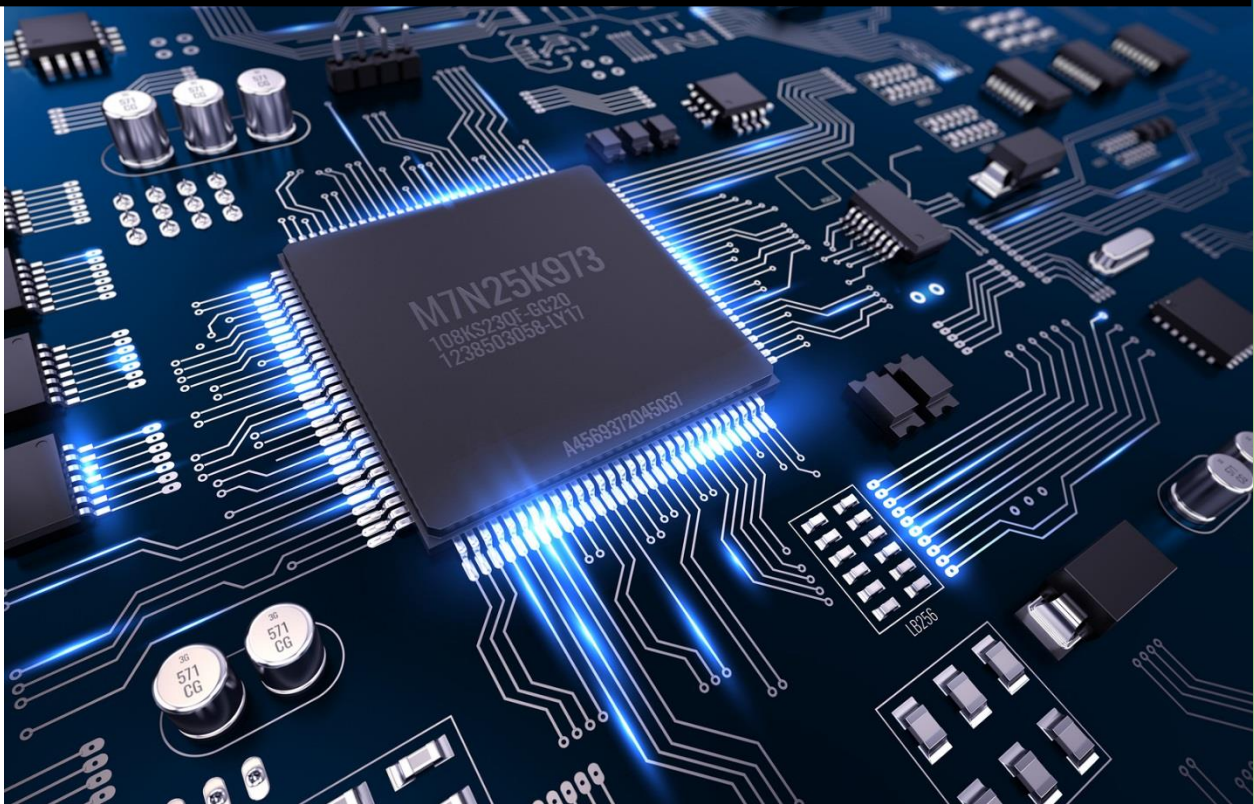


CECS 440

The Boiler Room MIPS Processor



Javier Garcia 012742702

Taylor Cochran 013330744

Class Time: 11am Tues/Thurs

Due: 11/27/2018

Instructor: R.W. Allison

Table of Contents

I. Purpose	2
II. Instruction Set Architecture	3
A. Harvard Memory Architecture and Organization	3
B. Machine Register Set	5
C. Data Types	7
1. Unsigned Integers	7
2. Signed Integers	7
D. Addressing Modes	8
1. Immediate	8
2. Register	9
3. Register Indirect	9
E. Instruction Set	10
1. R-Type Instructions	10
2. I-Type Instructions	12
3. J-Type Instructions	13
4. Enhanced Instructions	13
F. Instruction Format	52
1. R Type	52
2. I Type	54
3. J type	56
III. Verilog Implementation / Design / Verification	57
TOP LEVEL Source Code - 2 Pages	57
Modules and Sub Modules - 43 Pages	60
MEMORY MODULES - 3 Pages	105
ISIM Log Files - 32 Pages	109
IV. Hardware Implementation (10 Diagrams)	142
V. Additional Discussion/Comments	153
A look at the ISR	153
Works Cited	154

I. Purpose

The Instruction Set Architecture (ISA) for this 32-bit MIPS Processor was designed in the CECS 440 (Computer Architecture II) class in Fall of 2018 at California State University Long Beach. The details provided by this document will list the functionality of the MIPS Processor and all of its components. The research, effort, and ideas put into this project were done to further cultivate our knowledge as Computer Engineers, and provide an in depth experience with the design principles of Computer Architecture.

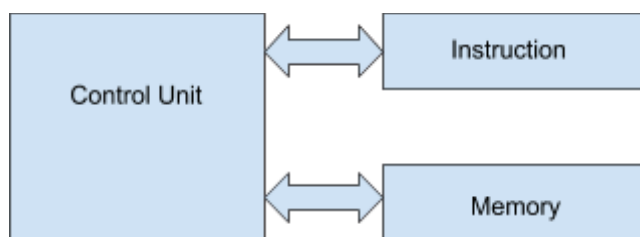
Any resource or information used that was not created by Javier or Taylor will be cited and credit provided to the original Author. Criteria and Grading for the project was done by Professor Allison. A total of seven Lab deadlines were made, with the 7th being the final due date. Each Lab checkpoint added in another piece of the MIPS Processor, starting with the ALU and finishing with the MIPS Control Unit. Each checkpoint was verified using a comprehensive Testbench in the Verilog Simulation. The breakdown of the Functionality, Structure, and Verification for each part of the Processor can be found using the Table of Contents.

II. Instruction Set Architecture

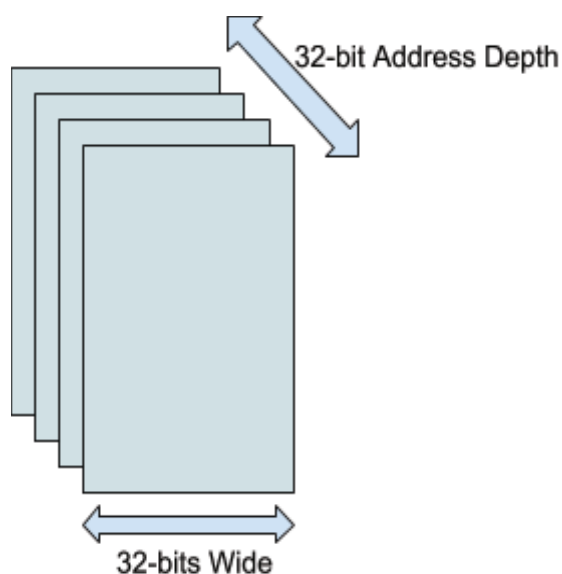
A. Harvard Memory Architecture and Organization

