Write Section - synchronous on posedge clock and reset signals
   always@(posedge CLK, posedge RESET) //can write to any register
         if(D_EN)
            registers[D_Addr] <= D;
         //otherwise, registers don't change

//Read Section  - asynchronous, continuous assign statements
   assign S = registers[S_Addr];
   assign T = registers[T_Addr];

endmodule
```

VECTOR_REGISTER_FILE

```
`timescale 1ns / 1ps
/*********************************************************************************
* Author(s): Brian Ortiz
*            Bryan Linares
*            Grace Daliwan
* Filename: VREGFILE64.v
* Project:  CECS 440 Senior Project GBRAINS
*
*********************************************************************************/
module VREGFILE64(CLK, RESET, D_Addr, S_Addr, T_Addr, D_EN, D, S, T, DOUT);

   input CLK;
   input RESET;

   input       D_EN;
   input [4:0] D_Addr;
   input [4:0] S_Addr;
   input [4:0] T_Addr;
   input [63:0] D;

   output [63:0] S;
   output [63:0] T;
   output [63:0] DOUT;

   reg [63:0] registers [31:0];

//Write Section - synchronous on posedge clock and reset signals
   always@(posedge CLK, posedge RESET) //can write to any register
         if(D_EN)
            registers[D_Addr] <= D;
         //otherwise, registers don't change

//Read Section  - asynchronous, continuous assign statements
   assign S    = registers[S_Addr];
   assign T    = registers[T_Addr];
   assign DOUT = registers[D_Addr];
endmodule
```

PROGRAM_COUNTER

```
`timescale 1ns / 1ps
/*******************************************************************
* Author(s):Bryan Linares
*           Grace Daliwan
*           Brian Ortiz
* Filename: PROGRAM_COUNTER.v
* Date:     Oct. 16, 2018
* Project:  CECS 440 Lab 5
* Version:  1.0
*
* Notes:    Program Counter module, register that holds the PC
*           Can be loaded and incremented by value of 4.
*           pc_ld active-hi loads the Reg and pc_inc counts up.
*           PC reg is 32 bits wide.
*
*******************************************************************/
module PROGRAM_COUNTER( CLK, RESET, pc_ld, pc_inc, PC_in, PC_out );

   input CLK, RESET;
   input pc_ld, pc_inc;
   input [31:0] PC_in;
   output reg [31:0] PC_out;

   always@(posedge CLK, posedge RESET)
   if(RESET)
      PC_out <= 32'h0;
   else
      begin
         case({pc_inc,pc_ld})
            2'b01: PC_out <= PC_in;
            2'b10: PC_out <= PC_out + 4;
         default: PC_out <= PC_out;
         endcase
      end
endmodule
```

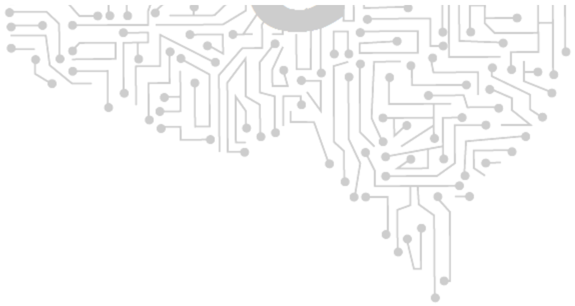
MIPS_32BIT

```
`timescale 1ns / 1ps
/************************************************************
* Author:   Bryan Linares
*           Brian Ortiz
*           Grace Daliwan
* Filename: MIPS_32.v
* Date:     Sep. 11, 2018
* Project:  CECS 440 Lab 1
* Version:  1.0
*
* Notes:    32 bit ALU operations module for MIPS ISA
*
************************************************************/

module MIPS_32( S, T, FS, V, C, Y );

   input [31:0] S;
   input [31:0] T;
   input [4:0] FS;

   output reg V;
   output reg C;
   output reg [31:0] Y;

   //Symbolic Constants for Operations,
   parameter PASS_S = 5'h00, PASS_T  = 5'h01, ADD  = 5'h02, SUB    = 5'h03,
             ADDU   = 5'h04, SUBU    = 5'h05, SLT  = 5'h06, SLTU   = 5'h07,
             AND    = 5'h08, OR      = 5'h09, XOR  = 5'h0A, NOR    = 5'h0B,
             SLL    = 5'h0C, SRL     = 5'h0D, SRA  = 5'h0E, ANDI   = 5'h16,
             ORI    = 5'h17, LUI     = 5'h18, XORI = 5'h19, INC    = 5'h0F,
             DEC    = 5'h10, INC4    = 5'h11, DEC4 = 5'h12, ZEROS  = 5'h13,
             ONES   = 5'h14, SP_INIT = 5'h15 ;

   always @ (*) begin
   {C, V} = {1'bX, 1'bX}; //If flag unaffected set to X
   case(FS)

   PASS_S: begin
      Y = S;
      {C, V} = {1'bX, 1'bX};
      end
   PASS_T: begin
      Y = T;
      {C, V} = {1'bX, 1'bX};
      end
   ADD: begin
      {C, Y} = S + T;
      if((S[31] == 1'b1) && (T[31] == 1'b1))
         V = (Y[31]) ? 1'b0 : 1'b1;
      if((S[31] == 1'b0) && (T[31] == 1'b0))
         V = (Y[31]) ? 1'b1 : 1'b0;
      else
      V = 1'b0;
      end
   SUB: begin
      {C, Y} = S - T;
      if((S[31] == 1'b0) && (T[31] == 1'b1))
         V = (Y[31]) ? 1'b1 : 1'b0;
      if((S[31] == 1'b1) && (T[31] == 1'b0))
         V = (Y[31]) ? 1'b0 : 1'b1;
      else
      V = 1'b0;
   end
ADDU: begin
   {C, Y} = S + T;
   if((Y < S) && (Y < T)) //if Carry 1, Overflowed
      V = 1'b1;
   else
      V = 1'b0;
   end
SUBU: begin
   {C, Y} = S - T;
   if(S < T)
      V = 1'b1;
   else
      V = 1'b0;
   end
SLT: begin
     Y = S - T;
     Y = (Y[31] == 1'b1) ? 1'b1 : 1'b0;
     {C, V} = {2'bXX};
     end
SLTU: begin
      Y = (S < T) ? 1'b1 : 1'b0;
      {C, V} = {2'bXX};
      end
AND: begin
      Y = S & T;
      {C, V} = {2'bXX};
      end
OR: begin
      Y = S | T;
      {C, V} = {2'bXX};
      end
XOR: begin
      Y = S ^ T;
      {C, V} = {2'bXX};
      end
NOR: begin
      Y = ~(S | T);
      {C, V} = {2'bXX};
      end
SLL: begin
      {C, Y} = T << 1;
      V = 1'bx;
      end
SRL: begin
      {C, Y} = {T[0], T >> 1};
      V = 1'bx;
      end
SRA: begin
      {C, Y} = {T[0], T[31], T[31:1]};
      V = Y[31] ^ T[31];
      end
ANDI: begin
      Y = S & {16'h0, T[15:0]};
      end
ORI: begin
      Y = S | {16'h0, T[15:0]};
      end
LUI: begin
      Y = {T[15:0], 16'h0};
      end
XORI: begin
```

```
            Y = S ^ {16'h0, T[15:0]};
            end
      INC: begin
            {C, Y} = S + 1;
            if(S[31] == 1'b0)
            V = (Y[31]) ? 1'b1 : 1'b0;
            else
            V = 1'b0;
            end
      DEC: begin
            {C, Y} = S - 1;
            if(S[31] == 1'b0)
            V = (Y[31]) ? 1'b1 : 1'b0;
            else
            V = 1'b0;
            end
      INC4: begin
            {C, Y} = S + 4;
            if(S[31] == 1'b0)
            V = (Y[31]) ? 1'b1 : 1'b0;
            else
            V = 1'b0;
            end
      DEC4: begin
            {C, Y} = S - 4;
            if(S[31] == 1'b0)
            V = (Y[31]) ? 1'b1 : 1'b0;
            else
            V = 1'b0;
            end
      ZEROS: begin
             Y = 32'h0;
             end
      ONES: begin
            Y = 32'hFFFFFFFF;
            end
      SP_INIT: begin
            Y = 32'h3FC;
            end
      default: begin //pass Source operand on default
            Y = S;
            {C, V} = {2'bXX};
            end
      endcase
      end // end of always
   endmodule
```

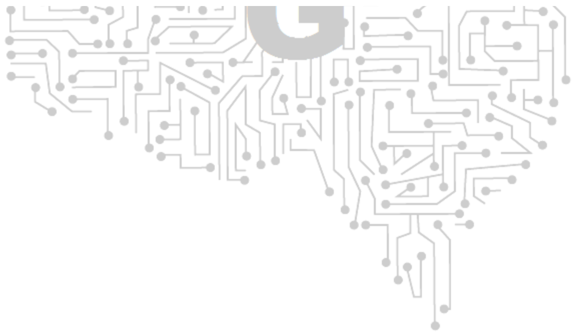
MULTIPLICATION_32BIT

```verilog
`timescale 1ns / 1ps
/************************************************************************
* Author:   Bryan Linares
*           Brian Ortiz
*           Grace Daliwan
* Filename: MPY_32.v
* Date:     Sep. 11, 2018
* Project:  CECS 440 Lab 1
* Version:  1.0
*
* Notes:    32 bit Multiplication module for MIPS ISA.
*           Casts raw input to Integer type for built-in calculation.
*
************************************************************************/
module MPY_32( s, t, product );

   input [31:0] s;
   input [31:0] t;

   output reg [63:0] product;

   integer int_s, int_t;

   always@(*) begin
      int_s = s;
      int_t = t;
      product  = int_s * int_t;
   end
endmodule
```



DIVISON_32BIT

```
`timescale 1ns / 1ps
/****************************************************************
* Author:   Bryan Linares
* Filename: MPY_32.v
* Date:     Sep. 11, 2018
* Project:  CECS 440 Lab 1
* Version:  1.0
*
* Notes:    32 bit Division module for MIPS ISA
*           Casts raw input to Integer type for calculation.
****************************************************************/
module DIV_32( s, t, remainder, quotient );

   input [31:0] s;
   input [31:0] t;

   output reg [31:0] remainder;
   output reg [31:0] quotient;

   integer int_s, int_t;

   always@(*) begin
      int_s = s;
      int_t = t;
      quotient  = int_s / int_t;
      remainder = int_s % int_t;
   end
endmodule
```



BARREL_SHIFTER

```
`timescale 1ns / 1ps
/*****************************************************************************
* Author(s): Brian Ortiz
*            Bryan Linares
*            Grace Daliwan
* Filename: Barrel_Shifter .v
* Project:  CECS 440 Senior Project GBRAINS
*
*****************************************************************************/
module Barrel_Shifter(FS, SHAMT, T, SHFT_OUT, C);
   input [4:0] FS, SHAMT;   // Function type and amount to be shifted
   input [31:0] T;                //data input
   output reg C;                  // Carry flag
   output reg [31:0] SHFT_OUT;    // data outpuT
   always@(*)
      case(FS)
         5'h0C: // SLL
            case(SHAMT)
               5'd 0: {C,SHFT_OUT} = {1'b0, T};
               5'd 1: {C,SHFT_OUT} = {T[31], T[30:0],  1'b0};
               5'd 2: {C,SHFT_OUT} = {T[30], T[29:0],  2'b0};
               5'd 3: {C,SHFT_OUT} = {T[29], T[28:0],  3'b0};
               5'd 4: {C,SHFT_OUT} = {T[28], T[27:0],  4'b0};
               5'd 5: {C,SHFT_OUT} = {T[27], T[26:0],  5'b0};
               5'd 6: {C,SHFT_OUT} = {T[26], T[25:0],  6'b0};
               5'd 7: {C,SHFT_OUT} = {T[25], T[24:0],  7'b0};
               5'd 8: {C,SHFT_OUT} = {T[24], T[23:0],  8'b0};
               5'd 9: {C,SHFT_OUT} = {T[23], T[22:0],  9'b0};
               5'd10: {C,SHFT_OUT} = {T[22], T[21:0], 10'b0};
               5'd11: {C,SHFT_OUT} = {T[21], T[20:0], 11'b0};
               5'd12: {C,SHFT_OUT} = {T[20], T[19:0], 12'b0};
               5'd13: {C,SHFT_OUT} = {T[19], T[18:0], 13'b0};
               5'd14: {C,SHFT_OUT} = {T[18], T[17:0], 14'b0};
               5'd15: {C,SHFT_OUT} = {T[17], T[16:0], 15'b0};
               5'd16: {C,SHFT_OUT} = {T[16], T[15:0], 16'b0};
               5'd17: {C,SHFT_OUT} = {T[15], T[14:0], 17'b0};
               5'd18: {C,SHFT_OUT} = {T[14], T[13:0], 18'b0};
               5'd19: {C,SHFT_OUT} = {T[13], T[12:0], 19'b0};
               5'd20: {C,SHFT_OUT} = {T[12], T[11:0], 20'b0};
               5'd21: {C,SHFT_OUT} = {T[11], T[10:0], 21'b0};
               5'd22: {C,SHFT_OUT} = {T[10], T[ 9:0], 22'b0};
               5'd23: {C,SHFT_OUT} = {T[ 9], T[ 8:0], 23'b0};
               5'd24: {C,SHFT_OUT} = {T[ 8], T[ 7:0], 24'b0};
               5'd25: {C,SHFT_OUT} = {T[ 7], T[ 6:0], 25'b0};
               5'd26: {C,SHFT_OUT} = {T[ 6], T[ 5:0], 26'b0};
               5'd27: {C,SHFT_OUT} = {T[ 5], T[ 4:0], 27'b0};
               5'd28: {C,SHFT_OUT} = {T[ 4], T[ 3:0], 28'b0};
               5'd29: {C,SHFT_OUT} = {T[ 3], T[ 2:0], 29'b0};
               5'd30: {C,SHFT_OUT} = {T[ 2], T[ 1:0], 30'b0};
               5'd31: {C,SHFT_OUT} = {T[ 1], T[0],    31'b0};