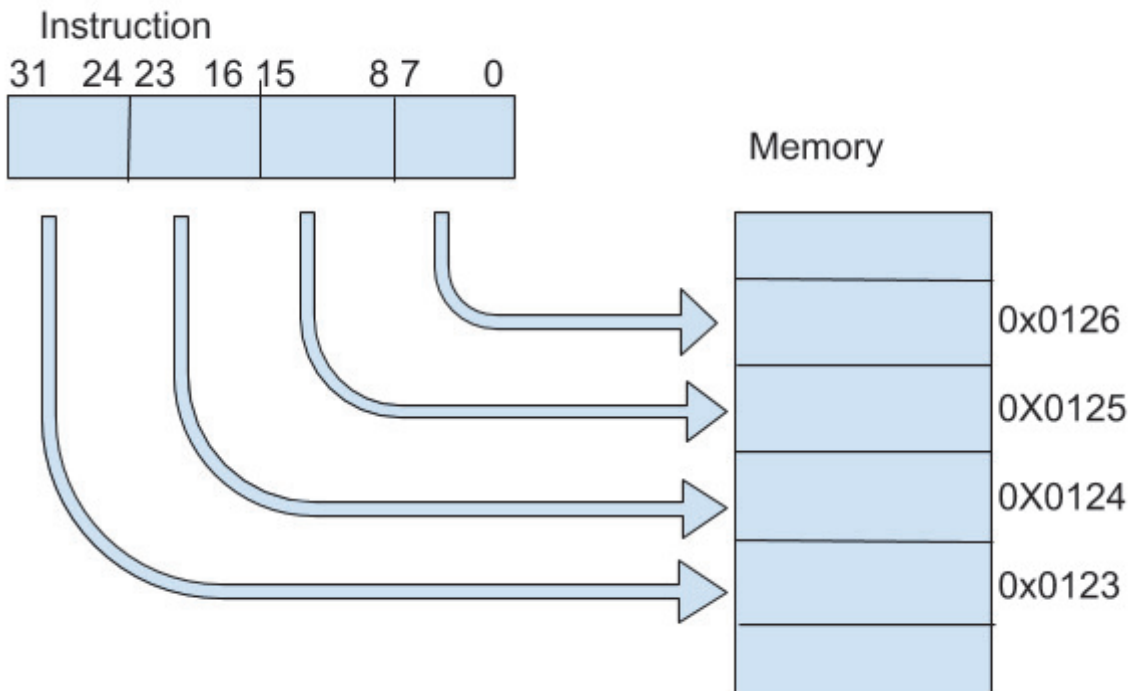
There are two main architectural styles used in implementing an ISA. For our purposes we rely on Harvard Style ISA. The main reason we choose to work with a Harvard ISA is because it has two separate busses, one is used for the instructions and the other is used to access the memory. The Von Neumann style shares a single bus for both functions.

The advantages of using a Harvard ISA are numerous. For one it allows for the system to access memory and perform an instruction in parallel. Increasing the speed and efficiency of the processor. To the right is a diagram showing the two separate busses that allow the control unit to run different signals in parallel.



The busses used in this particular design are 32-bits deep and 32-bits wide. Specifically talking about memory every location is 32-bits wide and the address depth is 32-bits as well. One thing to keep in mind for this MIPS RISC ISA is that the memory is byte addressable. Since one byte is equivalent to 8-bits our memory is broken up into 4-bytes per instruction. This means that every instruction in the memory will take up four address lines to store said instruction. Another thing to note about our memory is that it is stored in the Big Endian format, which takes the MSB will be stored in the lowest address. This is where the the four addresses come into play, the instruction is broken into four parts the msb gets stored in the lowest address and the lsb gets stored in the highest address. See example below.





Big Endian format where the MSB goes to lower locations and lower bits go to higher locations consecutively.

B. Machine Register Set

The MIPS Register File contains a total of 32 registers, each being 32 bits wide. All registers that are listed are available for use, but not all of them can be modified. There are 18 temporary registers that can be modified by the user and act as temporary or scratch registers. These plus the rest of the registers can be found in the table below.

Name	Number	Purpose	Preserved Across a Call?
\$zero	0	Constant value of 0	N.A.
\$at	1	Assembler Temporary	No
\$v0-\$v1	2-3	Values for Function Results and Expression Evaluation	No
\$a0-\$a3	4-7	Arguments	No
\$t0-\$t7	8-15	Temporaries	No
\$s0-\$s7	16-23	Saved Temporaries	Yes
\$t8-\$t9	24-25	Temporaries	No
\$k0-\$k1	26-27	Reserved for OS Kernel	No
\$gp	28	Global Pointer	Yes
\$sp	29	Stack Pointer	Yes
\$fp	30	Frame Pointer	Yes
\$ra	31	Return Address	Yes

There are three additional register types that this MIPS Processor relies on. The first being the Flags Reg, the second is the Program Counter (PC) Reg, and the third is the Instruction Register (IR). The Flags Register is what stores the 4 flags used to help determine operation outcomes. The flags are Carry, Negative, Zero, and Overflow.

The Carry flag is updated anytime an Instruction when a result being stored in the destination register is too large. Anything greater than 32 bits sets the Carry Flag to High.

The Negative flag is updated anytime an Instruction uses a signed number and the result is designated to be a signed number of a negative value (anything less than 0). The Negative flag will be set high in this case.

The Zero Flag is updated anytime an Instruction results in the destination register receiving a result of a zero.

The Overflow flag is updated when an Instruction has two signed operands and they roll over without carrying.

The second Register is the Program Counter (PC) which is a 32 bit wide value containing the address of the next Instruction in Memory to fetch. The Program Counter is Byte Addressable in MIPS, meaning it will always have an increment of four (0x00, 0x04, 0x08, 0x0C, 0x10) due to the Big Endian Memory design previously mentioned.

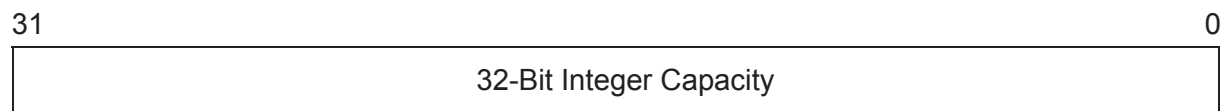
The third Register is the Instruction Register (IR) which is a 32 bit wide register containing the current Instruction. The PC points to the next instruction, which the contents of that address get loaded into the IR after the previous instruction executes. The current instruction is then decoded and executed. The Instruction Register has the machine code that can be read in Addressing mode format found below in section D.

C. Data Types

Although there are various possible data types that can be used in a given ISA, for our application desires we only work with two. Both data types are 32-bit integers but they vary in the range of numbers they represent.

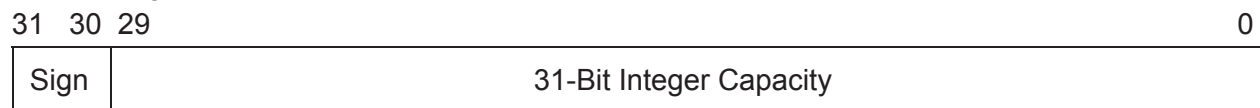
1. Unsigned Integers

Unsigned Integers are able to take full advantage of the 32-bits because they do not need to use a bit in order to represent their sign. This data type has a range [0,4,294,967,295].



2. Signed Integers

This data type have a lower range than its close relative the Unsigned integer. Signed integers need to use the MSB(Most Significant-Bit) in order to represent whether they number is a positive or a negative. The remaining 31 bits are used in order to represent the actual number whose range : [-2,147,483,648 , 2,147,483,647].



A thing to be aware about the signed integers is that the negative values are based on 2's complement. If sign is a 1 the integer is negative, otherwise it is considered a positive number.

D. Addressing Modes

The Address Modes listed below cover the different specifications for generating an Operand's address at runtime. These naming conventions are not unique to this processor, nor were they created for its sole use, but a generalization to better explain what occurs.

1. Immediate

The Immediate Address Mode is used in the occurrence of an I-type Instruction where the 16 least significant bits of the Operand are the designated value. The Immediate mode is most commonly used for Register Initialization or Constant values used in Assembly.

Example: $\$t0 \leftarrow \$zero + 0x188C$

Mem Address	Instruction	RT	RS	16 Bit Immediate
0x0124	addi	\$t0	\$zero	0x188C

The J-type instruction uses an Immediate mode as well when specifying a 26 bit address. The 26 least significant bits are concatenated with 2 bits of 0 value to make a Byte Addressable value and are further concatenated with the 4 most significant bits of the Program Counter's current value.

Example: $PC \leftarrow \{4'b[PC+4], 0x020188F, 2'b\ 00\}$

Mem Address	Instruction	26 Bit Immediate
0x0138	Jump	0x020188F

2. Register

The Register Addressing Mode is the most frequently used and the fastest type of addressing mode. It uses the contents of a specified register as the operand and is saved inside another Register. Any R-Type Instruction will always be Register Addressing Mode.

Example: $\$t0 \leftarrow \$zero + \$zero$

Mem Address	Instruction	RD	RS	RT
0x0124	add	\$t0	\$zero	\$zero

3. Register Indirect

The Register Indirect Addressing Mode is used for Load and Store instructions. A MIPS Processor cannot directly reference memory, it will have to store address locations inside a register in the Reg File. If larger than 16 bits is needed to specify the address, the Assembly Instructions will need to use a Lui Ori to Load two sets of 16 bits into the 32 bit register. Below an example will be provided of a Load Word Instruction using the Register Indirect Addressing Mode.

Example: $\$t0 \leftarrow \{16'b\ 0xFF13, 16'b\ 0\}$

Mem Address	Instruction	RS	RT	16 Bit Immediate
0x0124	Lui	0x0000	\$t0	0xFF13

E. Instruction Set

MIPS processors are able to take advantage of three different instruction types. This allows the processor flexibility in the form that it functions. Suppose you want to add two values, normally you'd add two register and then store them in the destination register. However if one of the values you wanted to add you'd have to take an additional step to store it before actually doing the arithmetic function. Now having an immediate type instruction eliminates the need to store the value beforehand. Just like this different instructions can be considered to be part of a different instruction set. All instructions sets are 32-bits the main difference lies when they are being read, they are broken up into different sections with varying lengths. Reference section F to see how the instruction types are formatted. The instructions sets used in this MIPS machine are shown below.

The following 4 subsections will each show a table containing all the corresponding instructions to the type. At the end of section E, there will be a page for each instruction containing an Example, a description of its functionality, the Machine code format, and the pseudo code.

1. R-Type Instructions

Instruction	Function code	Function
sll	0x00	$Rd = rt \ll shamt;$
srl	0x02	$Rd = shamt \gg rt$
sra	0x03	$Rd = Rt \ggg shamt \quad // Rt[31] \gg Rt$
jr	0x08	$PC = Rs$
mfhi	0x10	$Rd = HI$
mflo	0x12	$Rd = LO$
mult	0x18	$\{HI, LO\} = Rs * Rt$
div	0x1A	$HI = Rs \% Rt, LO = Rs / Rt$
add	0x20	$Rd = Rs + Rt$

addu	0x21	$Rd = Rs + Rt$ // unsigned integer
sub	0x22	$Rd = Rs - Rt$
subu	0x23	$Rd = Rs - Rt$ // unsigned integer
and	0x24	$Rd = Rs \& Rt$
or	0x25	$Rd = Rs Rt$
xor	0x26	$Rd = Rs \oplus Rt$
nor	0x27	$Rd = \sim(Rs Rt)$
slt	0x2A	$Rd = (Rs < Rt) ? 1 : 0$
sltu	0x2B	$Rd = (Rs < Rt) ? 1 : 0$
break	0x0D	
setie	0x1F	

2. I-Type Instructions

Instruction	Opcode	Description
beq	0x04	if (Rs==Rt) PC=PC+4+BranchAddr
bne	0x05	if (Rs!=Rt) PC=PC+4+BranchAddr
blez	0x06	PC = (Rs < R0)? Label:PC;
bgtz	0x07	PC = (Rs > R0)? Label:PC;
addi	0x08	Rt = rs + 16-bit immediate
sw	0x2B	Mem[Rs+signExtImm] = Rt
slti	0x0A	Rd = (Rs < SignExtImm) ? 1: 0
sltiu	0x0B	Rt = (Rs < SignExtImm) ? 1: 0
andi	0x0C	Rt = Rs + SignExtImm
ori	0x0D	Rt = Rs SignExtImm
xori	0x0E	Rt = Rs ^ SignExtImm
lui	0x0F	Rt = {16'b Imm, 16'b0}
lw	0x23	Rt = Mem[Rs+signExtImm]

3. J-Type Instructions

Instruction	Function code	Description
j	0x02	PC = Address
jal	0x03	{R[31],PC} = {PC+4, Address}

4. Enhanced Instructions

Instruction	Function code	Description
input	0x1C	Rt = ioMem[Rs+signExtImm]
output	0x1D	ioMem[Rs+signExtImm]= Rt
reti	0x1E	PC = dMEM[\$sp]//return addr FLAGS=dMEM[\$sp-4]//flags reg IE = 1'b1
"e_key"	0x1F	UNUSED

Shift Left Logical

R-Type

```
SLL $rd = $rt<<shamt
```

Shift Left Logical will take the contents in the \$rt address and shift the bits left according to the amount specified by shamt.

Shamt stands for Shift Amount and will shift up to 31 bits total.

The function field for a shift will be 6'bits 0.

Carry flag gets the shifted out bit, Overflow is 'Don't Care', Negative and Zero will depend on the result.

```
SLL $R3, $R4, 4
```

```
$r4 = 0x0000_000F
$r3 = 0x0000_0000
$r3 = $r4 << 4
$r3 = 0x0000_00F0
```

```
Flags:  C_V_N_Z
Result: 0_X_0_0
```

Field	Size
OP	6'bits
\$rs	5'bits
\$rt	5'bits
\$rd	5'bits
shamt	5'bits
funct	6'bits

00_0000	S_Addr	T_Addr	D_Addr	0_0100	00_0000
OP	\$rs	\$rt	\$rd	Shamt	Funct

Shift Right Logical R-Type

```
SRL $rd = $rt>>shamt
```

Shift Right Logical will take the contents in the \$rt address and shift the bits right according to the amount specified by shamt.

Shamt stands for Shift Amount and will shift up to 31 bits total.

The function field for a shift will be 6'bits 0x02.

Carry flag gets the shifted out bit, Overflow is 'Don't Care', Negative and Zero will depend on the result.

```
SRL $R3, $R4, 4
```

```
$r4 = 0x0000_00F0
$r3 = 0x0000_0000
$r3 = $r4 >> 4
$r3 = 0x0000_000F
```

```
Flags:  C_V_N_Z
Result: 0_X_0_0
```

Field	Size
OP	6'bits
\$rs	5'bits
\$rt	5'bits
\$rd	5'bits
shamt	5'bits
funct	6'bits

00_0000	S_Addr	T_Addr	D_Addr	0_0100	00_0010
OP	\$rs	\$rt	\$rd	Shamt	Funct

Shift Right Arithmetic R-Type

SRA \$rd = \$rt>>shamt

Shift Right Arithmetic will take the contents in the \$rt address and shift the bits right according to the amount specified by shamt, and will fill in the MSB with the same sign. Shamt stands for Shift Amount and will shift up to 31 bits total.

The function field for a shift will be 6'bits 0x03.

Carry flag gets the shifted out bit, Overflow is 'Don't Care', Negative and Zero will depend on the result.

SRA \$R3, \$R4, 4

\$r4 = 0x0000_000F
\$r3 = 0x0000_0000
\$r3 = \$r4 >> 4
\$r3 = 0x0000_00F0

Flags: C_V_N_Z
Result: 0_X_0_0

Field	Size
OP	6'bits
\$rs	5'bits
\$rt	5'bits
\$rd	5'bits
shamt	5'bits
funct	6'bits

00_0000	S_Addr	T_Addr	D_Addr	0_0100	00_0011
OP	\$rs	\$rt	\$rd	Shamt	Funct

Jump Register

R-Type

JR PC = \$rs

Jump Register will load the Program Counter (PC) with the contents at the register location at S_Address.

Since the Memory Architecture is Byte Addressable, make sure that any JR instructions will have an appropriate value that ends in 2'bits 0. Ex: 0x00, 0x04, 0x08, 0x0C, 0x10

Carry and Overflow are 'Don't Care', Negative and Zero will depend on the result.

JR \$R3

```
$r3 = 0x0000_001C
PC  = $r3
PC  = 0x0000_001C
Flags: C_V_N_Z
Result: X_X_0_0
```

Field	Size
OP	6'bits
\$rs	5'bits
\$rt	5'bits
\$rd	5'bits
shamt	5'bits
funct	6'bits

00_0000	S_Addr	T_Addr	D_Addr	0_0000	00_1000
OP	\$rs	\$rt	\$rd	Shamt	Funct

Move From High

R-Type

MFHI \$rd = Hi

Move From High will be used to take the contents from the HI register and load them into the register specified by the D_Address.