            endcase
         5'h0D: // SRL
            case(SHAMT)
               5'd 0: {C,SHFT_OUT} = {1'b0, T};
               5'd 1: {C,SHFT_OUT} = {T[ 0],  1'b0, T[31: 1]};
               5'd 2: {C,SHFT_OUT} = {T[ 1],  2'b0, T[31: 2]};
               5'd 3: {C,SHFT_OUT} = {T[ 2],  3'b0, T[31: 3]};
               5'd 4: {C,SHFT_OUT} = {T[ 3],  4'b0, T[31: 4]};
               5'd 5: {C,SHFT_OUT} = {T[ 4],  5'b0, T[31: 5]};
               5'd 6: {C,SHFT_OUT} = {T[ 5],  6'b0, T[31: 6]};
               5'd 7: {C,SHFT_OUT} = {T[ 6],  7'b0, T[31: 7]};
               5'd 8: {C,SHFT_OUT} = {T[ 7],  8'b0, T[31: 8]};
               5'd 9: {C,SHFT_OUT} = {T[ 8],  9'b0, T[31: 9]};
               5'd10: {C,SHFT_OUT} = {T[ 9], 10'b0, T[31: 10]};
               5'd11: {C,SHFT_OUT} = {T[10], 11'b0, T[31: 11]};
               5'd12: {C,SHFT_OUT} = {T[11], 12'b0, T[31: 12]};
               5'd13: {C,SHFT_OUT} = {T[12], 13'b0, T[31: 13]};
               5'd14: {C,SHFT_OUT} = {T[13], 14'b0, T[31: 14]};
               5'd15: {C,SHFT_OUT} = {T[14], 15'b0, T[31: 15]};
               5'd16: {C,SHFT_OUT} = {T[15], 16'b0, T[31: 16]};
               5'd17: {C,SHFT_OUT} = {T[16], 17'b0, T[31: 17]};
               5'd18: {C,SHFT_OUT} = {T[17], 18'b0, T[31: 18]};
               5'd19: {C,SHFT_OUT} = {T[18], 19'b0, T[31: 19]};
               5'd20: {C,SHFT_OUT} = {T[19], 20'b0, T[31: 20]};
               5'd21: {C,SHFT_OUT} = {T[20], 21'b0, T[31: 21]};
               5'd22: {C,SHFT_OUT} = {T[21], 22'b0, T[31: 22]};
               5'd23: {C,SHFT_OUT} = {T[22], 23'b0, T[31: 23]};
               5'd24: {C,SHFT_OUT} = {T[23], 24'b0, T[31: 24]};
               5'd25: {C,SHFT_OUT} = {T[24], 25'b0, T[31: 25]};
               5'd26: {C,SHFT_OUT} = {T[25], 26'b0, T[31: 26]};
               5'd27: {C,SHFT_OUT} = {T[26], 27'b0, T[31: 27]};
               5'd28: {C,SHFT_OUT} = {T[27], 28'b0, T[31: 28]};
               5'd29: {C,SHFT_OUT} = {T[28], 29'b0, T[31: 29]};
               5'd30: {C,SHFT_OUT} = {T[29], 30'b0, T[31: 30]};
               5'd31: {C,SHFT_OUT} = {T[30], 31'b0, T[31]};
            endcase
          5'h0E: // SRA
            case(SHAMT)
               5'd 0: {SHFT_OUT} = T;
               5'd 1: {SHFT_OUT} = {T[31],   T[31: 1]};
               5'd 2: {SHFT_OUT} = {{2{T[31]}}, T[31:2]};
               5'd 3: {SHFT_OUT} = {{3{T[31]}}, T[31:3]};
               5'd 4: {SHFT_OUT} = {{4{T[31]}}, T[31:4]};
               5'd 5: {SHFT_OUT} = {{5{T[31]}}, T[31:5]};
               5'd 6: {SHFT_OUT} = {{6{T[31]}}, T[31:6]};
               5'd 7: {SHFT_OUT} = {{7{T[31]}}, T[31:7]};
               5'd 8: {SHFT_OUT} = {{8{T[31]}}, T[31:8]};
               5'd 9: {SHFT_OUT} = {{9{T[31]}}, T[31:9]};
               5'd10: {SHFT_OUT} = {{10{T[31]}}, T[31:10]};
               5'd11: {SHFT_OUT} = {{11{T[31]}}, T[31:11]};
               5'd12: {SHFT_OUT} = {{12{T[31]}}, T[31:12]};
               5'd13: {SHFT_OUT} = {{13{T[31]}}, T[31:13]};
               5'd14: {SHFT_OUT} = {{14{T[31]}}, T[31:14]};
               5'd15: {SHFT_OUT} = {{15{T[31]}}, T[31:15]};
               5'd16: {SHFT_OUT} = {{16{T[31]}}, T[31:16]};
               5'd17: {SHFT_OUT} = {{17{T[31]}}, T[31:17]};
               5'd18: {SHFT_OUT} = {{18{T[31]}}, T[31:18]};
               5'd19: {SHFT_OUT} = {{19{T[31]}}, T[31:19]};
               5'd20: {SHFT_OUT} = {{20{T[31]}}, T[31:20]};
               5'd21: {SHFT_OUT} = {{21{T[31]}}, T[31:21]};
               5'd22: {SHFT_OUT} = {{22{T[31]}}, T[31:22]};
               5'd23: {SHFT_OUT} = {{23{T[31]}}, T[31:23]};
               5'd24: {SHFT_OUT} = {{24{T[31]}}, T[31:24]};
               5'd25: {SHFT_OUT} = {{25{T[31]}}, T[31:25]};
               5'd26: {SHFT_OUT} = {{26{T[31]}}, T[31:26]};
               5'd27: {SHFT_OUT} = {{27{T[31]}}, T[31:27]};
               5'd28: {SHFT_OUT} = {{28{T[31]}}, T[31:28]};
               5'd29: {SHFT_OUT} = {{29{T[31]}}, T[31:29]};
               5'd30: {SHFT_OUT} = {{30{T[31]}}, T[31:30]};
               5'd31: {SHFT_OUT} = {{32{T[31]}}};
            endcase
      endcase
endmodule
```

REGISTER_32

```
`timescale 1ns / 1ps
/****************************************************************
* Author:   Bryan Linares
*           Brian Ortiz
*           Grace Daliwan
* Filename: REG32.v
* Date:     Nov. 27, 2018
* Project:  CECS 440 Senior Project
* Version:  1.0
*
* Notes:    32 bit load Register to support the mips processor
*
****************************************************************/
module REG32( CLK, RESET, ld, D, Q );

   input       CLK, RESET;
   input       ld;
   input [31:0] D;

  output reg [31:0] Q;

   always@(posedge CLK, posedge RESET)
      if(RESET) Q <= 32'b0;
      else      Q <= ld? D : Q;

endmodule
```

C. Instruction Memory Modules with Annotated Log Files

The data below details the instructions run through the project simulator to verify that the operations work as designed from modules 1-14 and an enhanced instruction memory and data. Instructions and their relevant register assignments are boxed together for easy parsing.

Module 1

```
@0
3c 01 12 34  // main:       lui  $01, 0x1234
34 21 56 78  //             ori  $01, 0x5678
3c 02 87 65  //             lui  $02, 0x8765
34 42 43 21  //             ori  $02, 0x4321
00 01 18 20  //             add  $03, $00, $01

10 22 00 01  //             beq  $01, $02,
no_eq
10 23 00 03  //             beq  $01, $03,
yes_eq
3c 0e ff ff  // no_eq:      lui  $14, 0xFFFF
35 ce ff ff  //             ori  $14, 0xFFFF
00 00 00 0d  //             breaK

00 00 70 20  // yes_eq:     add  $14, $0, $0

14 23 00 01  //             bne  $01, $03,
no_ne
14 22 00 03  //             bne  $01, $02,
yes_ne
3c 0f ff ff  // no_ne:      lui  $15, 0xFFFF
35 ef ff ff  //             ori  $15, 0xFFFF
00 00 00 0d  //             break

00 00 78 20  // yes_ne:     add  $15, $0, $0
3c 0d 10 01  //             lui  $13, 0x1001
35 ad 00 c0  //             ori  $13, 0x00C0
ad a1 00 00  //             sw   $01, 0($13)
00 00 00 0d  //             break
```

```
time:  641.0 ns       $r[0]:  00000000
time:  651.0 ns       $r[1]:  12345678
time:  661.0 ns       $r[2]:  87654321
time:  671.0 ns       $r[3]:  12345678

time:  681.0 ns       $r[4]:  xxxxxxxx
time:  691.0 ns       $r[5]:  xxxxxxxx
time:  701.0 ns       $r[6]:  xxxxxxxx
time:  711.0 ns       $r[7]:  xxxxxxxx
time:  721.0 ns       $r[8]:  xxxxxxxx
time:  731.0 ns       $r[9]:  xxxxxxxx
time:  741.0 ns       $r[10]: xxxxxxxx
time:  751.0 ns       $r[11]: xxxxxxxx
time:  761.0 ns       $r[12]: xxxxxxxx

time:  771.0 ns       $r[13]: 100100c0
time:  781.0 ns       $r[14]: 00000000
time:  791.0 ns       $r[15]: 00000000


time=  792.0 ns     DM[0c0]=12345678
```

This module tests the branch if equal and branch if not equal operations. If the branch function work correctly, register 13 will be written to 100100c0 and data memory at 0c0 will store the value of register 1 which was loaded with 12345678. If any branch is not working as intended, register 15 will be written with a fail flag of FFFFFFFF and the program will prematurely exit.

Module 2

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

00 01 08 42  // top:  srl  $01, $01, 1
ad e1 00 00  //       sw   $01, 0($15)
21 ef 00 04  //       addi $15, $15, 4
20 42 ff ff  //       addi $02, $02, -1
14 40 ff fb  //       bne  $02, $00, top
08 10 00 0c  //       j    exit
00 00 00 0d  //       break

3c 0e 5a 5a  // exit: lui  $14, 0x5A5A
35 ce 3c 3c  //       ori  $14, 0x3C3C
00 00 00 0d  //       break
```

```
time: 3591.0 ns      $r[0]:  00000000
time: 3601.0 ns      $r[1]:  0000ffff
time: 3611.0 ns      $r[2]:  00000000

time: 3621.0 ns      $r[3]:  xxxxxxxx
time: 3631.0 ns      $r[4]:  xxxxxxxx
time: 3641.0 ns      $r[5]:  xxxxxxxx
time: 3651.0 ns      $r[6]:  xxxxxxxx
time: 3661.0 ns      $r[7]:  xxxxxxxx
time: 3671.0 ns      $r[8]:  xxxxxxxx
time: 3681.0 ns      $r[9]:  xxxxxxxx
time: 3691.0 ns      $r[10]: xxxxxxxx
time: 3701.0 ns      $r[11]: xxxxxxxx
time: 3711.0 ns      $r[12]: xxxxxxxx
time: 3721.0 ns      $r[13]: xxxxxxxx

time: 3731.0 ns      $r[14]: 5a5a3c3c
time: 3741.0 ns      $r[15]: 10010100

time=3742.0 ns     DM[0c0]=7fffffff
time=3742.0 ns     DM[0c4]=3fffffff
time=3742.0 ns     DM[0c8]=1fffffff
time=3742.0 ns     DM[0cc]=0fffffff
time=3742.0 ns     DM[0d0]=07ffffff
time=3742.0 ns     DM[0d4]=03ffffff
time=3742.0 ns     DM[0d8]=01ffffff
time=3742.0 ns     DM[0dc]=00ffffff
time=3742.0 ns     DM[0e0]=007fffff
time=3742.0 ns     DM[0e4]=003fffff
time=3742.0 ns     DM[0e8]=001fffff
time=3742.0 ns     DM[0ec]=000fffff
time=3742.0 ns     DM[0f0]=0007ffff
time=3742.0 ns     DM[0f4]=0003ffff
time=3742.0 ns     DM[0f8]=0001ffff
time=3742.0 ns     DM[0fc]=0000ffff
time=3742.0 ns     DM[3fc]=xxxxxxxx
```

This module tests the shift right logical operation. If the function works correctly, data memory will be loaded with zeros shifting in from the left. If the function is not performing correctly, this will either cause an illegal operation call and a premature program exit or an infinite loop where the program cannot exit. To verify the program's proper functions, data memory should be written from 0c0 to 0fc with a zero bit shifted in from the left at each sequential memory space ending with 0000ffff.

Module 3

```
3c 01 80 00  // main:      lui  $01, 0x8000
34 21 ff ff  //            ori  $01, 0xFFFF
20 02 00 10  //            addi $02, $00, 0x10
3c 0f 10 01  //            lui  $15, 0x1001
35 ef 00 c0  //            ori  $15, 0x00C0

00 01 08 43  // top:       sra  $01, $01, 1
ad e1 00 00  //            sw   $01, 0($15)
21 ef 00 04  //            addi $15, $15, 4
20 42 ff ff  //            addi $02, $02, -1
14 40 ff fb  //            bne  $02, $00, top

08 10 00 0c  //            j    exit
00 00 00 0d  //            break

3c 0e 5a 5a  // exit:      lui  $14, 0x5A5A
35 ce 3c 3c  //            ori  $14, 0x3C3C
00 00 00 0d  //            break
```

```
time: 3591.0 ns      $r[0]:  00000000
time: 3601.0 ns      $r[1]:  ffff8000
time: 3611.0 ns      $r[2]:  00000000
time: 3621.0 ns      $r[3]:  xxxxxxxx
time: 3631.0 ns      $r[4]:  xxxxxxxx
time: 3641.0 ns      $r[5]:  xxxxxxxx
time: 3651.0 ns      $r[6]:  xxxxxxxx
time: 3661.0 ns      $r[7]:  xxxxxxxx
time: 3671.0 ns      $r[8]:  xxxxxxxx
time: 3681.0 ns      $r[9]:  xxxxxxxx
time: 3691.0 ns      $r[10]: xxxxxxxx
time: 3701.0 ns      $r[11]: xxxxxxxx
time: 3711.0 ns      $r[12]: xxxxxxxx
time: 3721.0 ns      $r[13]: xxxxxxxx
time: 3731.0 ns      $r[14]: 5a5a3c3c
time: 3741.0 ns      $r[15]: 10010100
```

```
time=3742.0 ns      DM[0c0]=c0007fff
time=3742.0 ns      DM[0c4]=e0003fff
time=3742.0 ns      DM[0c8]=f0001fff
time=3742.0 ns      DM[0cc]=f8000fff
time=3742.0 ns      DM[0d0]=fc0007ff
time=3742.0 ns      DM[0d4]=fe0003ff
time=3742.0 ns      DM[0d8]=ff0001ff
time=3742.0 ns      DM[0dc]=ff8000ff
time=3742.0 ns      DM[0e0]=ffc0007f
time=3742.0 ns      DM[0e4]=ffe0003f
time=3742.0 ns      DM[0e8]=fff0001f
time=3742.0 ns      DM[0ec]=fff8000f
time=3742.0 ns      DM[0f0]=fffc0007
time=3742.0 ns      DM[0f4]=fffe0003
time=3742.0 ns      DM[0f8]=ffff0001
time=3742.0 ns      DM[0fc]=ffff8000
time=3742.0 ns      DM[3fc]=xxxxxxxx
```

This module tests the shift right arithmetic operation. This operation is essentially a divide by 2, causing memory to receive number values decreasing by half. If this operation is not performing as intended, the program will may have an illegal operation call and exit prematurely or have an infinite loop where the program cannot exit. Data memory should be written to with sign-extended values ending with ffff8000.

Module 4

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

00 01 08 40  // top:       sll  $01, $01, 1
ad e1 00 00  //            sw   $01, 0($15)
21 ef 00 04  //            addi $15, $15, 4
20 42 ff ff  //            addi $02, $02, -1
00 02 18 2a  //            slt  $03, $00, $02
14 60 ff fa  //            bne  $03, $00, top

08 10 00 0d  //            j    exit
00 00 00 0d  //            break

3c 0e 5a 5a  // exit:      lui  $14, 0x5A5A
35 ce 3c 3c  //            ori  $14, 0x3C3C
00 00 00 0d  //            break
```

```
time: 4231.0 ns      $r[0]:  00000000
time: 4241.0 ns      $r[1]:  ffff0000
time: 4251.0 ns      $r[2]:  00000000
time: 4261.0 ns      $r[3]:  00000000
time: 4271.0 ns      $r[4]:  xxxxxxxx
time: 4281.0 ns      $r[5]:  xxxxxxxx
time: 4291.0 ns      $r[6]:  xxxxxxxx
time: 4301.0 ns      $r[7]:  xxxxxxxx
time: 4311.0 ns      $r[8]:  xxxxxxxx
time: 4321.0 ns      $r[9]:  xxxxxxxx
time: 4331.0 ns      $r[10]: xxxxxxxx
time: 4341.0 ns      $r[11]: xxxxxxxx
time: 4351.0 ns      $r[12]: xxxxxxxx
time: 4361.0 ns      $r[13]: xxxxxxxx
time: 4371.0 ns      $r[14]: 5a5a3c3c
time: 4381.0 ns      $r[15]: 10010100

time=4382.0 ns       DM[0c0]=fffffffe
time=4382.0 ns       DM[0c4]=fffffffc
time=4382.0 ns       DM[0c8]=fffffff8
time=4382.0 ns       DM[0cc]=fffffff0
time=4382.0 ns       DM[0d0]=ffffffe0
time=4382.0 ns       DM[0d4]=ffffffc0
time=4382.0 ns       DM[0d8]=ffffff80
time=4382.0 ns       DM[0dc]=ffffff00
time=4382.0 ns       DM[0e0]=fffffe00
time=4382.0 ns       DM[0e4]=fffffc00
time=4382.0 ns       DM[0e8]=fffff800
time=4382.0 ns       DM[0ec]=fffff000
time=4382.0 ns       DM[0f0]=ffffe000
time=4382.0 ns       DM[0f4]=ffffc000
time=4382.0 ns       DM[0f8]=ffff8000
time=4382.0 ns       DM[0fc]=ffff0000
time=4382.0 ns       DM[3fc]=xxxxxxxx
```

This module tests the shift left logical operation and the set if less than operation. If the operation works correctly, data memory will be stored with values having zeros coming in from the right. This is similar to the output of module 2 but in the opposite direction of shifting in the zero bit. The last memory space written to should be 0fc and the last data value written in should be ffff0000.

Module 5

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

00 01 08 40   // top:       sll  $01, $01, 1
ad e1 00 00   //            sw   $01, 0($15)
21 ef 00 04   //            addi $15, $15, 4
20 42 00 01   //            addi $02, $02, 1
28 43 00 00   //            slti $03, $02, 0
14 60 ff fa   //            bne  $03, $00, top

08 10 00 0d   //            j    exit
00 00 00 0d   //            break

3c 0e 5a 5a   // exit:      lui  $14, 0x5A5A
35 ce 3c 3c   //            ori  $14, 0x3C3C
00 00 00 0d   //            break
```

```
time: 4231.0 ns      $r[0]:  00000000
time: 4241.0 ns      $r[1]:  ffff0000
time: 4251.0 ns      $r[2]:  00000000
time: 4261.0 ns      $r[3]:  00000000
time: 4271.0 ns      $r[4]:  xxxxxxxx
time: 4281.0 ns      $r[5]:  xxxxxxxx
time: 4291.0 ns      $r[6]:  xxxxxxxx
time: 4301.0 ns      $r[7]:  xxxxxxxx
time: 4311.0 ns      $r[8]:  xxxxxxxx
time: 4321.0 ns      $r[9]:  xxxxxxxx
time: 4331.0 ns      $r[10]: xxxxxxxx
time: 4341.0 ns      $r[11]: xxxxxxxx
time: 4351.0 ns      $r[12]: xxxxxxxx
time: 4361.0 ns      $r[13]: xxxxxxxx
time: 4371.0 ns      $r[14]: 5a5a3c3c
time: 4381.0 ns      $r[15]: 10010100

time=4382.0 ns      DM[0c0]=fffffffe
time=4382.0 ns      DM[0c4]=fffffffc
time=4382.0 ns      DM[0c8]=fffffff8
time=4382.0 ns      DM[0cc]=fffffff0
time=4382.0 ns      DM[0d0]=ffffffe0
time=4382.0 ns      DM[0d4]=ffffffc0
time=4382.0 ns      DM[0d8]=ffffff80
time=4382.0 ns      DM[0dc]=ffffff00
time=4382.0 ns      DM[0e0]=fffffe00
time=4382.0 ns      DM[0e4]=fffffc00
time=4382.0 ns      DM[0e8]=fffff800
time=4382.0 ns      DM[0ec]=fffff000
time=4382.0 ns      DM[0f0]=ffffe000
time=4382.0 ns      DM[0f4]=ffffc000
time=4382.0 ns      DM[0f8]=ffff8000
time=4382.0 ns      DM[0fc]=ffff0000
time=4382.0 ns      DM[3fc]=xxxxxxxx
```

This module tests the set if less than immediate operation. This will affect the branching of the program as the program will infinitely loop if this function is not operational. This output should be similar to module 4's output as it is essentially doing the same thing, but testing with an immediate value rather than a register value.

Module 6

```
@0
3c 0f 10 01  // lui  $15, 0x1001
35 ef 00 00  // ori  $15, 0x0000
3c 0e 10 01  // lui  $14, 0x1001
35 ce 00 c0  // ori  $14, 0x00C0
20 0d 00 10  // addi $13, $00, 16
8d e1 00 04  // lw   $01, 04($15)
8d e2 00 08  // lw   $02, 08($15)
8d e3 00 0c  // lw   $03, 12($15)
8d e4 00 10  // lw   $04, 16($15)
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)
ad d1 00 00  // sw   $17, 00($14)
21 ef 00 04  // addi $15, $15, 04
21 ce 00 04  // addi $14, $14, 04
21 ad ff ff  // addi $13, $13, -1
15 a0 ff fa  // bne  $13, $00, mem2mem
00 00 00 0d  // break
```

```
time: 4881.0 ns $r[0]:  00000000
time: 4891.0 ns $r[1]:  12345678
time: 4901.0 ns $r[2]:  89abcdef
time: 4911.0 ns $r[3]:  a5a5a5a5
time: 4921.0 ns $r[4]:  5a5a5a5a
time: 4931.0 ns $r[5]:  2468ace0
time: 4941.0 ns $r[6]:  13579bdf
time: 4951.0 ns $r[7]:  0f0f0f0f
time: 4961.0 ns $r[8]:  f0f0f0f0
time: 4971.0 ns $r[9]:  00000009
time: 4981.0 ns $r[10]: 0000000a
time: 4991.0 ns $r[11]: 0000000b
time: 5001.0 ns $r[12]: 0000000c
time: 5011.0 ns $r[13]: 00000000
time: 5021.0 ns $r[14]: 10010100
time: 5031.0 ns $r[15]: 10010040

time: 5041.0 ns $r[16]: xxxxxxxx
time: 5051.0 ns $r[17]: 000075cc
time: 5061.0 ns $r[18]: xxxxxxxx
time: 5071.0 ns $r[19]: xxxxxxxx
time: 5081.0 ns $r[20]: xxxxxxxx
time: 5091.0 ns $r[21]: xxxxxxxx
time: 5101.0 ns $r[22]: xxxxxxxx
time: 5111.0 ns $r[23]: xxxxxxxx
time: 5121.0 ns $r[24]: xxxxxxxx
time: 5131.0 ns $r[25]: xxxxxxxx
time: 5141.0 ns $r[26]: xxxxxxxx
time: 5151.0 ns $r[27]: xxxxxxxx
time: 5161.0 ns $r[28]: xxxxxxxx
time: 5171.0 ns $r[29]: 000003fc
time: 5181.0 ns $r[30]: xxxxxxxx
time: 5191.0 ns $r[31]: xxxxxxxx
```

```
time=5191.0 ns DM[0c0]=c3c3c3c3
time=5191.0 ns DM[0c4]=12345678
time=5191.0 ns DM[0c8]=89abcdef
time=5191.0 ns DM[0cc]=a5a5a5a5
time=5191.0 ns DM[0d0]=5a5a5a5a
time=5191.0 ns DM[0d4]=2468ace0
time=5191.0 ns DM[0d8]=13579bdf
time=5191.0 ns DM[0dc]=0f0f0f0f
time=5191.0 ns DM[0e0]=f0f0f0f0
time=5191.0 ns DM[0e4]=00000009
time=5191.0 ns DM[0e8]=0000000a
time=5191.0 ns DM[0ec]=0000000b
time=5191.0 ns DM[0f0]=0000000c
time=5191.0 ns DM[0f4]=0000000d
time=5191.0 ns DM[0f8]=fffffff8
time=5191.0 ns DM[0fc]=000075cc
time=5191.0 ns DM[3fc]=xxxxxxxx
```

```
This module tests load word and store word, having corresponding values brought from memory and
written to memory. Registers 1 through 12 should be loaded with values from data memory 004 through
030 respectively. Register 17 is used as a temporary to store the values from 000 through 03c to
memory at 0c0 through 0fc.
```

Module 7

```
@0
3c 0f 10 01  // main:       lui  $15, 0x1001
35 ef 00 00  //             ori  $15, 0x0000
3c 0e 10 01  //             lui  $14, 0x1001
35 ce 00 c0  //             ori  $14, 0x00C0
20 0d 00 10  //             addi $13, $00, 16
8d e1 00 04  //             lw   $01, 04($15)
8d e2 00 08  //             lw   $02, 08($15)
8d e3 00 0c  //             lw   $03, 12($15)
8d e4 00 10  //             lw   $04, 16($15)
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
00 00 00 0d  //             break

8d f1 00 00  // mem2mem:    lw   $17, 00($15)
ad d1 00 00  //             sw   $17, 00($14)
21 ef 00 04  //             addi $15, $15, 04
21 ce 00 04  //             addi $14, $14, 04
21 ad ff ff  //             addi $13, $13, -1
15 a0 ff fa  //             bne  $13, $00,
mem2mem
03 e0 00 08  //             jr   $31
00 00 00 0d  //             break
```

```
time: 5041.0 ns $r[0]:  00000000
time: 5051.0 ns $r[1]:  12345678
time: 5061.0 ns $r[2]:  89abcdef
time: 5071.0 ns $r[3]:  a5a5a5a5
time: 5081.0 ns $r[4]:  5a5a5a5a
time: 5091.0 ns $r[5]:  2468ace0
time: 5101.0 ns $r[6]:  13579bdf
time: 5111.0 ns $r[7]:  0f0f0f0f
time: 5121.0 ns $r[8]:  f0f0f0f0
time: 5131.0 ns $r[9]:  00000009
time: 5141.0 ns $r[10]: 0000000a
time: 5151.0 ns $r[11]: 0000000b
time: 5161.0 ns $r[12]: 0000000c
time: 5171.0 ns $r[13]: 00000000
time: 5181.0 ns $r[14]: 10010100
time: 5191.0 ns $r[15]: ffffffff
time: 5201.0 ns $r[16]: xxxxxxxx
time: 5211.0 ns $r[17]: 000075cc
time: 5221.0 ns $r[18]: xxxxxxxx
time: 5231.0 ns $r[19]: xxxxxxxx
time: 5241.0 ns $r[20]: xxxxxxxx
time: 5251.0 ns $r[21]: xxxxxxxx
time: 5261.0 ns $r[22]: xxxxxxxx
time: 5271.0 ns $r[23]: xxxxxxxx
time: 5281.0 ns $r[24]: xxxxxxxx
time: 5291.0 ns $r[25]: xxxxxxxx
time: 5301.0 ns $r[26]: xxxxxxxx
time: 5311.0 ns $r[27]: xxxxxxxx
time: 5321.0 ns $r[28]: xxxxxxxx
time: 5331.0 ns $r[29]: 000003fc
time: 5341.0 ns $r[30]: xxxxxxxx
time: 5351.0 ns $r[31]: 00000048

time=5351.0 ns DM[0c0]=c3c3c3c3
time=5351.0 ns DM[0c4]=12345678
time=5351.0 ns DM[0c8]=89abcdef
time=5351.0 ns DM[0cc]=a5a5a5a5
time=5351.0 ns DM[0d0]=5a5a5a5a
time=5351.0 ns DM[0d4]=2468ace0
time=5351.0 ns DM[0d8]=13579bdf
time=5351.0 ns DM[0dc]=0f0f0f0f
time=5351.0 ns DM[0e0]=f0f0f0f0
time=5351.0 ns DM[0e4]=00000009
time=5351.0 ns DM[0e8]=0000000a
time=5351.0 ns DM[0ec]=0000000b
time=5351.0 ns DM[0f0]=0000000c
time=5351.0 ns DM[0f4]=0000000d
time=5351.0 ns DM[0f8]=fffffff8
time=5351.0 ns DM[0fc]=000075cc
time=5351.0 ns DM[3fc]=xxxxxxxx
```

This module tests jump and link. If jump and link does not work correctly, register 31 will not be written to and several values in memory will not be written to correctly. The output should be similar to that of module 6 but the memory storage routine should be jumped to by jump and link and the program should return to the instruction count it jumped from.

Module 8

```
@0
3c 0f 10 01  // main:      lui  $15, 0x1001
35 ef 00 00  //            ori  $15, 0x0000
8d e1 00 00  //            lw   $01, 00($15)
8d e2 00 04  //            lw   $02, 04($15)
8d e3 00 08  //            lw   $03, 08($15)
8d e4 00 0c  //            lw   $04, 12($15)
8d e5 00 10  //            lw   $05, 16($15)
8d e6 00 14  //            lw   $06, 20($15)
8d e7 00 18  //            lw   $07, 24($15)
00 22 00 18  //            mult $01, $02
00 00 40 12  //            mflo $08
14 a8 00 10  //            bne  $05, $08,
fail1
00 62 00 18  //            mult $03, $02
00 00 48 12  //            mflo $09
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
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
14 ad 00 12  //            bne  $05, $13,
fail4

3c 0e 00 00  // pass:      lui  $14, 0x0000
35 ce 00 00  //            ori  $14, 0x0000
00 00 00 0d  //            break
3c 0e ff ff  // fail1:     lui  $14, 0xFFFF
35 ce ff ff  //            ori  $14, 0xFFFF
00 00 00 0d  //            break
3c 0e ff ff  // fail2L:    lui  $14, 0xFFFF
35 ce ff fe  //            ori  $14, 0xFFFE
00 00 00 0d  //            break
3c 0e ff ff  // fail2H:    lui  $14, 0xFFFF
35 ce ff fd  //            ori  $14, 0xFFFD
00 00 00 0d  //            break
3c 0e ff ff  // fail3L:    lui  $14, 0xFFFF
35 ce ff fc  //            ori  $14, 0xFFFC
00 00 00 0d  //            break
3c 0e ff ff  // fail3H:    lui  $14, 0xFFFF
35 ce ff fb  //            ori  $14, 0xFFFB
00 00 00 0d  //            break
3c 0e ff ff  // fail4:     lui  $14, 0xFFFF
35 ce ff fa  //            ori  $14, 0xFFFA
00 00 00 0d  //            break
```

```
time: 1131.0 ns $r[0]:  00000000
time: 1141.0 ns $r[1]:  00000019
time: 1151.0 ns $r[2]:  000003e8
time: 1161.0 ns $r[3]:  ffffffe7
time: 1171.0 ns $r[4]:  fffffc18
time: 1181.0 ns $r[5]:  000061a8
time: 1191.0 ns $r[6]:  ffff9e58
time: 1201.0 ns $r[7]:  ffffffff
time: 1211.0 ns $r[8]:  000061a8
time: 1221.0 ns $r[9]:  ffff9e58
time: 1231.0 ns $r[10]: ffffffff
time: 1241.0 ns $r[11]: ffff9e58
time: 1251.0 ns $r[12]: ffffffff
time: 1261.0 ns $r[13]: 000061a8
time: 1271.0 ns $r[14]: 00000000
time: 1281.0 ns $r[15]: 10010000

time: 1291.0 ns $r[16]: xxxxxxxx
time: 1301.0 ns $r[17]: xxxxxxxx
time: 1311.0 ns $r[18]: xxxxxxxx
time: 1321.0 ns $r[19]: xxxxxxxx
time: 1331.0 ns $r[20]: xxxxxxxx
time: 1341.0 ns $r[21]: xxxxxxxx
time: 1351.0 ns $r[22]: xxxxxxxx
time: 1361.0 ns $r[23]: xxxxxxxx
time: 1371.0 ns $r[24]: xxxxxxxx
time: 1381.0 ns $r[25]: xxxxxxxx
time: 1391.0 ns $r[26]: xxxxxxxx
time: 1401.0 ns $r[27]: xxxxxxxx
time: 1411.0 ns $r[28]: xxxxxxxx
time: 1421.0 ns $r[29]: 000003fc
time: 1431.0 ns $r[30]: xxxxxxxx
time: 1441.0 ns $r[31]: xxxxxxxx

time=1441.0 ns DM[0c0]=xxxxxxxx
time=1441.0 ns DM[0c4]=xxxxxxxx
time=1441.0 ns DM[0c8]=xxxxxxxx
time=1441.0 ns DM[0cc]=xxxxxxxx
time=1441.0 ns DM[0d0]=xxxxxxxx
time=1441.0 ns DM[0d4]=xxxxxxxx
time=1441.0 ns DM[0d8]=xxxxxxxx
time=1441.0 ns DM[0dc]=xxxxxxxx
time=1441.0 ns DM[0e0]=xxxxxxxx
time=1441.0 ns DM[0e4]=xxxxxxxx
time=1441.0 ns DM[0e8]=xxxxxxxx
time=1441.0 ns DM[0ec]=xxxxxxxx
time=1441.0 ns DM[0f0]=xxxxxxxx
time=1441.0 ns DM[0f4]=xxxxxxxx
```

```
time=1441.0 ns DM[0f8]=xxxxxxxx                    time=1441.0 ns DM[3fc]=xxxxxxxx
time=1441.0 ns DM[0fc]=xxxxxxxx
```