The HI register contains the upper 32 bits from the 64 bit ALU output, typically used to access the remainder after DIV or the upper half of MULT.

All Flags would be 'Don't Care' in this condition because the ALU will have nothing passing through.

MFHI \$r3

\$HI = 0x0000_FFFF
 \$r3 = 0x0000_0000
 \$r3 = \$HI
 \$r3 = 0x0000_FFFF
 Flags: C_V_N_Z
 Result: X_X_X_X

Field	Size
OP	6'bits
\$rs	5'bits
\$rt	5'bits
\$rd	5'bits
shamt	5'bits
funct	6'bits

00_0000	S_Addr	T_Addr	D_Addr	0_0000	01_0000
OP	\$rs	\$rt	\$rd	Shamt	Funct

Move From Low

R-Type

MFLO \$rd = LO

Move From Low will be used to take the contents from the LO register and load them into the register specified by the D_Address.

The LO register contains the lower 32 bits from the 64 bit ALU output, typically used to access the result after DIV or the lower half of MULT.

All Flags would be 'Don't Care' in this condition because the ALU will have nothing passing through.

MFLO \$r3

\$LO = 0x0000_FFFF
 \$r3 = 0x0000_0000
 \$r3 = \$LO
 \$r3 = 0x0000_FFFF
 Flags: C_V_N_Z
 Result: X_X_X_X

Field	Size
OP	6'bits
\$rs	5'bits
\$rt	5'bits
\$rd	5'bits
shamt	5'bits
funct	6'bits

00_0000	S_Addr	T_Addr	D_Addr	0_0000	01_0010
OP	\$rs	\$rt	\$rd	Shamt	Funct

Multiplication

R-Type

`MULT {$HI, $LO} = $rs*$rt`

Multiply will take two 32'bit registers specified by the S_Address and T_Address respectively, and store the 64 bit result of the multiplication into 2 separate loadable registers.

The upper 32'bits [63:32] will be stored inside the HI reg, and the lower 32'bits [31:0] will be stored inside the LO reg. Both of these registers require a MULT or DIV to change their contents.

Carry and Overflow flags would be 'Don't Care' in this condition. The Negative and Zero Flags will be determined by the result of the operation.

`MULT $r4, $r5`

```
$HI = 0x0000_0000
$LO = 0x0000_0000
$r4 = 0x0000_1234
$r5 = 0x5678_0000
{$HI, $LO} = $r4 * $r5
$HI = 0x0000_0626
$LO = 0x0060_0000
Flags:  C_V_N_Z
Result: X_X_0_0
```

Field	Size
OP	6'bits
\$rs	5'bits
\$rt	5'bits
\$rd	5'bits
shamt	5'bits
funct	6'bits

00_0000	S_Addr	T_Addr	D_Addr	0_0000	01_1000
OP	\$rs	\$rt	\$rd	Shamt	Funct

Division

R-Type

```
DIV $HI = $rs % $rt
    $LO = $rs / $rt
```

Divide will take two 32'bit registers specified by the S_Address and T_Address respectively, and store the 32 bit result of the modulus and the division into 2 separate loadable registers.

The remainder 32'bits will be stored inside the HI reg, and the quotient 32'bits will be stored inside the LO reg. Both of these registers require a MULT or DIV to change their contents.

Carry and Overflow flags would be 'Don't Care' in this condition. The Negative and Zero Flags will be determined by the result of the operation.

```
DIV $r5, $r4
```

```
$HI = 0x0000_0000
$LO = 0x0000_0000
$r4 = 0x0000_1234
$r5 = 0x5678_0000
$HI = $r5 % $r4
$LO = $r5 / $r4
$HI = 0x0000_0128
$LO = 0x0004_C00E
```

```
Flags: C_V_N_Z
Result: X_X_0_0
```

Field	Size
OP	6'bits
\$rs	5'bits
\$rt	5'bits
\$rd	5'bits
shamt	5'bits
funct	6'bits

00_0000	S_Addr	T_Addr	D_Addr	0_0000	01_1010
OP	\$rs	\$rt	\$rd	Shamt	Funct

Addition Signed

R-Type

ADD \$rd = \$rs + \$rt

Add is the Signed Addition of the two registers specified by the S_Address and T_Address. The result of the operation will be stored in the destination register (D_Address).

As there is no copy, in this ISA, a good substitute would be adding any register with \$r0 for a pseudo copy.

Carry flag gets 33rd bit as a result of a carry out. Overflow, Negative and Zero will depend on the result.

ADD \$R3, \$R4, \$R5

\$r3 = 0x0000_0000
\$r4 = 0x0000_0003
\$r5 = 0x0000_0032
\$r3 = \$r4 + \$r5
\$r3 = 0x0000_0035
Flags: C_V_N_Z
Result: 0_0_0_0

Field	Size
OP	6'bits
\$rs	5'bits
\$rt	5'bits
\$rd	5'bits
shamt	5'bits
funct	6'bits

00_0000	S_Addr	T_Addr	D_Addr	0_0000	10_0000
OP	\$rs	\$rt	\$rd	Shamt	Funct

Addition Unsigned

R-Type

```
ADDU $rd = $rs + $rt
```

AddU is the Unsigned Addition of the two registers specified by the S_Address and T_Address. The result of the operation will be stored in the destination register (D_Address).

As there is no copy, in this ISA, a good substitute would be adding any register with \$r0 for a pseudo copy.

Carry flag gets 33rd bit as a result of a carry out. Overflow, Negative and Zero will depend on the result.

```
ADDU $R3, $R4, $R5
```

```
$r3 = 0x0000_0000
$r4 = 0x0000_0003
$r5 = 0x0000_0032
$r3 = $r4 + $r5
$r3 = 0x0000_0035
Flags:  C_V_N_Z
Result: 0_0_0_0
```

Field	Size
OP	6'bits
\$rs	5'bits
\$rt	5'bits
\$rd	5'bits
shamt	5'bits
funct	6'bits

00_0000	S_Addr	T_Addr	D_Addr	0_0000	10_0001
OP	\$rs	\$rt	\$rd	Shamt	Funct

Subtraction Signed

R-Type

SUB \$rd = \$rs - \$rt

Sub is the Signed Subtraction of the two registers specified by the S_Address and T_Address. The result of the operation will be stored in the destination register (D_Address).

Carry flag will be the result of the two numbers needing a carry to finish the subtraction in the MSB. Overflow, Negative and Zero will depend on the result.

SUB \$R3, \$R5, \$R4

\$r3 = 0x0000_0000
\$r4 = 0x0000_0003
\$r5 = 0x0000_0032
\$r3 = \$r5 - \$r4
\$r3 = 0x0000_002F

Flags: C_V_N_Z
Result: 0_0_0_0

Field	Size
OP	6'bits
\$rs	5'bits
\$rt	5'bits
\$rd	5'bits
shamt	5'bits
funct	6'bits

00_0000	S_Addr	T_Addr	D_Addr	0_0000	10_0010
OP	\$rs	\$rt	\$rd	Shamt	Funct

Subtraction Unsigned R-Type

`SUBU $rd = $rs - $rt`

SubU is the Unsigned Subtraction of the two registers specified by the S_Address and T_Address. The result of the operation will be stored in the destination register (D_Address).

Carry and Overflow flag will be the result of the two numbers needing a carry to finish the subtraction in the MSB. Negative and Zero will depend on the result.

`SUBU $R3, $R5, $R4`

`$r3 = 0x0000_0000`
`$r4 = 0x0000_0003`
`$r5 = 0x0000_0032`
`$r3 = $r5 - $r4`
`$r3 = 0x0000_002F`

Flags: C_V_N_Z
 Result: 0_0_0_0

Field	Size
OP	6'bits
\$rs	5'bits
\$rt	5'bits
\$rd	5'bits
shamt	5'bits
funct	6'bits

00_0000	S_Addr	T_Addr	D_Addr	0_0000	10_0011
OP	\$rs	\$rt	\$rd	Shamt	Funct

AND

R-Type

```
AND $rd = $rs & $rt
```

AND is a logical type instruction that performs a bit-wise AND between each bit in the two registers specified by the S_Address and T_Address.