This module tests the multiply, move from hi, and move from low operations. If the functions are correct, register 14 will be written to with zeroes, otherwise, fail "flag"s will be written and the program will exit prematurely.

Module 9

```
@0
3c 0f 10 01  // main:      lui  $15, 0x1001
35 ef 00 c0  //            ori  $15, 0x00C0
20 01 ff 8a  //            addi $01, $00, -118
20 02 00 8a  //            addi $02  $00,  138
0c 10 00 22  //            jal  slt_tests

3c 0d 77 88  //            lui  $13, 0x7788
35 ad 77 88  //            ori  $13, 0x7788
3c 0c 88 77  //            lui  $12, 0x8877
35 8c 88 77  //            ori  $12, 0x8877
3c 0b ff ff  //            lui  $11, 0xFFFF
35 6b ff ff  //            ori  $11, 0xFFFF

01 ac 50 26  //            xor  $10, $13, $12
11 4b 00 02  //            beq  $10, $11,
xor_pass
20 0e ff fb  //            addi $14, $00, -5
00 00 00 0d  //            break
01 ac 48 24  // xor_pass:  and  $09, $13, $12
11 20 00 02  //            beq  $09, $00,
and_pass
20 0e ff fa  //            addi $14, $00, -6
00 00 00 0d  //            break
01 e2 48 25  // and_pass:  or   $09, $15, $02
3c 08 10 01  //            lui  $08, 0x1001
35 08 00 ca  //            ori  $08, 0x00CA
11 09 00 02  //            beq  $08, $09,
or_pass
20 0e ff f9  //            addi $14, $00, -7
00 00 00 0d  //            break
01 e2 48 27  // or_pass:   nor  $09, $15, $02
3c 08 ef fe  //            lui  $08, 0xEFFE
35 08 ff 35  //            ori  $08, 0xFF35
11 09 00 02  //            beq  $08, $09,
nor_pass
20 0e ff f8  //            addi $14, $00, -8
00 00 00 0d  //            break
ad e8 00 10  // nor_pass:  sw   $08, 0x10($15)
00 00 70 20  //            add  $14, $00, $00
00 00 00 0d  //            break
             //
```

```
time: 1761.0 ns $r[0]:  00000000
time: 1771.0 ns $r[1]:  ffffff8a
time: 1781.0 ns $r[2]:  0000008a
time: 1791.0 ns $r[3]:  00000000
time: 1801.0 ns $r[4]:  000000c0
time: 1811.0 ns $r[5]:  000000c4
time: 1821.0 ns $r[6]:  000000c8
time: 1831.0 ns $r[7]:  000000cc
time: 1841.0 ns $r[8]:  effeff35
time: 1851.0 ns $r[9]:  effeff35
time: 1861.0 ns $r[10]: ffffffff
time: 1871.0 ns $r[11]: ffffffff
time: 1881.0 ns $r[12]: 88778877
time: 1891.0 ns $r[13]: 77887788
time: 1901.0 ns $r[14]: 00000000
time: 1911.0 ns $r[15]: 100100c0

time: 1921.0 ns $r[16]: xxxxxxxx
time: 1931.0 ns $r[17]: xxxxxxxx
time: 1941.0 ns $r[18]: xxxxxxxx
time: 1951.0 ns $r[19]: xxxxxxxx
time: 1961.0 ns $r[20]: xxxxxxxx
time: 1971.0 ns $r[21]: xxxxxxxx
time: 1981.0 ns $r[22]: xxxxxxxx
time: 1991.0 ns $r[23]: xxxxxxxx
time: 2001.0 ns $r[24]: xxxxxxxx
time: 2011.0 ns $r[25]: xxxxxxxx
time: 2021.0 ns $r[26]: xxxxxxxx
time: 2031.0 ns $r[27]: xxxxxxxx
time: 2041.0 ns $r[28]: xxxxxxxx
time: 2051.0 ns $r[29]: 000003fc
time: 2061.0 ns $r[30]: xxxxxxxx
time: 2071.0 ns $r[31]: 00000014

time=2071.0 ns DM[0c0]=000000c0
time=2071.0 ns DM[0c4]=000000c4
time=2071.0 ns DM[0c8]=000000c8
time=2071.0 ns DM[0cc]=000000cc
time=2071.0 ns DM[0d0]=effeff35
time=2071.0 ns DM[0d4]=xxxxxxxx
time=2071.0 ns DM[0d8]=xxxxxxxx
time=2071.0 ns DM[0dc]=xxxxxxxx
time=2071.0 ns DM[0e0]=xxxxxxxx
time=2071.0 ns DM[0e4]=xxxxxxxx
time=2071.0 ns DM[0e8]=xxxxxxxx
time=2071.0 ns DM[0ec]=xxxxxxxx
```

```
time=2071.0 ns DM[0f0]=xxxxxxxx
time=2071.0 ns DM[0f4]=xxxxxxxx
time=2071.0 ns DM[0f8]=xxxxxxxx
time=2071.0 ns DM[0fc]=xxxxxxxx
time=2071.0 ns DM[3fc]=xxxxxxxx
```

```
00 22 18 2a  // slt_tests: slt  $03, $01, $02
14 60 00 02  //            bne  $03, $00, slt1
20 0e ff ff  //            addi $14, $00, -1
00 00 00 0d  //            break
20 04 00 c0  // slt1:      addi $04, $00, 0xC0
ad e4 00 00  //            sw   $04, 0x00($15)

00 41 18 2b  //            sltu $03, $02, $01
14 60 00 02  //            bne  $03, $00, slt2
20 0e ff fe  //            addi $14, $00, -2
00 00 00 0d  //            break
20 05 00 c4  // slt2:      addi $05, $00, 0xC4
ad e5 00 04  //            sw   $05, 0x04($15)

00 41 18 2a  //            slt  $03, $02, $01
10 60 00 02  //            beq  $03, $00, slt3
20 0e ff fd  //            addi $14, $00, -3
00 00 00 0d  //            break
20 06 00 c8  // slt3:      addi $06, $00, 0xC8
ad e6 00 08  //            sw   $06, 0x08($15)

00 22 18 2b  //            sltu $03, $01, $02
10 60 00 02  //            beq  $03, $00, slt4
20 0e ff fc  //            addi $14, $00, -4
00 00 00 0d  //            break

20 07 00 cc  // slt4:      addi $07, $00, 0xCC
ad e7 00 0c  //            sw   $07, 0x0C($15)
03 e0 00 08  //            jr   $31
```

```
time: 1761.0 ns $r[0]:  00000000
time: 1771.0 ns $r[1]:  ffffff8a
time: 1781.0 ns $r[2]:  0000008a
time: 1791.0 ns $r[3]:  00000000
time: 1801.0 ns $r[4]:  000000c0
time: 1811.0 ns $r[5]:  000000c4
time: 1821.0 ns $r[6]:  000000c8
time: 1831.0 ns $r[7]:  000000cc
time: 1841.0 ns $r[8]:  effeff35
time: 1851.0 ns $r[9]:  effeff35
time: 1861.0 ns $r[10]: ffffffff
time: 1871.0 ns $r[11]: ffffffff
time: 1881.0 ns $r[12]: 88778877
time: 1891.0 ns $r[13]: 77887788
time: 1901.0 ns $r[14]: 00000000
time: 1911.0 ns $r[15]: 100100c0

time: 1921.0 ns $r[16]: xxxxxxxx
time: 1931.0 ns $r[17]: xxxxxxxx
time: 1941.0 ns $r[18]: xxxxxxxx
time: 1951.0 ns $r[19]: xxxxxxxx
time: 1961.0 ns $r[20]: xxxxxxxx
time: 1971.0 ns $r[21]: xxxxxxxx
time: 1981.0 ns $r[22]: xxxxxxxx
time: 1991.0 ns $r[23]: xxxxxxxx
time: 2001.0 ns $r[24]: xxxxxxxx
time: 2011.0 ns $r[25]: xxxxxxxx
time: 2021.0 ns $r[26]: xxxxxxxx
time: 2031.0 ns $r[27]: xxxxxxxx
time: 2041.0 ns $r[28]: xxxxxxxx
time: 2051.0 ns $r[29]: 000003fc
time: 2061.0 ns $r[30]: xxxxxxxx
time: 2071.0 ns $r[31]: 00000014
```

```
time=2071.0 ns DM[0c0]=000000c0
time=2071.0 ns DM[0c4]=000000c4
time=2071.0 ns DM[0c8]=000000c8
time=2071.0 ns DM[0cc]=000000cc
time=2071.0 ns DM[0d0]=effeff35
time=2071.0 ns DM[0d4]=xxxxxxxx
time=2071.0 ns DM[0d8]=xxxxxxxx
time=2071.0 ns DM[0dc]=xxxxxxxx
time=2071.0 ns DM[0e0]=xxxxxxxx
time=2071.0 ns DM[0e4]=xxxxxxxx
time=2071.0 ns DM[0e8]=xxxxxxxx
```

```
time=2071.0 ns DM[0ec]=xxxxxxxx          time=2071.0 ns DM[0f8]=xxxxxxxx
time=2071.0 ns DM[0f0]=xxxxxxxx          time=2071.0 ns DM[0fc]=xxxxxxxx
time=2071.0 ns DM[0f4]=xxxxxxxx          time=2071.0 ns DM[3fc]=xxxxxxxx
```

This module tests xor, and, or, and nor. If these functions do not work correctly the program will prematurely terminate and fail flags will be set. All slt tests affect registers 3-7 and data sections 0c0 through 0d0. All xor, and, or and nor instructions only affect registers 8 through 14.

Module 10

```
@0
3c 0f 10 01  // main:     lui  $15, 0x1001
35 ef 00 00  //           ori  $15, 0x0000
8d e1 00 00  //           lw   $01, 00($15)
8d e2 00 04  //           lw   $02, 04($15)
8d e3 00 08  //           lw   $03, 08($15)
8d e4 00 0c  //           lw   $04, 12($15)
8d e5 00 10  //           lw   $05, 16($15)
8d e6 00 14  //           lw   $06, 20($15)
8d e7 00 18  //           lw   $07, 24($15)
8d e8 00 1c  //           lw   $08, 28($15)

00 22 00 1a  //           div  $01, $02
00 00 48 12  //           mflo $09
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
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
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
00 00 50 10  //           mfhi $10
15 25 00 19  //           bne  $09, $05,
fail4Q
15 48 00 1b  //           bne  $10, $08,
fail4R
```

```
time: 1441.0 ns $r[0]:  00000000
time: 1451.0 ns $r[1]:  00040911
time: 1461.0 ns $r[2]:  000003e8
time: 1471.0 ns $r[3]:  fffbf6ef
time: 1481.0 ns $r[4]:  fffffc18
time: 1491.0 ns $r[5]:  00000108
time: 1501.0 ns $r[6]:  000001d1
time: 1511.0 ns $r[7]:  fffffef8
time: 1521.0 ns $r[8]:  fffffe2f
time: 1531.0 ns $r[9]:  00000108
time: 1541.0 ns $r[10]: fffffe2f
time: 1551.0 ns $r[11]: 00000000
time: 1561.0 ns $r[12]: 00000000
time: 1571.0 ns $r[13]: 00000000
time: 1581.0 ns $r[14]: 00000000
time: 1591.0 ns $r[15]: 10010000
```

```
3c 0b 00 00  // pass:      lui  $11, 0x0000
35 6b 00 00  //            ori  $11, 0x0000
00 0b 60 20  //            add  $12, $00, $11
00 0b 68 20  //            add  $13, $00, $11
00 0b 70 20  //            add  $14, $00, $11
00 00 00 0d  //            break

3c 0e ff ff  // fail1Q:    lui  $14, 0xFFFF
35 ce ff ff  //            ori  $14, 0xFFFF
00 00 00 0d  //            break
3c 0e ff ff  // fail1R:    lui  $14, 0xFFFF
35 ce ff fe  //            ori  $14, 0xFFFE
00 00 00 0d  //            break
3c 0e ff ff  // fail2Q:    lui  $14, 0xFFFF
35 ce ff fd  //            ori  $14, 0xFFFD
00 00 00 0d  //            break
3c 0e ff ff  // fail2R:    lui  $14, 0xFFFF
35 ce ff fc  //            ori  $14, 0xFFFC
00 00 00 0d  //            break
3c 0e ff ff  // fail3Q:    lui  $14, 0xFFFF
35 ce ff fb  //            ori  $14, 0xFFFB
00 00 00 0d  //            break
3c 0e ff ff  // fail3R:    lui  $14, 0xFFFF
35 ce ff fa  //            ori  $14, 0xFFFA
00 00 00 0d  //            break
3c 0e ff ff  // fail4Q:    lui  $14, 0xFFFF
35 ce ff f9  //            ori  $14, 0xFFF9
00 00 00 0d  //            break
3c 0e ff ff  // fail4R:    lui  $14, 0xFFFF
35 ce ff f8  //            ori  $14, 0xFFF8
00 00 00 0d  //            break
```

```
time: 1441.0 ns $r[0]:  00000000
time: 1451.0 ns $r[1]:  00040911
time: 1461.0 ns $r[2]:  000003e8
time: 1471.0 ns $r[3]:  fffbf6ef
time: 1481.0 ns $r[4]:  fffffc18
time: 1491.0 ns $r[5]:  00000108
time: 1501.0 ns $r[6]:  000001d1
time: 1511.0 ns $r[7]:  fffffef8
time: 1521.0 ns $r[8]:  fffffe2f
time: 1531.0 ns $r[9]:  00000108
time: 1541.0 ns $r[10]: fffffe2f
time: 1551.0 ns $r[11]: 00000000
time: 1561.0 ns $r[12]: 00000000
time: 1571.0 ns $r[13]: 00000000
time: 1581.0 ns $r[14]: 00000000
time: 1591.0 ns $r[15]: 10010000
```

This module tests the divide operation. If the function does not work correctly, fail flags will be set and the program will terminate prematurely. If all functions work as intended, registers 11 through 14 should be written with zeros.