The result of the operation is stored in the write back destination specified by the D_Address in the \$rd data field. The Function Select for the AND is 0x24.

The Carry and Overflow Flags are 'Don't Care'. While the Negative and Zero Flag depend on the result.

```
AND $R3, $R4, $R5
```

```
$r3 = 0x0000_0000
$r4 = 0xFFFF_0010
$r5 = 0xF0F0_0030
$r3 = $r4 & $r5
$r3 = 0xF0F0_0010
Flags:  C_V_N_Z
Result: X_X_1_0
```

Field	Size
OP	6'bits
\$rs	5'bits
\$rt	5'bits
\$rd	5'bits
shamt	5'bits
funct	6'bits

00_0000	S_Addr	T_Addr	D_Addr	0_0000	10_0100
OP	\$rs	\$rt	\$rd	Shamt	Funct

OR

R-Type

OR \$rd = \$rs | \$rt

OR is a logical type instruction that performs a bit-wise OR between each bit in the two registers specified by the S_Address and T_Address.

The result of the operation is stored in the write back destination specified by the D_Address in the \$rd data field. The Function Select for the OR is 0x25.

The Carry and Overflow Flags are 'Don't Care'. While the Negative and Zero Flag depend on the result.

OR \$R3, \$R4, \$R5

```
$r3 = 0x0000_0000
$r4 = 0xFFFF_0017
$r5 = 0xF0F0_2040
$r3 = $r4 | $r5
$r3 = 0xFFFF_2057
Flags:  C_V_N_Z
Result: X_X_1_0
```

Field	Size
OP	6'bits
\$rs	5'bits
\$rt	5'bits
\$rd	5'bits
shamt	5'bits
funct	6'bits

00_0000	S_Addr	T_Addr	D_Addr	0_0000	10_0101
OP	\$rs	\$rt	\$rd	Shamt	Funct

Exclusive OR

R-Type

```
XOR $rd = $rs ^ $rt
```

XOR is a logical type instruction that performs a bit-wise Exclusive OR between each bit in the two registers specified by the S_Address and T_Address.

The result of the operation is stored in the write back destination specified by the D_Address in the \$rd data field. The Function Select for the XOR is 0x26.

The Carry and Overflow Flags are 'Don't Care'. While the Negative and Zero Flag depend on the result.

```
XOR $R3, $R4, $R5
```

```
$r3 = 0x0000_0000
$r4 = 0xFFFF_0010
$r5 = 0xF0F0_0030
$r3 = $r4 ^ $r5
$r3 = 0x0F0F_0020
Flags:  C_V_N_Z
Result: X_X_0_0
```

Field	Size
OP	6'bits
\$rs	5'bits
\$rt	5'bits
\$rd	5'bits
shamt	5'bits
funct	6'bits

00_0000	S_Addr	T_Addr	D_Addr	0_0000	10_0110
OP	\$rs	\$rt	\$rd	Shamt	Funct

Not OR

R-Type

`NOR $rd = ~($rs|$rt)`

NOR is a logical type instruction that performs a bit-wise OR with a NOT on the results between each bit in the two registers specified by the S_Address / T_Address.

The result of the operation is stored in the write back destination specified by the D_Address in the \$rd data field. The Function Select for the NOR is 0x27.

The Carry and Overflow Flags are 'Don't Care'. While the Negative and Zero Flag depend on the result.

`NOR $R3, $R4, $R5`

```
$r3 = 0x0000_0000
$r4 = 0xF0F0_1F07
$r5 = 0xFF05_1010
$r3 = ~($r4 | $r5)
$r3 = 0x000A_E0E8
Flags:  C_V_N_Z
Result: X_X_0_0
```

Field	Size
OP	6'bits
\$rs	5'bits
\$rt	5'bits
\$rd	5'bits
shamt	5'bits
funct	6'bits

00_0000	S_Addr	T_Addr	D_Addr	0_0000	10_0111
OP	\$rs	\$rt	\$rd	Shamt	Funct

Set Less Than (Signed) R-Type

```
SLT $rd = ($rs<$rt) ? 1 : 0
```

Set Less Than (Signed) is a logical type operation that will compare to registers specified by the S_ and T_ Addresses in the \$rs/\$rt data fields, respectively.

If the \$rs value is less than \$rt, the destination register (specified by the D_Address in the \$rd field) will be set to 0x0000_0001 but if the comparison fails, then the destination will be loaded with 0x0000_0000 and the zero flag will be set high.

The Carry and Overflow Flags are 'Don't Care'. While the Zero Flag depend on the result.

The Negative flag will always be a low (0) in this operation as each result yields a positive number.

```
SLT $R3, $R4, $R5
```

```
$r3 = 0x0000_0000
$r4 = 0x8001_2345
$r5 = 0x0012_3456
$r3 = ($r4<$r5) ? 1 : 0
```

```
$r3 = 0x0000_0001
```

```
Flags: C_V_N_Z
```

```
Result: X_X_0_0
```

Field	Size
OP	6'bits
\$rs	5'bits
\$rt	5'bits
\$rd	5'bits
shamt	5'bits
funct	6'bits

00_0000	S_Addr	T_Addr	D_Addr	0_0000	10_1010
OP	\$rs	\$rt	\$rd	Shamt	Funct

Set Less Than Unsigned R-Type

```
SLTU $rd = ($rs < $rt) ? 1 : 0
```

Set Less Than Unsigned is a logical type operation that will compare to registers specified by the S_ and T_ Addresses in the \$rs/\$rt data fields, respectively.

If the \$rs value is less than \$rt, the destination register (specified by the D_Address in the \$rd field) will be set to 0x0000_0001 but if the comparison fails, then the destination will be loaded with 0x0000_0000 and the zero flag will be set high.

The Carry and Overflow Flags are 'Don't Care'. While the Zero Flag depend on the result.

The Negative flag will always be a low (0) in this operation as each result yields a positive number.

```
SLTU $R3, $R4, $R5
```

```
$r3 = 0x0000_0000
$r4 = 0x0001_2345
$r5 = 0x0012_3456
$r3 = ($r4 < $r5) ? 1 : 0
```

```
$r3 = 0x0000_0001
```

```
Flags: C_V_N_Z
```

```
Result: X_X_0_0
```

Field	Size
OP	6'bits
\$rs	5'bits
\$rt	5'bits
\$rd	5'bits
shamt	5'bits
funct	6'bits

00_0000	S_Addr	T_Addr	D_Addr	0_0000	10_1011
OP	\$rs	\$rt	\$rd	Shamt	Funct

Break

R-Type

Finishes Simulation

For the purposes of simulation, the BREAK op is used as a way to reach a Halt point. The Halt was used to provide proof that our Branches, Jumps, and ISR all worked properly.

Each module for 1-12 will display just the Data Memory (dMEM) contents for lines 0x0C0 up to 0x0FF.

In modules 13 and 14, the display contents will include the Input/Output Memory (ioMEM) contents for lines 0x0C0 to 0x0FF. In addition, the dMEM will display the contents at address 0x3F0.

All 32 CPU registers will have their contents dumped, using two columns to present the information. The format can be further seen by looking at the ISIM .log files included later.

Break

Displays CPU Register contents into 2 columns:

R0 = 0x____ R16 = 0x____
R1 = 0x____ R17 = 0x____

Displays dMEM contents at lines: 0x0C0 to 0x0FF (1-12)
dMEM[0x3F0] (13/14)

Displays IOMEM contents at lines: 0x0C0 to 0x0FF(13/14)

Field	Size
OP	6'bits
\$rs	5'bits
\$rt	5'bits
\$rd	5'bits
shamt	5'bits
funct	6'bits

00_0000	S_Addr	T_Addr	D_Addr	0_0000	00_1101
OP	\$rs	\$rt	\$rd	Shamt	Funct

Set Interrupt Enable R-Type

SETIE IE = 1'bit 1

Set Interrupt Enable (to HIGH) will cause the processor to prime itself to accept External interrupt signals.

Before the activation of SETIE, the processor will follow any input instructions without reacting too, and handling an interrupt. By running a SETIE instruction, the Interrupt Enable signal (IE) will be set to a HIGH state.