Module 11

```
@0
3c 0f 10 01  // main:       lui  $15, 0x1001
35 ef 00 c0  //             ori  $15, 0x00C0
20 01 ff 8a  //             addi $01, $00, -118
20 02 00 8a  //             addi $02  $00,  138
0c 10 00 1a  //             jal  sltiu_tests

3c 0d ff ff  //             lui  $13, 0xFFFF
35 ad 55 55  //             ori  $13, 0x5555
3c 0c ff ff  //             lui  $12, 0xFFFF
35 8c fa f5  //             ori  $12, 0xFAF5
3c 0b ff ff  //             lui  $11, 0xFFFF
35 6b ff ff  //             ori  $11, 0xFFFF
3c 0a 00 00  //             lui  $10, 0x0000
35 4a f0 f0  //             ori  $10, 0xF0F0

39 a9 aa aa  //             xori $09, $13,
0xAAAA
01 2b 40 22  //             sub  $08, $09, $11
11 00 00 02  //             beq  $08, $00,
xor_p1
20 0e ff f9  //             addi $14, $00, -7
00 00 00 0d  //             break
31 87 f5 fa  // xor_p1:     andi $07, $12,
0xF5FA
00 ea 40 22  //             sub  $08, $07, $10
11 00 00 02  //             beq  $08, $00,
xor_p2
20 0e ff f8  //             addi $14, $00, -8
00 00 00 0d  //             break
ad e1 00 18  // xor_p2:     sw   $01, 0x18($15)
00 00 00 0d  //             break
00 00 00 0d  //             break
```

```
time: 1921.0 ns $r[0]:  00000000
time: 1931.0 ns $r[1]:  ffffff8a
time: 1941.0 ns $r[2]:  0000008a
time: 1951.0 ns $r[3]:  00000000
time: 1961.0 ns $r[4]:  000000c0
time: 1971.0 ns $r[5]:  000000c4
time: 1981.0 ns $r[6]:  000000d4
time: 1991.0 ns $r[7]:  0000f0f0
time: 2001.0 ns $r[8]:  00000000
time: 2011.0 ns $r[9]:  ffffffff
time: 2021.0 ns $r[10]: 0000f0f0
time: 2031.0 ns $r[11]: ffffffff
time: 2041.0 ns $r[12]: ffffaf5
time: 2051.0 ns $r[13]: ffff5555
time: 2061.0 ns $r[14]: 00000000
time: 2071.0 ns $r[15]: 100100c0

time: 2081.0 ns $r[16]: xxxxxxxx
time: 2091.0 ns $r[17]: xxxxxxxx
time: 2101.0 ns $r[18]: xxxxxxxx
time: 2111.0 ns $r[19]: xxxxxxxx
time: 2121.0 ns $r[20]: xxxxxxxx
time: 2131.0 ns $r[21]: xxxxxxxx
time: 2141.0 ns $r[22]: xxxxxxxx
time: 2151.0 ns $r[23]: xxxxxxxx
time: 2161.0 ns $r[24]: xxxxxxxx
time: 2171.0 ns $r[25]: xxxxxxxx
time: 2181.0 ns $r[26]: xxxxxxxx
time: 2191.0 ns $r[27]: xxxxxxxx
time: 2201.0 ns $r[28]: xxxxxxxx
time: 2211.0 ns $r[29]: 000003fc
time: 2221.0 ns $r[30]: xxxxxxxx
time: 2231.0 ns $r[31]: 00000014


time=2231.0 ns DM[0c0]=000000c0
```

```
time=2231.0 ns DM[0c4]=000000c4          time=2231.0 ns DM[0e4]=xxxxxxxx
time=2231.0 ns DM[0c8]=000000c8          time=2231.0 ns DM[0e8]=xxxxxxxx
time=2231.0 ns DM[0cc]=000000cc          time=2231.0 ns DM[0ec]=xxxxxxxx
time=2231.0 ns DM[0d0]=000000d0          time=2231.0 ns DM[0f0]=xxxxxxxx
time=2231.0 ns DM[0d4]=000000d4          time=2231.0 ns DM[0f4]=xxxxxxxx
time=2231.0 ns DM[0d8]=ffffff8a          time=2231.0 ns DM[0f8]=xxxxxxxx
time=2231.0 ns DM[0dc]=xxxxxxxx          time=2231.0 ns DM[0fc]=xxxxxxxx
time=2231.0 ns DM[0e0]=xxxxxxxx          time=2231.0 ns DM[3fc]=xxxxxxxx
```

```
This module tests the set if less than immediate unsigned operation. The program
should jump to the sltiu test section and write to memory at 0c0 through 0d4 if it
functions as intended. ffffff9a should be stored to data memory at 0d8 if all
operations functioned as intended.
```

```
            // sltiu_tests:
2c 23 ff 8b  //              sltiu  $03, $01, -
117
14 60 00 02  //              bne    $03, $00,
slt1_p1
20 0e ff ff  //              addi   $14, $00, -1
00 00 00 0d  //              break
20 04 00 c0  // slt1_p1:     addi   $04, $00,
0xC0
ad e4 00 00  //              sw     $04,
0x00($15)
2c 23 ff 89  //              sltiu  $03, $01, -
119
10 60 00 02  //              beq    $03, $00,
slt_p2
20 0e ff fe  //              addi   $14, $00, -2
00 00 00 0d  //              break
20 05 00 c4  // slt_p2:      addi   $05, $00,
0xC4
ad e5 00 04  //              sw     $05,
0x04($15)

2c 23 ff 8a  //              sltiu  $03, $01, -
118
10 60 00 02  //              beq    $03, $00,
slt_p3
20 0e ff fd  //              addi   $14, $00, -3
00 00 00 0d  //              break
20 06 00 c8  // slt_p3:      addi   $06, $00,
0xC8
ad e6 00 08  //              sw     $06,
0x08($15)

2c 43 00 8b  //              sltiu  $03, $02,
0x008B
14 60 00 02  //              bne    $03, $00,
slt1_p4
20 0e ff fc  //              addi   $14, $00, -4
00 00 00 0d  //              break
20 07 00 cc  // slt1_p4:     addi   $07, $00,
0xCC
ad e7 00 0c  //              sw     $07,
0x0C($15)
```

```
2c 43 00 89  //              sltiu  $03, $02,
0x0089
10 60 00 02  //              beq    $03, $00,
slt_p5
20 0e ff fb  //              addi   $14, $00, -5
00 00 00 0d  //              break
20 08 00 d0  // slt_p5:      addi   $08, $00,
0xD0
ad e8 00 10  //              sw     $08
0x10($15)
2c 43 00 8a  //              sltiu  $03, $02,
0x008A
10 60 00 02  //              beq    $03, $00,
slt_p6
20 0e ff fa  //              addi   $14, $00, -6
00 00 00 0d  //              break
20 06 00 d4  // slt_p6:      addi   $06, $00,
0xD4
ad e6 00 14  //              sw     $06,
0x14($15)
20 0e 00 00  //              addi   $14, $00, 0
03 e0 00 08  //              jr   $31
```

```
time: 1921.0 ns $r[0]:  00000000
time: 1931.0 ns $r[1]:  ffffff8a
time: 1941.0 ns $r[2]:  0000008a
time: 1951.0 ns $r[3]:  00000000
time: 1961.0 ns $r[4]:  000000c0
time: 1971.0 ns $r[5]:  000000c4
time: 1981.0 ns $r[6]:  000000d4
time: 1991.0 ns $r[7]:  0000f0f0
time: 2001.0 ns $r[8]:  00000000
time: 2011.0 ns $r[9]:  ffffffff
time: 2021.0 ns $r[10]: 0000f0f0
time: 2031.0 ns $r[11]: ffffffff
time: 2041.0 ns $r[12]: ffffaf5
time: 2051.0 ns $r[13]: ffff5555
time: 2061.0 ns $r[14]: 00000000
time: 2071.0 ns $r[15]: 100100c0

time: 2081.0 ns $r[16]: xxxxxxxx
time: 2091.0 ns $r[17]: xxxxxxxx
time: 2101.0 ns $r[18]: xxxxxxxx
time: 2111.0 ns $r[19]: xxxxxxxx
time: 2121.0 ns $r[20]: xxxxxxxx
```

```
time: 2131.0 ns $r[21]: xxxxxxxx
time: 2141.0 ns $r[22]: xxxxxxxx
time: 2151.0 ns $r[23]: xxxxxxxx
time: 2161.0 ns $r[24]: xxxxxxxx
time: 2171.0 ns $r[25]: xxxxxxxx
time: 2181.0 ns $r[26]: xxxxxxxx
time: 2191.0 ns $r[27]: xxxxxxxx
time: 2201.0 ns $r[28]: xxxxxxxx
time: 2211.0 ns $r[29]: 000003fc
time: 2221.0 ns $r[30]: xxxxxxxx
time: 2231.0 ns $r[31]: 00000014
```

```
time=2231.0 ns DM[0c0]=000000c0
time=2231.0 ns DM[0c4]=000000c4
time=2231.0 ns DM[0c8]=000000c8
time=2231.0 ns DM[0cc]=000000cc
time=2231.0 ns DM[0d0]=000000d0
time=2231.0 ns DM[0d4]=000000d4
time=2231.0 ns DM[0d8]=ffffff8a
time=2231.0 ns DM[0dc]=xxxxxxxx
```

```
time=2231.0 ns DM[0e0]=xxxxxxxx
time=2231.0 ns DM[0e4]=xxxxxxxx
time=2231.0 ns DM[0e8]=xxxxxxxx
time=2231.0 ns DM[0ec]=xxxxxxxx
time=2231.0 ns DM[0f0]=xxxxxxxx
time=2231.0 ns DM[0f4]=xxxxxxxx
time=2231.0 ns DM[0f8]=xxxxxxxx
time=2231.0 ns DM[0fc]=xxxxxxxx
time=2231.0 ns DM[3fc]=xxxxxxxx
```

Registers 6 through 8 are rewritten in after the jump register instruction.

Module 12

```
@0
3c 0f 10 01  // main:      lui   $15, 0x1001
35 ef 00 c0  //            ori   $15, 0x00C0

20 01 ff 8a  //            addi $01, $00, -118
20 02 00 8a  //            addi $02  $00,  138
0c 10 00 08  //            jal   blt_tests
ad e1 00 18  //            sw    $01, 0x18($15)
ad e2 00 1c  //            sw    $02, 0x1C($15)
00 00 00 0d  //            break
```

```
time: 1251.0 ns $r[0]:  00000000
time: 1261.0 ns $r[1]:  ffffff8a
time: 1271.0 ns $r[2]:  0000008a
time: 1281.0 ns $r[3]:  000000c0
time: 1291.0 ns $r[4]:  000000c4
time: 1301.0 ns $r[5]:  000000c8
time: 1311.0 ns $r[6]:  000000cc
time: 1321.0 ns $r[7]:  000000d0
time: 1331.0 ns $r[8]:  000000d4
time: 1341.0 ns $r[9]:  xxxxxxxx
time: 1351.0 ns $r[10]: xxxxxxxx
time: 1361.0 ns $r[11]: xxxxxxxx
time: 1371.0 ns $r[12]: xxxxxxxx
time: 1381.0 ns $r[13]: xxxxxxxx
time: 1391.0 ns $r[14]: 00000000
time: 1401.0 ns $r[15]: 100100c0
time: 1411.0 ns $r[16]: xxxxxxxx
time: 1421.0 ns $r[17]: xxxxxxxx
time: 1431.0 ns $r[18]: xxxxxxxx
time: 1441.0 ns $r[19]: xxxxxxxx
time: 1451.0 ns $r[20]: xxxxxxxx
time: 1461.0 ns $r[21]: xxxxxxxx
time: 1471.0 ns $r[22]: xxxxxxxx
time: 1481.0 ns $r[23]: xxxxxxxx
```

```
time: 1491.0 ns $r[24]: xxxxxxxx
time: 1501.0 ns $r[25]: xxxxxxxx
time: 1511.0 ns $r[26]: xxxxxxxx
time: 1521.0 ns $r[27]: xxxxxxxx
time: 1531.0 ns $r[28]: xxxxxxxx
time: 1541.0 ns $r[29]: 000003fc
time: 1551.0 ns $r[30]: xxxxxxxx
time: 1561.0 ns $r[31]: 00000014

time=1561.0 ns DM[0c0]=000000c0
time=1561.0 ns DM[0c4]=000000c4
time=1561.0 ns DM[0c8]=000000c8
time=1561.0 ns DM[0cc]=000000cc
```

```
time=1561.0 ns DM[0d0]=000000d0
time=1561.0 ns DM[0d4]=000000d4
time=1561.0 ns DM[0d8]=ffffff8a
time=1561.0 ns DM[0dc]=0000008a
time=1561.0 ns DM[0e0]=xxxxxxxx
time=1561.0 ns DM[0e4]=xxxxxxxx
time=1561.0 ns DM[0e8]=xxxxxxxx
time=1561.0 ns DM[0ec]=xxxxxxxx
time=1561.0 ns DM[0f0]=xxxxxxxx
time=1561.0 ns DM[0f4]=xxxxxxxx
time=1561.0 ns DM[0f8]=xxxxxxxx
time=1561.0 ns DM[0fc]=xxxxxxxx
time=1561.0 ns DM[3fc]=xxxxxxxx
```

```
18 20 00 02  // blt_tests: blez $01, blez_p1
20 0e ff ff  //            addi $14, $00, -1
00 00 00 0d  //            break
20 03 00 c0  // blez_p1:   addi $03, $00, 0xC0
ad e3 00 00  //            sw   $03, 0x00($15)
18 40 00 03  //            blez $02, blez_f2
20 04 00 c4  //            addi $04, $00, 0xC4
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
00 00 00 0d  //            break
18 00 00 02  // blez_p2:   blez $0, blez_p3
20 0e ff fd  //            addi $14, $00, -3
00 00 00 0d  //            break
20 05 00 c8  // blez_p3:   addi $05, $00, 0xC8
ad e5 00 08  //            sw   $05, 0x08($15)

1c 40 00 02  //            bgtz $02, bgtz_p1
20 0e ff fc  //            addi $14, $00, -4
00 00 00 0d  //            break
20 06 00 cc  // bgtz_p1:   addi $06, $00, 0xCC
ad e6 00 0c  //            sw   $06, 0x0C($15)
1c 20 00 03  //            bgtz $01, bgtz_f2
20 07 00 d0  //            addi $07, $00, 0xD0
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
00 00 00 0d  //            break
1c 20 00 03  // bgtz_p2:   bgtz $01, bgtz_f3
20 08 00 d4  //            addi $08, $00, 0xD4
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
00 00 00 0d  //            break
20 0e 00 00  // bgtz_p3:   addi $14, $00, 0
03 e0 00 08  //            jr   $31
```

```
time: 1251.0 ns $r[0]:  00000000
time: 1261.0 ns $r[1]:  ffffff8a
time: 1271.0 ns $r[2]:  0000008a
time: 1281.0 ns $r[3]:  000000c0
time: 1291.0 ns $r[4]:  000000c4
time: 1301.0 ns $r[5]:  000000c8
time: 1311.0 ns $r[6]:  000000cc
time: 1321.0 ns $r[7]:  000000d0
time: 1331.0 ns $r[8]:  000000d4
time: 1341.0 ns $r[9]:  xxxxxxxx
time: 1351.0 ns $r[10]: xxxxxxxx
time: 1361.0 ns $r[11]: xxxxxxxx
time: 1371.0 ns $r[12]: xxxxxxxx
time: 1381.0 ns $r[13]: xxxxxxxx
time: 1391.0 ns $r[14]: 00000000
time: 1401.0 ns $r[15]: 100100c0
time: 1411.0 ns $r[16]: xxxxxxxx
time: 1421.0 ns $r[17]: xxxxxxxx
time: 1431.0 ns $r[18]: xxxxxxxx
time: 1441.0 ns $r[19]: xxxxxxxx
time: 1451.0 ns $r[20]: xxxxxxxx
time: 1461.0 ns $r[21]: xxxxxxxx
time: 1471.0 ns $r[22]: xxxxxxxx
time: 1481.0 ns $r[23]: xxxxxxxx
time: 1491.0 ns $r[24]: xxxxxxxx
time: 1501.0 ns $r[25]: xxxxxxxx
time: 1511.0 ns $r[26]: xxxxxxxx
time: 1521.0 ns $r[27]: xxxxxxxx
time: 1531.0 ns $r[28]: xxxxxxxx
time: 1541.0 ns $r[29]: 000003fc
time: 1551.0 ns $r[30]: xxxxxxxx
time: 1561.0 ns $r[31]: 00000014
```

```
time=1561.0 ns DM[0c0]=000000c0                    time=1561.0 ns DM[0e4]=xxxxxxxx
time=1561.0 ns DM[0c4]=000000c4                    time=1561.0 ns DM[0e8]=xxxxxxxx
time=1561.0 ns DM[0c8]=000000c8                    time=1561.0 ns DM[0ec]=xxxxxxxx
time=1561.0 ns DM[0cc]=000000cc                    time=1561.0 ns DM[0f0]=xxxxxxxx
time=1561.0 ns DM[0d0]=000000d0                    time=1561.0 ns DM[0f4]=xxxxxxxx
time=1561.0 ns DM[0d4]=000000d4                    time=1561.0 ns DM[0f8]=xxxxxxxx
time=1561.0 ns DM[0d8]=ffffff8a                    time=1561.0 ns DM[0fc]=xxxxxxxx
time=1561.0 ns DM[0dc]=0000008a                    time=1561.0 ns DM[3fc]=xxxxxxxx
time=1561.0 ns DM[0e0]=xxxxxxxx
```

This module tests the branch is less than zero and branch if greater than zero operations. If all tests operate as intended data memory at 0c0 through 0d4 should be written with the values of register 3 through 8 respectively and data memory at 0d8 through 0dc should be written with registers 1 and 2 respectively.

Module 13

```
@0
00 00 00 1f  // main:       setie
3c 01 12 34  //             lui  $01, 0x1234
34 21 56 78  //             ori  $01, 0x5678
3c 02 87 65  //             lui  $02, 0x8765
34 42 43 21  //             ori  $02, 0x4321
3c 03 ab cd  //             lui  $03, 0xABCD
34 63 ef 01  //             ori  $03, 0xEF01
3c 04 01 fe  //             lui  $04, 0x01FE
34 84 dc ba  //             ori  $04, 0xDCBA
3c 05 5a 5a  //             lui  $05, 0x5A5A
34 a5 5a 5a  //             ori  $05, 0x5A5A
3c 06 ff ff  //             lui  $06, 0xFFFF
34 c6 ff ff  //             ori  $06, 0xFFFF
3c 07 ff ff  //             lui  $07, 0xFFFF
34 e7 ff 00  //             ori  $07, 0xFF00

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
ac ef 00 00  //             sw   $15, 0($07)
00 00 00 0d  //             break

@200
3c 10 10 01  // isr:        lui    $16, 0x1001
36 10 00 c0  //             ori    $16, 0x00C0
3c 11 80 00  //             lui    $17, 0x8000
36 31 ff ff  //             ori    $17, 0xFFFF
20 12 00 10  //             addi   $18, $0,
0x10

76 11 00 00  // out_IO:     output $17,  0($16)
00 11 88 83  //             sra    $17, $17, 2
22 10 00 04  //             addi   $16, $16, 4
22 52 ff ff  //             addi   $18, $18, -1
16 40 ff fb  //             bne    $18, $00,
out_IO

3c 10 10 01  //             lui    $16, 0x1001
36 10 00 c0  //             ori    $16, 0x00C0
72 13 00 00  //             input  $19,  0($16)
72 14 00 04  //             input  $20,  4($16)
72 15 00 08  //             input  $21,  8($16)
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
```

```
time: 4971.0 ns $r[0]:  00000000
time: 4981.0 ns $r[1]:  12345678
time: 4991.0 ns $r[2]:  87654321
time: 5001.0 ns $r[3]:  abcdef01
time: 5011.0 ns $r[4]:  01fedcba
time: 5021.0 ns $r[5]:  5a5a5a5a
time: 5031.0 ns $r[6]:  ffffffff
time: 5041.0 ns $r[7]:  100103f0
time: 5051.0 ns $r[8]:  fffffeff
time: 5061.0 ns $r[9]:  fffffefe
time: 5071.0 ns $r[10]: fffffefd
time: 5081.0 ns $r[11]: fffffefc
time: 5091.0 ns $r[12]: fffffefb
time: 5101.0 ns $r[13]: fffffefa
time: 5111.0 ns $r[14]: fffffef9
time: 5121.0 ns $r[15]: fffffef8
time: 5131.0 ns $r[16]: 100100c0
time: 5141.0 ns $r[17]: ffffffff
time: 5151.0 ns $r[18]: 00000000
time: 5161.0 ns $r[19]: 8000ffff
time: 5171.0 ns $r[20]: e0003fff
time: 5181.0 ns $r[21]: f8000fff
time: 5191.0 ns $r[22]: fe0003ff
time: 5201.0 ns $r[23]: ff8000ff
time: 5211.0 ns $r[24]: ffe0003f
time: 5221.0 ns $r[25]: xxxxxxxx
time: 5231.0 ns $r[26]: xxxxxxxx
time: 5241.0 ns $r[27]: xxxxxxxx
time: 5251.0 ns $r[28]: xxxxxxxx
time: 5261.0 ns $r[29]: 000003fc
time: 5271.0 ns $r[30]: xxxxxxxx
time: 5281.0 ns $r[31]: 00000064

time=5281.0 ns DM[3fc]=00000200

time=5281.0 ns IOM[0c0]=8000ffff
time=5281.0 ns IOM[0c4]=e0003fff
time=5281.0 ns IOM[0c8]=f8000fff
time=5281.0 ns IOM[0cc]=fe0003ff
time=5281.0 ns IOM[0d0]=ff8000ff
time=5281.0 ns IOM[0d4]=ffe0003f
time=5281.0 ns IOM[0d8]=fff8000f
time=5281.0 ns IOM[0dc]=fffe0003
time=5281.0 ns IOM[0e0]=ffff8000
time=5281.0 ns IOM[0e4]=ffffe000
time=5281.0 ns IOM[0e8]=fffff800
time=5281.0 ns IOM[0ec]=fffffe00
time=5281.0 ns IOM[0f0]=ffffff80
time=5281.0 ns IOM[0f4]=ffffffe0
time=5281.0 ns IOM[0f8]=fffffff8
time=5281.0 ns IOM[0fc]=fffffffe
```

This module tests the input output module and operations. The output to I/O should be similar to that of the data memory in module 3 with the corresponding instruction change from load and store to input and output.

Module 14

```
@0
00 00 00 1f  // main:      setie
3c 01 12 34  //            lui  $01, 0x1234
34 21 56 78  //            ori  $01, 0x5678
3c 02 87 65  //            lui  $02, 0x8765
34 42 43 21  //            ori  $02, 0x4321
3c 03 ab cd  //            lui  $03, 0xABCD
34 63 ef 01  //            ori  $03, 0xEF01
3c 04 01 fe  //            lui  $04, 0x01FE
34 84 dc ba  //            ori  $04, 0xDCBA
3c 05 5a 5a  //            lui  $05, 0x5A5A
34 a5 5a 5a  //            ori  $05, 0x5A5A
3c 06 ff ff  //            lui  $06, 0xFFFF
34 c6 ff ff  //            ori  $06, 0xFFFF
3c 07 ff ff  //            lui  $07, 0xFFFF
34 e7 ff 00  //            ori  $07, 0xFF00

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
ac ef 00 00  //            sw   $15, 0($07)
00 00 00 0d  //            break

@200
3c 10 10 01  // isr:       lui     $16, 0x1001
36 10 00 c0  //            ori     $16, 0x00C0
3c 11 80 00  //            lui     $17, 0x8000
36 31 ff ff  //            ori     $17, 0xFFFF
20 12 00 10  //            addi    $18, $0,
0x10

76 11 00 00  // out_IO:    output  $17, 0($16)
00 11 88 83  //            sra     $17, $17, 2
22 10 00 04  //            addi    $16, $16, 4
22 52 ff ff  //            addi    $18, $18, -1
16 40 ff fb  //            bne     $18, $00,
out_IO

3c 10 10 01  //            lui     $16, 0x1001
36 10 00 c0  //            ori     $16, 0x00C0
72 13 00 00  //            input   $19,  0($16)
72 14 00 04  //            input   $20,  4($16)
72 15 00 08  //            input   $21,  8($16)
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
```

```
time: 5071.0 ns $r[0]:  00000000
time: 5081.0 ns $r[1]:  12345678
time: 5091.0 ns $r[2]:  87654321
time: 5101.0 ns $r[3]:  abcdef01
time: 5111.0 ns $r[4]:  01fedcba
time: 5121.0 ns $r[5]:  5a5a5a5a
time: 5131.0 ns $r[6]:  ffffffff
time: 5141.0 ns $r[7]:  100103f0
time: 5151.0 ns $r[8]:  fffffeff
time: 5161.0 ns $r[9]:  fffffefe
time: 5171.0 ns $r[10]: fffffefd
time: 5181.0 ns $r[11]: fffffefc
time: 5191.0 ns $r[12]: fffffefb
time: 5201.0 ns $r[13]: fffffefa
time: 5211.0 ns $r[14]: fffffef9
time: 5221.0 ns $r[15]: fffffef8

time: 5231.0 ns $r[16]: 100100c0
time: 5241.0 ns $r[17]: ffffffff
time: 5251.0 ns $r[18]: 00000000
time: 5261.0 ns $r[19]: 8000ffff
time: 5271.0 ns $r[20]: e0003fff
time: 5281.0 ns $r[21]: f8000fff
time: 5291.0 ns $r[22]: fe0003ff
time: 5301.0 ns $r[23]: ff8000ff
time: 5311.0 ns $r[24]: ffe0003f
time: 5321.0 ns $r[25]: xxxxxxxx
time: 5331.0 ns $r[26]: xxxxxxxx
time: 5341.0 ns $r[27]: xxxxxxxx
time: 5351.0 ns $r[28]: xxxxxxxx
time: 5361.0 ns $r[29]: 000003fc
time: 5371.0 ns $r[30]: xxxxxxxx
time: 5381.0 ns $r[31]: xxxxxxxx

time=5381.0 ns DM[3f0]=fffffef8

time=5381.0 ns DM[3f4]=f1a95000
time=5381.0 ns DM[3f8]=00000064
time=5381.0 ns DM[3fc]=00000200

time=5381.0 ns IOM[0c0]=8000ffff
time=5381.0 ns IOM[0c4]=e0003fff
time=5381.0 ns IOM[0c8]=f8000fff
time=5381.0 ns IOM[0cc]=fe0003ff
time=5381.0 ns IOM[0d0]=ff8000ff
time=5381.0 ns IOM[0d4]=ffe0003f
time=5381.0 ns IOM[0d8]=fff8000f
time=5381.0 ns IOM[0dc]=fffe0003
time=5381.0 ns IOM[0e0]=ffff8000
time=5381.0 ns IOM[0e4]=ffffe000
time=5381.0 ns IOM[0e8]=fffff800
time=5381.0 ns IOM[0ec]=fffffe00
time=5381.0 ns IOM[0f0]=ffffff80
time=5381.0 ns IOM[0f4]=ffffffe0
time=5381.0 ns IOM[0f8]=fffffff8
time=5381.0 ns IOM[0fc]=fffffffe
```

This module tests the I/O operations using the save to stack method of storing PC which is highlighted in red. All other outputs should be similar to that of module 13.