The Interrupt Service Routine (ISR) of the processor uses the IE in order to make sure a currently active ISR will not be interrupted by a new External interrupt signal. To leave the ISR and re-enable the IE, make sure to use a RETI instruction. Details on this instruction can be found in the Enhanced Instructions section.

More information on the ISR can be found in the ISR section, which can be located via the Table of Contents at the beginning of this document.

When using this processor, do not use a SETIE while inside the ISR unless you wish for that Routine to have the potential to be interrupted.

SETIE

IE = 1'bit 0

//Setie instruction used
SETIE

IE = 1'bit 1

Field	Size
OP	6'bits
\$rs	5'bits
\$rt	5'bits
\$rd	5'bits
shamt	5'bits
funct	6'bits

00_0000	S_Addr	T_Addr	D_Addr	0_0000	01_1111
OP	\$rs	\$rt	\$rd	Shamt	Funct

Branch if Equal

I-Type

```
BEQ if($rs==$rt)
PC = PC+4+(Label Offset)
```

Branch IF Equal is an instruction that will fork the code to 1 of 2 possible paths. A pass will cause the Program Counter (PC) to be loaded with the address in Instruction Memory (IR_MEM) of the label specified in the assembly code.

The BEQ works by taking two register values specified by the S_ and T_ Addresses with their respective \$rs and \$rt fields (as shown below), and performs a signed subtraction. The Zero flag is checked to determine whether or not the branch passes.

A HIGH Zero Flag will be the result of two equal register values, and the PC will branch to the new subroutine in the assembly.

A LOW Zero Flag will be the result of two different register values, and the PC will proceed to the next instruction in the IR_MEM.

The rest of the Flags will contain values dependent on the result of the Subtraction, but are not used in this instruction.

```
BEQ $r3, $r4, Label
```

```
$r3 = 0x0000_0050
$r4 = 0x0000_0050
PC = 0x0000_002C
Label offset = 0x0040
SE_16=0x0000_0040
Z_Flag = ($r3-$r4) ? 0 : 1

if(Z_Flag)
    PC = PC+4 + SE_16
else
    PC = PC+4
PC = 0x0000_0070 //beq Passed
```

Field	Size
OP	6'bits
\$rs	5'bits
\$rt	5'bits
Immediate Value	16'bits
SE_16	32'bits
Label Offset	16'bits

00_0100	S_Addr	T_Addr	Branch Label Offset
OP	\$rs	\$rt	16'bit Immediate Value

Branch if NOT Equal I-Type

```
BNE if($rs!=$rt)
PC = PC+4+(Label Offset)
```

Branch if NOT Equal is an instruction that will fork the code to 1 of 2 possible paths. A pass will cause the Program Counter (PC) to be loaded with the address in Instruction Memory (IR_MEM) of the label specified in the assembly code.

The BNE works by taking two register values specified by the S_ and T_ Addresses with their respective \$rs and \$rt fields (as shown below), and performs a signed subtraction. The Zero flag is checked to determine whether or not the branch passes.

A HIGH Zero Flag will be the result of two equal register values, and the PC will proceed to the next instruction in the IR_MEM.

A LOW Zero Flag will be the result of two different register values, and the PC will branch to the new subroutine in the assembly.

The rest of the Flags will contain values dependent on the result of the Subtraction, but are not used in this instruction.

```
BNE $r3, $r4, Label
```

```
$r3 = 0x0000_0050
$r4 = 0x0000_FFFF
PC = 0x0000_002C
Label offset = 0x0040
SE_16=0x0000_0040
Z_Flag = ($r3-$r4) ? 0 : 1

if(!Z_Flag)
    PC = PC+4 + SE_16
else
    PC = PC+4
PC = 0x0000_0070 //bne Passed
```

Field	Size
OP	6'bits
\$rs	5'bits
\$rt	5'bits
Immediate Value	16'bits
SE_16	32'bits
Label Offset	16'bits

00_0101	S_Addr	T_Addr	Branch Label Offset
OP	\$rs	\$rt	16'bit Immediate Value

Branch if Less-Than or Equal to Zero

I-Type

```
BLEZ if($rs<=32'h0)
PC = PC+4+(Label Offset)
```

Branch if Less Than or Equal (BLEZ) is an instruction that will fork the code to 1 of 2 possible paths. A pass will cause the Program Counter (PC) to be loaded with the address in Instruction Memory (IR_MEM) of the label specified in the assembly code.

The BLEZ works by taking the register value from the S_Address, specified by the \$rs field, and the contents from the \$r0 register. The \$rt field in the instruction will be left blank as to access the \$r0 register for the 32'bit 0 value stored there.

Upon signed subtraction:

- HIGH Z or N Flag will result in a pass and the PC will be loaded with the new Address of the subroutine.

- LOW Z & N Flag will result in a fail and the PC will get PC+4 and move to the next line in the IR_MEM.

The rest of the Flags will contain values dependent on the result of the Subtraction, but are not used in this instruction.

```
BLEZ $r3, Label
```

```
$r3 = 0x8000_0050
$r0 = 0x0000_0000
PC = 0x0000_002C
Label offset = 0x0040
SE_16=0x0000_0040
{Flags} = SUB $r3 - $r0
```

```
if(Z_Flag | N_Flag)
    PC = PC+4 + SE_16
else
    PC = PC+4
PC = 0x0000_0070 //blez Passed
```

Field	Size
OP	6'bits
\$rs	5'bits
\$rt	5'bits
Immediate Value	16'bits
SE_16	32'bits
Label Offset	16'bits

00_0110	S_Addr	0_0000	Branch Label Offset
OP	\$rs	\$rt	16'bit Immediate Value

Branch if Greater than Zero I-Type

BGTZ if(\$rs > 32'h0)
PC = PC+4+(Label Offset)

Branch if Greater Than Equal (BGTZ) is an instruction that will fork the code to 1 of 2 possible paths. A pass will cause the Program Counter (PC) to be loaded with the address in Instruction Memory (IR_MEM) of the label specified in the assembly code.

The BLEZ works by taking the register value from the S_Address, specified by the \$rs field, and the contents from the \$r0 register. The \$rt field in the instruction will be left blank as to access the \$r0 register for the 32'bit 0 value stored there.

Upon signed subtraction:

- LOW Z & N Flag will result in a pass and the PC will be loaded with the new Address of the subroutine.

- HIGH Z or N Flag will result in a fail and the PC will get PC+4 and move to the next line in the IR_MEM.

The rest of the Flags will contain values dependent on the result of the Subtraction, but are not used in this instruction.

BGTZ \$r3, Label

```
$r3 = 0x0000_2F50
$r0 = 0x0000_0000
PC = 0x0000_002C
Label offset = 0x0040
SE_16=0x0000_0040
{Flags} = SUB $r3 - $r0
```

```
if(!Z_Flag & !N_Flag)
    PC = PC+4 + SE_16
else
    PC = PC+4
PC = 0x0000_0070 //bgtz Passed
```

Field	Size
OP	6'bits
\$rs	5'bits
\$rt	5'bits
Immediate Value	16'bits
SE_16	32'bits
Label Offset	16'bits

00_0111	S_Addr	0_0000	Branch Label Offset
OP	\$rs	\$rt	16'bit Immediate Value

Add Immediate

I-Type

```
ADDI $rt = $rs + 16'bit Imm.
```

Add Immediate is the addition of a register value, specified by the S_Address and the \$rs field below, and the 16'bit Immediate Value provided by the assembly instruction.

The 16 bit Immediate will be Sign Extended to a full 32'bit Value, and added with the register. The result is stored in the register destination specified by the T_Address and the \$rt field below.

All flags will be affected, with the Carry being the 33rd bit of the addition. The Overflow, Negative, and Zero flags all depend on the result.

```
ADDI $r3, $r4, 16'imm
```

```
16'Imm = 0x_1234  
$r3 = 0x0000_0000  
$r4 = 0x0000_2000  
SE16= 0x0000_1234
```

```
$r3 = $r4 + SE16
```

```
$r3 = 0x0000_3234
```

```
Flags: C_V_N_Z  
Result: 0_0_0_0
```

Field	Size
OP	6'bits
\$rs	5'bits
\$rt	5'bits
Immediate Value	16'bits
SE_16	32'bits
Label Offset	16'bits

00_1000	S_Addr	T_Addr	Immediate Value
OP	\$rs	\$rt	16'bit Immediate Value

Store Word

I-Type

```
SW dMEM[SE_16 + $rs] = $rt
```

Store Word (SW) is the only way to write to Data Memory (dMEM) in this Architecture. The instruction relies on a register value, specified by S_Address and the \$rs field, to act as the address to the desired dMEM location.

The 16'bit Immediate Value is used for the Effective Address (EA) Calculation and can be used as a 0 value to directly access the Registers location in dMEM, or can be used as an offset. The offset style of EA allows the register to act as a pointer, commonly used in higher level language like C or C++.

The EA is done by using Signed Addition to create the address for memory. The register contents specified by the T_Address and \$rt field will be stored into dMEM[EA].

```
SW $r3, 16'imm($r4)
```

```
16'Imm. = 0xFFFC
$r3 = 0x1234_5678
$r4 = 0x0000_03FC
SE16= 0xFFFF_FFFC
```

```
E.A.= 0x0000_03F8
```

```
//Store
dMEM[0x3F8] = 0x1234_5678
```

Field	Size
OP	6'bits
\$rs	5'bits
\$rt	5'bits
Immediate Value	16'bits
SE_16	32'bits
Label Offset	16'bits

10_1011	S_Addr	T_Addr	Memory Address Offset
OP	\$rs	\$rt	16'bit Immediate Value

Set Less Than Immediate

I-Type

```
SLTI if($rs < 16'Imm) $rt=1
      else $rt=0
```

Set Less Than Immediate is the instruction to compare if the contents of a register, specified by the S_Address in the \$rs field below, are Less Than a 16'bit value given by the assembly instruction.

The 16'bit Value will be Sign Extended (SE16) to 32'bits and then the comparison occurs. If the \$rs register value is Less Than the SE16 value, the destination register will be set to a 1. If Greater than or Equal to the SE16 value, the destination will be set to a 0.

The destination register is specified by the T_Address and the \$rt field found below.

The Carry and Overflow flags are 'Don't Care', while the Zero Flag is determined by the result. The Negative Flag will always be a Zero as it can never actually reach a Negative value since the outputs of the ALU are either 0 or 1, both of which are non-negative numbers.

```
SLTI $r3, $r4, 16'imm
```

```
16'Imm = 0x_7FFF
$r3 = 0x0000_0000
$r4 = 0x0000_2468
SE16= 0x0000_7FFF
```

```
if($r4 < SE16)
    $r3 = 0x0000_0001
else
    $r3 = 0x0000_0000
Flags:  C_V_N_Z
Result: X_X_0_0
```

Field	Size
OP	6'bits
\$rs	5'bits
\$rt	5'bits
Immediate Value	16'bits
SE_16	32'bits
Label Offset	16'bits

00_1010	S_Addr	T_Addr	Immediate Value
OP	\$rs	\$rt	16'bit Immediate Value

Set Less Than Immediate Unsigned

I-Type

```
SLTIU if($rs < 16'Imm) $rt=1
      else $rt=0
```

Set Less Than Immediate Unsigned is the instruction to compare if the contents of a register, specified by the S_Address in the \$rs field below, are Less Than a 16'bit value given by the assembly instruction.

The 16'bit Value will be Sign Extended (SE16) to 32'bits and then the comparison occurs. However, even though the Immediate is Sign Extended, the actual comparison is unsigned and treats both numbers as positives. If the \$rs register value is Less Than the SE16 value, the destination register will be set to a 1. If Greater than or Equal to the SE16 value, the destination will be set to a 0.

The destination register is specified by the T_Address and the \$rt field found below.

The Carry and Overflow flags are 'Don't Care', while the Zero Flag is determined by the result. The Negative Flag will always be a Zero as it can never actually reach a Negative value since the outputs of the ALU are either 0 or 1, both of which are non-negative numbers.

```
SLTIU $r3, $r4, 16'imm
```

```
16'Imm = 0x_FFFF
$r3 = 0x0000_0000
$r4 = 0x0000_2468
SE16= 0xFFFF_FFFF
```

```
if($r4 < SE16)
    $r3 = 0x0000_0001
else
    $r3 = 0x0000_0000
Flags:  C_V_N_Z
Result: X_X_0_0
```

Field	Size
OP	6'bits
\$rs	5'bits
\$rt	5'bits
Immediate Value	16'bits
SE_16	32'bits
Label Offset	16'bits

00_1011	S_Addr	T_Addr	Immediate Value
OP	\$rs	\$rt	16'bit Immediate Value

And Immediate

I-Type

```
ANDI $rt = $rs & 16'bit Imm.
```

And Immediate is the instruction for the bit-wise AND of the register contents specified by the S_Address and \$rs field below and the 16'bit Immediate Value from the assembly Instruction.

The 16'bit Immediate Value will be Sign Extended (SE16) and then the results of the bit-wise AND will be stored in the register destination specified by the T_Address in the \$rt field below.

The bit-wise and will and each bit of both 32 bit values and put the result in the same bit position in the destination register.

Both the Carry and Overflow flags will be 'Don't Care' state, and the Negative and Zero flags will both depend on the result of the bitwise AND.

```
ANDI $r3, $r4, 16'imm
```

```
16'Imm = 0x_0FF0  
$r3 = 0x0000_0000  
$r4 = 0xFFFF_1234  
SE16= 0x0000_0FF0
```

```
$r3 = $r4 & SE16
```

```
$r3 = 0x0000_0230
```

```
Flags: C_V_N_Z  
Result: X_X_0_0
```

Field	Size
OP	6'bits
\$rs	5'bits
\$rt	5'bits
Immediate Value	16'bits
SE_16	32'bits
Label Offset	16'bits

00_1100	S_Addr	T_Addr	Immediate Value
OP	\$rs	\$rt	16'bit Immediate Value

OR Immediate

I-Type

```
ORI $rt = $rs | 16'bit Imm.
```

OR Immediate is the instruction for the bit-wise OR of the register contents specified by the S_Address and \$rs field below and the 16'bit Immediate Value from the assembly Instruction.

The 16'bit Immediate Value will be Sign Extended (SE16) and then the results of the bit-wise OR will be stored in the register destination specified by the T_Address in the \$rt field below.

The bit-wise OR will write any HIGH bit of either 32 bit values and put the result in the same bit position in the destination register.

Both the Carry and Overflow flags will be 'Don't Care' state, and the Negative and Zero flags will both depend on the result of the bitwise OR.

```
ORI $r3, $r4, 16'imm
```

```
16'Imm = 0x_0FF0  
$r3 = 0x0000_0000  
$r4 = 0xFFFF_1234  
SE16= 0x0000_0FF0
```

```
$r3 = $r4 | SE16
```

```
$r3 = 0xFFFF_1FF4
```

```
Flags: C_V_N_Z  
Result: X_X_0_0
```

Field	Size
OP	6'bits
\$rs	5'bits
\$rt	5'bits
Immediate Value	16'bits
SE_16	32'bits
Label Offset	16'bits

00_1101	S_Addr	T_Addr	Immediate Value
OP	\$rs	\$rt	16'bit Immediate Value

XOR Immediate

I-Type

```
XORI $rt = $rs | 16'bit Imm.
```

XOR Immediate is the instruction for the bit-wise Exclusive OR of the register contents specified by the S_Address and \$rs field below and the 16'bit Immediate Value from the assembly Instruction.

The 16'bit Immediate Value will be Sign Extended (SE16) and then the results of the bit-wise XOR will be stored in the register destination specified by the T_Address in the \$rt field below.

The bit-wise XOR will only write a HIGH bit to the destination register when there is a HIGH bit in either the \$rs or SE16, but not the other. This gives it the exclusive property that defines XOR.