Enhanced Operations Module

```
@0
3c 01 12 34  //             lui   $01, 0x1234
34 21 56 78  //             ori   $01, 0x5678
3c 02 87 65  //             lui   $02, 0x8765
34 42 43 21  //             ori   $02, 0x4321
3c 03 ab cd  //             lui   $03, 0xABCD
34 63 ef 01  //             ori   $03, 0xEF01
3c 04 01 fe  //             lui   $04, 0x01FE
34 84 dc ba  //             ori   $04, 0xDCBA
3c 05 5a 5a  //             lui   $05, 0x5A5A
34 a5 5a 5a  //             ori   $05, 0x5A5A
3c 06 40 09  //             lui   $06, 0x4009
34 c6 21 fb  //             ori   $06, 0x21FB
3c 07 54 44  //             lui   $07, 0x5444
34 e7 2d 11  //             ori   $07, 0x2D11
3c 08 40 00  //             lui   $08, 0x4000
3c 0f 10 01  //             lui   $15, 0x1001
35 ef 00 00  //             ori   $15, 0x0000
8d ed 00 04  //             lw    $13, 04($15)
8d ee 00 00  //             lw    $14, 00($15)
8d eb 00 0C  //             lw    $11, 12($15)
8d ec 00 08  //             lw    $12, 08($15)
8d ea 00 10  //             lw    $10, 16($15)
8d e9 00 14  //             lw    $9,  14($15)

7C 22 00 00  //             MVFR  $00, $01, $02
7C C7 08 00  //             MVFR  $01, $06, $07
7D 00 10 00  //             MVFR  $02, $08, $00
7C 22 18 01  //             FMULT $03, $01, $02
7C 61 20 02  //             FDIV  $04, $03, $02
7C 61 28 03  //             FADD  $05, $03, $01
7C 41 30 04    //           FSUB  $06, $02, $01
7C 26 38 05  //             FZERO $07, $01, $06

7C 26 00 06  //             MVVR  $00, $01,$06
7d ce 08 06  //             MVVR  $01, $14,$14
7d ad 10 06  //             MVVR  $02, $13,$13
7c 22 80 07  //             VADDS $16, $01,$02

7d 6c 18 06  //             MVVR   $03, $11,$12
7d 6a 20 06  //             MVVR   $04, $11,$10
7c 0b 28 06  //             MVVR   $05, $11,$11
7c 64 28 08  //             MULADD $05, $3,$4

7d 29 48 06  //             MVVR $9, $9, $9
7d 30 30 09  //             VAND  $6,  $9, $16
7d 30 38 0a  //             VCEQ   $7,  $9, $16
7d 30 40 0b  //             VCLT  $$8, $9, $16

00 00 00 0d  //             break
```

```
time: 1691.0 ns       $r[0]: 00000000
time: 1701.0 ns       $r[1]: 12345678
time: 1711.0 ns       $r[2]: 87654321
time: 1721.0 ns       $r[3]: abcdef01
time: 1731.0 ns       $r[4]: 01fedcba
time: 1741.0 ns       $r[5]: 5a5a5a5a
time: 1751.0 ns       $r[6]: 400921fb
time: 1761.0 ns       $r[7]: 54442d11
time: 1771.0 ns       $r[8]: 40000000
time: 1781.0 ns       $r[9]: ffffffff
time: 1791.0 ns       $r[10]: 00000004
time: 1801.0 ns       $r[11]: 00000005
time: 1811.0 ns       $r[12]: 00000007
time: 1821.0 ns       $r[13]: 05060506
time: 1831.0 ns       $r[14]: fd03fd03
time: 1841.0 ns       $r[15]: 10010000
time: 2011.0 ns       $f[0]: 1234567887654321
time: 2021.0 ns       $f[1]: 400921fb54442d11
time: 2031.0 ns       $f[2]: 4000000000000000
time: 2041.0 ns       $f[3]: 401921fb54442d11
time: 2051.0 ns       $f[4]: 4000000000000000
time: 2061.0 ns       $f[5]: 4022d97c7f3321cd
time: 2071.0 ns       $f[6]: bff243f6a8885a22
time: 2081.0 ns       $f[7]: 0000000000000000
time: 2091.0 ns       $f[8]: xxxxxxxxxxxxxxxx
time: 2101.0 ns       $f[9]: xxxxxxxxxxxxxxxx
time: 2111.0 ns       $f[10]: xxxxxxxxxxxxxxxx
time: 2121.0 ns       $f[11]: xxxxxxxxxxxxxxxx
time: 2131.0 ns       $f[12]: xxxxxxxxxxxxxxxx
time: 2141.0 ns       $f[13]: xxxxxxxxxxxxxxxx
time: 2151.0 ns       $f[14]: xxxxxxxxxxxxxxxx
time: 2161.0 ns       $f[15]: xxxxxxxxxxxxxxxx

The double at F[$01] is 3.141593
The double at F[$02] is 2.000000
The double at F[$03] is 6.283185
The double at F[$04] is 2.000000
The double at F[$05] is 9.424778
The double at F[$06] is -1.141593
The double at F[$07] is 0.000000

time: 2331.0 ns       $v[0]: 12345678400921fb
time: 2341.0 ns       $v[1]: fd03fd03fd03fd03
time: 2351.0 ns       $v[2]: 0506050605060506
time: 2361.0 ns       $v[3]: 0000000500000007
time: 2371.0 ns       $v[4]: 0000000500000004
time: 2381.0 ns       $v[5]: 0000001e00000021
time: 2391.0 ns       $v[6]: ff09ff09ff09ff09
time: 2401.0 ns       $v[7]: ff00ff00ff00ff00
time: 2411.0 ns       $v[8]: 0000000000000000
time: 2421.0 ns       $v[9]: ffffffffffffffff
time: 2431.0 ns       $v[10]: xxxxxxxxxxxxxxxx
time: 2441.0 ns       $v[11]: xxxxxxxxxxxxxxxx
time: 2451.0 ns       $v[12]: xxxxxxxxxxxxxxxx
time: 2461.0 ns       $v[13]: xxxxxxxxxxxxxxxx
time: 2471.0 ns       $v[14]: xxxxxxxxxxxxxxxx
time: 2481.0 ns       $v[15]: xxxxxxxxxxxxxxxx
time: 2491.0 ns       $v[16]: ff09ff09ff09ff09
```

This module tests the Enhanced instruction set of our project. Integer register values are copied to floating point registers as well as vector registers then operations are done on the values in their respective datapaths using register-type operations. The bitwise conversion to real numbers is displayed for comparison for the double precision floating-point.

D. Data memory reference logs

DM 1-7 modules

```
@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
```

DM 13-14 modules

```
@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
```

DM 8 module

```
@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
```

DM 9-12 modules

```
@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
```

DM Enhanced module

```
@0
FD 03 FD 03
05 06 05 06
00 00 00 07
00 00 00 05
00 00 00 04
FF FF FF FF
00 00 00 20
00 00 00 25
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
```

IV. Hardware Implementation Diagrams

V. Additional Discussions or Comments

Future Enhancements

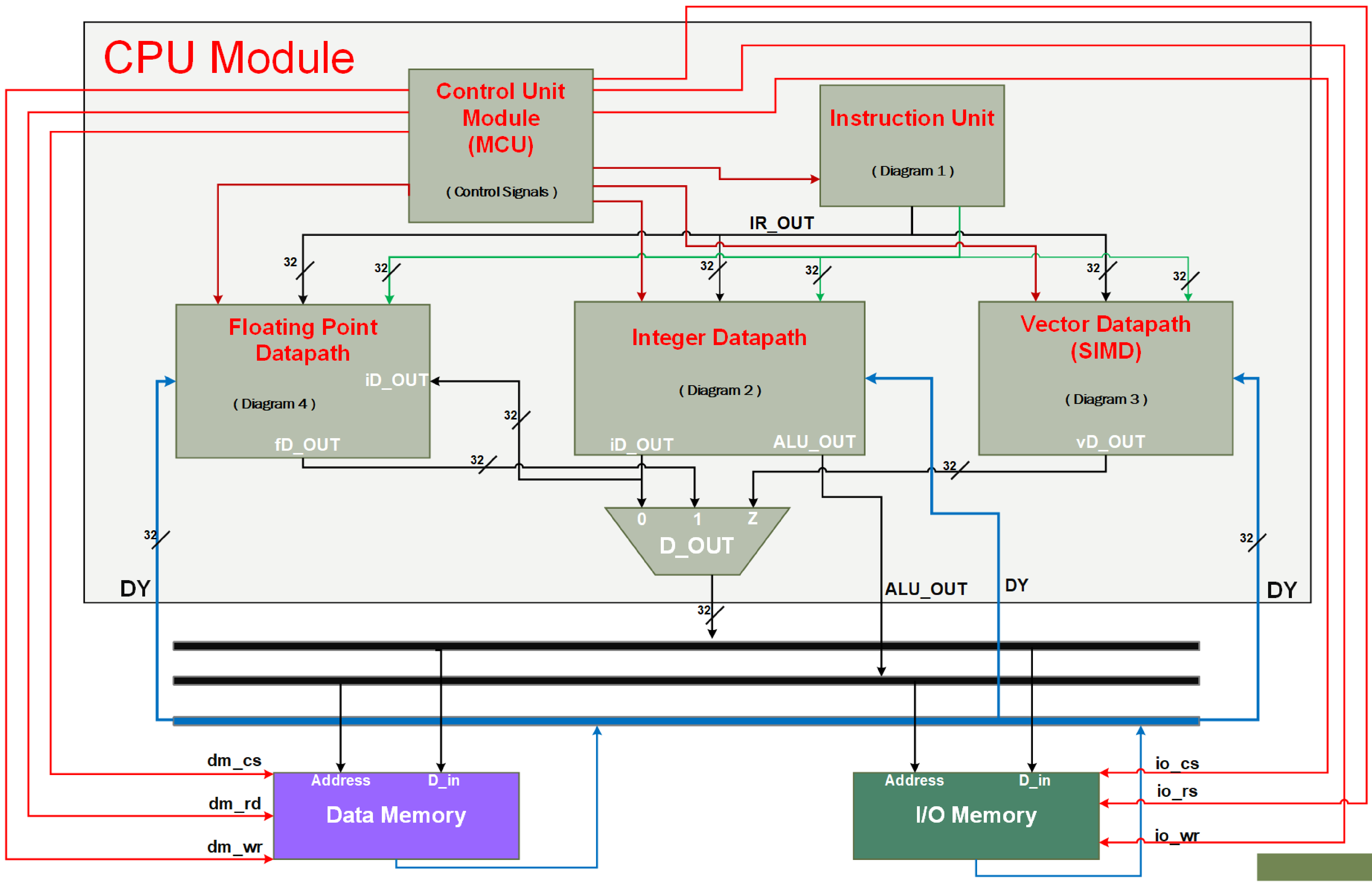
Will include pipelining the GBRAINS CPU, expanding the memory usage operations for the enhance registers. Also to increase the memory access capabilities of the enhanced registers. Expand and include more operations.

Builder

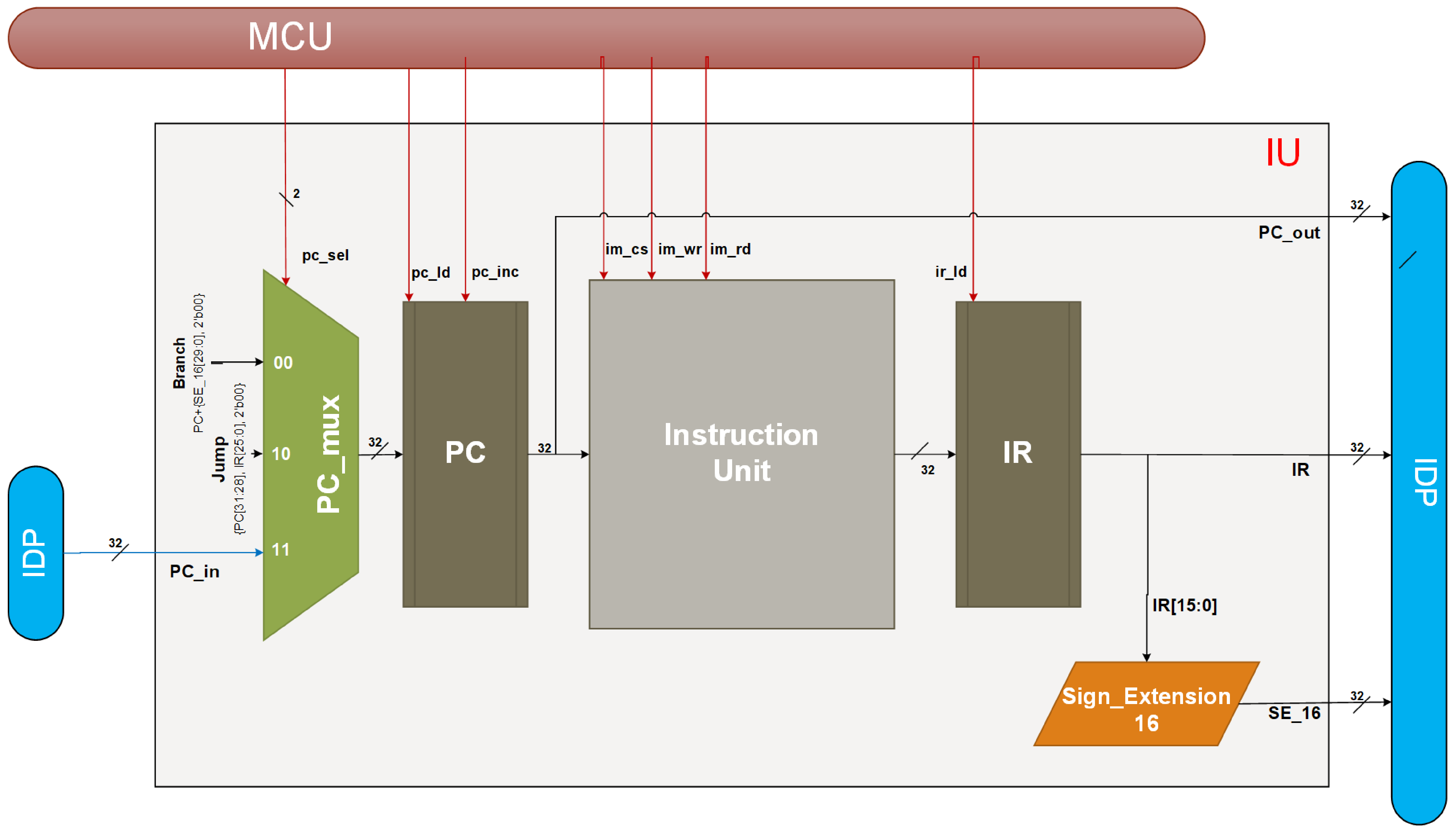
We used the following module as a tool to write out custom instructions to reduce the stress of fiddling with bits on the windows calculator, which is a terrible little program.

```
`timescale 1ns / 1ps
/****************************************************************************
* Author(s):Bryan Linares
*           Grace Daliwan
*           Brian Ortiz
* Filename: Builder.v
* Date:     Nov. 17, 2018
* Project:  CECS 440 Senior Project
*
* Notes:    Tool module: efficient way to print out custom instructions for our
*                        enhanced IMem file.
*
****************************************************************************/
module builder();

    reg  [ 4:0] rs,rt,rd,fmt;
    reg  [ 5:0] funct, hfunct, op;
    reg  [15:0] imm;

    wire         c;
    wire [ 7:0]  test;
    wire [31:0]  IR, IRI;

    assign IR = {op,rs,rt,rd,fmt,funct};
    assign IRI= {op, rs, rt, imm};
    assign {c,test} = 8'hFD + 8'h05;

    initial begin
    op    = 6'h1F;
    rs    = 5'h9;
    rt    = 5'h10;
    rd    = 5'h8;
    fmt   = 5'h0;
    funct = 6'hB;

    #100;
    $display("%h", IR);
    $display("IR %h, es %h, et %h, ed %h, fmt %h, funct %h ",
               IR, IR[25:21],IR[20:16],IR[15:11],IR[10:6],IR[5:0]);

    op    = 6'h1F;
    rs    = 5'h9;
    rt    = 5'h10;
    rd    = 5'h7;
    fmt   = 5'h0;
    funct = 6'hA;

    #100;
    $display("%h", IR);

    end
endmodule
```

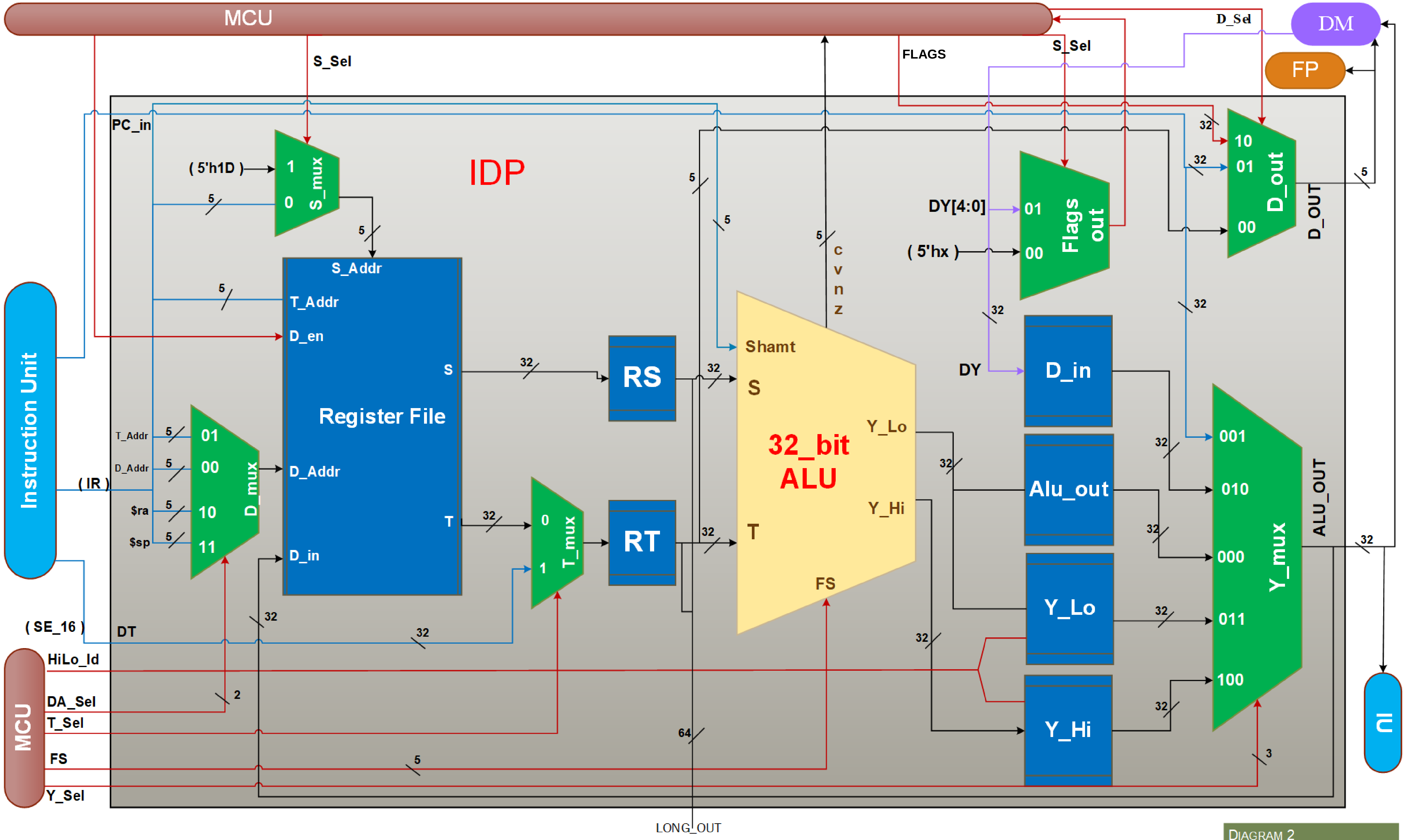
VI. CD-ROM


CPU Module
Control Unit Module (MCU)
(Control Signals)
Instruction Unit
(Diagram 1)
IR_OUT
32
Floating Point Datapath
(Diagram 4)
iD_OUT
fD_OUT
Integer Datapath
(Diagram 2)
iD_OUT
ALU_OUT
Vector Datapath (SIMD)
(Diagram 3)
vD_OUT
0
1
Z
D_OUT
DY
ALU_OUT
dm_cs
dm_rd
dm_wr
Address
D_in
Data Memory
io_cs
io_rs
io_wr
I/O Memory

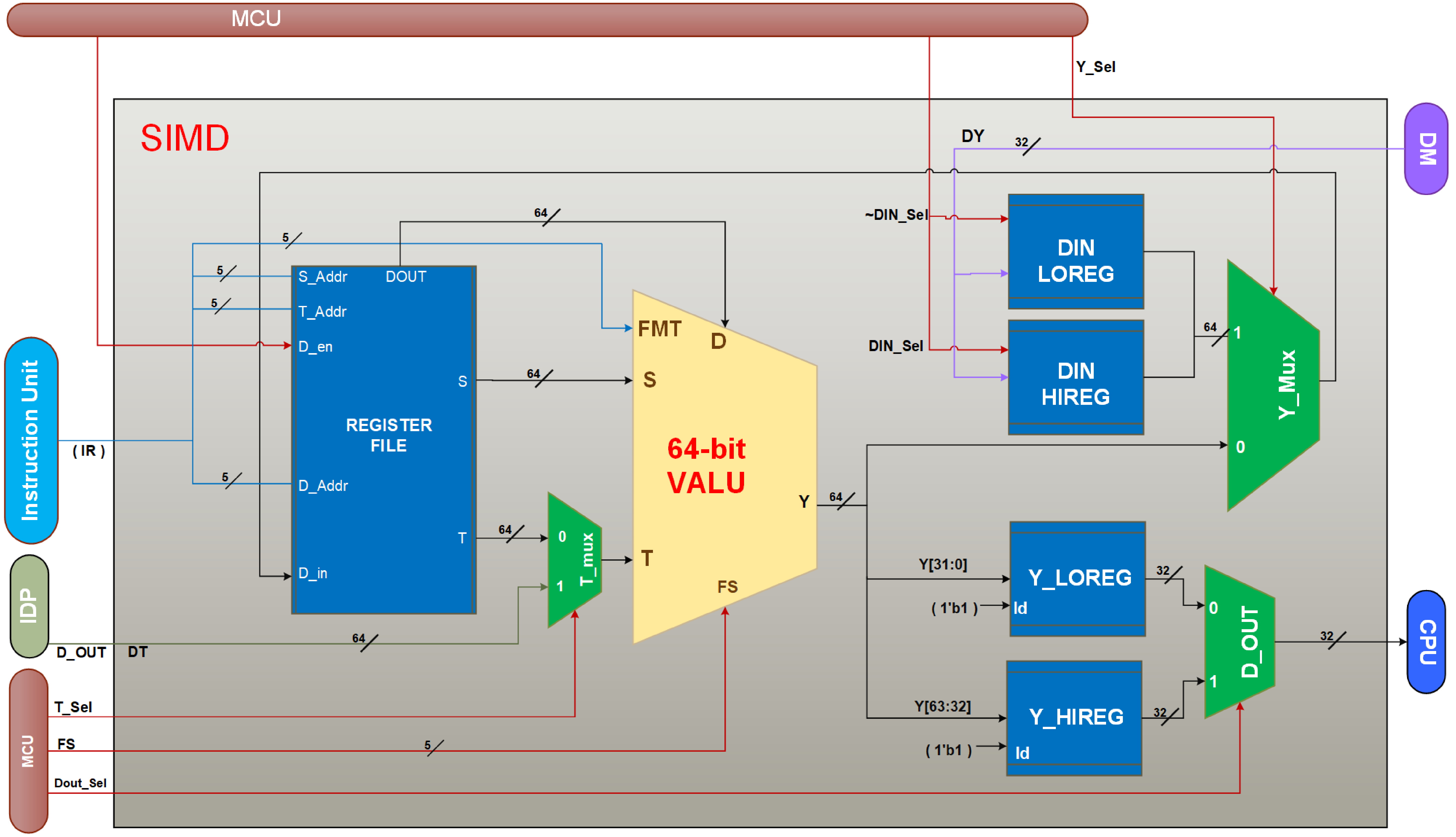
CPU: Instruction Unit



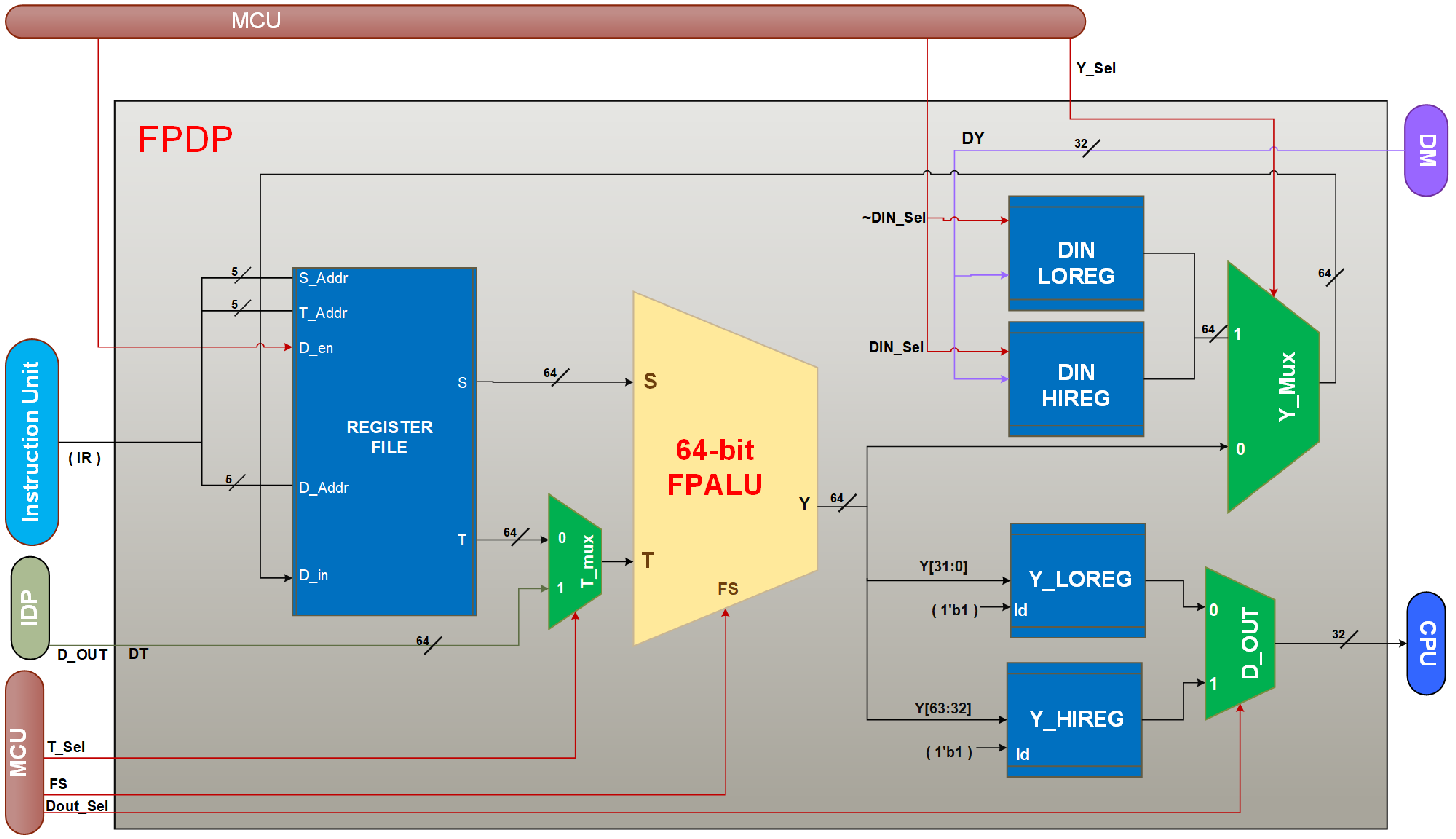
GBRAINS CPU: Integer Datapath



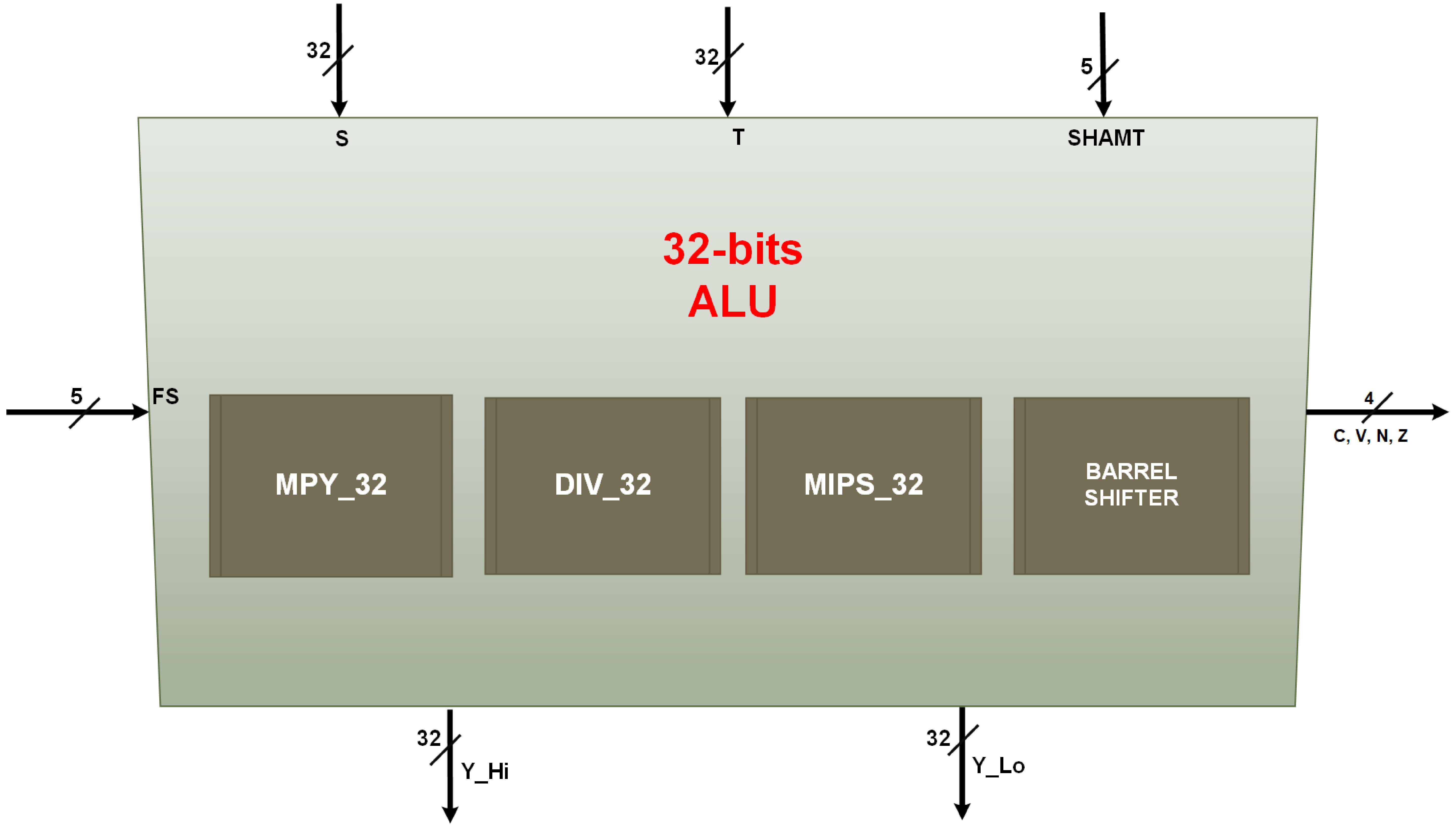
GBRAINS CPU: SIMD Datapath



CPU: Floating Point Datapath



ALU


32
S
32
T
5
SHAMT
32-bits
ALU
5
FS
MPY_32
DIV_32
MIPS_32
BARREL
SHIFTER
4
C, V, N, Z
32
Y_Hi
32
Y_Lo