Both the Carry and Overflow flags will be 'Don't Care' state, and the Negative and Zero flags will both depend on the result of the bitwise XOR.

```
XORI $r3, $r4, 16'imm
```

```
16'Imm = 0x_0FF0  
$r3 = 0x0000_0000  
$r4 = 0xFFFF_1234  
SE16= 0x0000_0FF0
```

```
$r3 = $r4 ^ SE16
```

```
$r3 = 0xFFFF_1DC4
```

```
Flags: C_V_N_Z  
Result: X_X_0_0
```

Field	Size
OP	6'bits
\$rs	5'bits
\$rt	5'bits
Immediate Value	16'bits
SE_16	32'bits
Label Offset	16'bits

00_1101	S_Addr	T_Addr	Immediate Value
OP	\$rs	\$rt	16'bit Immediate Value

Load Upper Immediate I-Type

```
LUI $rt = {16'bit Imm, 16'b0}
```

Load Upper Immediate (LUI) is the instruction most commonly used in combination with ORI to load a CPU register with a full 32'bit immediate value.

The functionality of a LUI instruction is such that it will clear the destination register, specified by the T_Address in the \$rt field below, and then load the upper 16 bits of that register.

The lower 16 bits are filled with 0's by the ALU and the upper 16 uses the Immediate value in the assembly instruction.

This sets up a register that can be easily used with ORI to provide another 16'bits that will bit-wise OR with 0s copying the other Immediate value, and giving a full 32 bit value.

As seen in the example, the current values in the \$rt register are cleared out. This prevents LUI from modifying the upper bits and leaving the lower bits unchanged.

The Carry and Overflow flags are 'Don't Care' state, while the Negative and Zero Flags depend on the result.

```
LUI $r3, 16'imm
```

```
16'Imm = 0x_8642
```

```
$r3 = 0x1234_5678
```

```
$r3 = {16'Imm, 0x0000}
```

```
$r3 = 0x8642_0000
```

```
Flags: C_V_N_Z
```

```
Result: X_X_0_0
```

Field	Size
OP	6'bits
\$rs	5'bits
\$rt	5'bits
Immediate Value	16'bits
SE_16	32'bits
Label Offset	16'bits

00_1111	0_0000	T_Addr	Immediate Value
OP	\$rs	\$rt	16'bit Immediate Value

Load Word

I-Type

```
LW $rt = dMEM[SE_16 + $rs]
```

Load Word (SW) is the only way to Read from Data Memory (dMEM) in this Architecture. The instruction relies on a register value, specified by S_Address and the \$rs field, to act as the address to the desired dMEM location.

The 16'bit Immediate Value is used for the Effective Address (EA) Calculation and can be used as a 0 value to directly read the location in dMEM pointed to by the register, or can be used as an offset. The offset style of EA allows the register to act as a pointer, commonly used in higher level language like C or C++.

The EA is done by using Signed Addition to create the address for memory. The register specified by the T_Address and \$rt field will be loaded with the contents of dMEM[EA].

```
LW $r3, 16'imm($r4)
```

```
16'Imm. = 0xFFFC
$r3 = 0xA5A5_A5A5
$r4 = 0x0000_03FC
SE16= 0xFFFF_FFFC
E.A.= 0x0000_03F8
```

```
//Memory Contains
dMEM[0x3F8] = 0x1234_5678
```

```
//Load to RegFile
$r3 = dMEM[0x3F8]
$r3 = 0x1234_5678
```

Field	Size
OP	6'bits
\$rs	5'bits
\$rt	5'bits
Immediate Value	16'bits
SE_16	32'bits
Label Offset	16'bits

10_0011	S_Addr	T_Addr	Memory Address Offset
OP	\$rs	\$rt	16'bit Immediate Value

Jump

J-Type

J PC = Jump Address

JUMP (J) is an unconditional change of the Program Counter (PC) and its contents. The JUMP instruction is designed to load the PC with the address of the subroutine specified by the Jump_Label.

The Jump_Label is an offset that will be used to along with the current PC contents to calculate the Jump Address. The Jump value is handled by the assembler and the processor to make sure the location of the Label is byte addressable.

The Jump can be used to access up to 64 Mbytes (2^{26}), or 32 Mbytes above and below the current instruction. However, the JUMP instruction doesn't limit what Instructions can be accessed. The IR_MEM in this Processor is only 1064 Kbytes x 32'bits effective address.

J jump_Label

26'Imm. = Jump_Label

PC = 0x0000_0244

PC = {PC[31:29]+ Jump_Label + 2'b00}

PC = Jump Address

Field	Size
OP	6'bits
Jump_Label	26'bit Immediate Value

00_0010	Jump_Label Address
OP	26'bit Immediate Value

Jump and Link

J-Type

```
JAL {R[31],PC}={PC+4,JumpAddress}
```

JAL (J) is an unconditional change of the Program Counter (PC) and its contents. The JAL instruction is designed to load the PC with the address of the subroutine specified by the Jump_Label. In addition, the current address of the Next Instruction in the IMEM will be loaded into the Return Address Register (\$ra).

The Jump_Label is an offset that will be used to along with the current PC contents to calculate the Jump Address. The JAL has similar constraints as the JUMP in terms of instructions it can move to.

The benefit to JAL is the programmer's ability to use \$ra as a way to return to the previous subroutine. A possible use could be to branch around a JAL, but on a failed branch it would execute. The JAL could then access a subroutine to perform a task, and return back to the main routine.

```
JAL jump_Label
```

```
26'Imm. = Jump_Label
```

```
PC = 0x0000_0244
```

```
$r31 = 0x0000_0000
```

```
//Execute JAL
```

```
PC = {PC[31:29]+ Jump_Label  
+ 2'b00}
```

```
PC = Jump Address
```

```
$r31 = 0x0000_0244
```

Field	Size
OP	6'bits
\$rs	5'bits

00_0011	Jump_Label Address
OP	26'bit Immediate Value

Input

Enhanced-Type

```
INPUT $rt = ioMEM[SE_16 + $rs]
```

INPUT is an Enhanced Instruction designed to access the contents of the Input/Output Memory (IO_MEM) contents. Input is similar to the Store Word, and can be visualized as a value being input into the Processor from an External Source.

The External Input value can be found by using the contents in the register specified by the S_Address in the \$rs field below. The 16'bit Immediate value will act as an Offset or as a 0 value to point to the contents in the IO_MEM.

The destination of the INPUT will be specified by the T_Address in the \$rt field and will be loaded with the contents of the IO_MEM at the calculated address.

```
INPUT $r3, 16'imm($r4)
```

```
16'Imm. = 0xFFFC
$r3 = 0xA5A5_A5A5
$r4 = 0x0000_03FC
SE16= 0xFFFF_FFFC
E.A.= 0x0000_03F8
```

```
//IO_Memory Contains
ioMEM[0x3F8] = 0x1234_5678
```

```
//Input to RegFile
$r3 = ioMEM[0x3F8]
$r3 = 0x1234_5678
```

Field	Size
OP	6'bits
\$rs	5'bits
\$rt	5'bits
Immediate Value	16'bits

01_1100	S_Addr	T_Addr	IO_Memory Address Offset
OP	\$rs	\$rt	16'bit Immediate Value

Output

Enhanced-Type

```
OUTPUT ioMEM[SE_16 +$rs] = $rt
```

OUTPUT is an Enhanced Instruction designed to access the contents of the Input/Output Memory (IO_MEM) contents. Output is similar to the Load Word, and can be visualized as a value being Output from the Processor to an External Source.

The Address to Store the CPU register value can be found by using the contents in the register specified by the S_Address in the \$rs field below. The 16'bit Immediate value will act as an Offset or as a 0 value to point to the location in IO_MEM.

The source of the OUTPUT will be specified by the T_Address in the \$rt field and will retain the contents it output to IO_MEM after the instruction executes.

```
OUTPUT $r3, 16'imm($r4)
```

```
16'Imm. = 0xFFFC
```

```
$r3 = 0x1234_5678
```

```
$r4 = 0x0000_03FC
```

```
SE16= 0xFFFF_FFFC
```

```
E.A.= 0x0000_03F8
```

```
//Store to IO_MEM
```

```
ioMEM[0x3F8] = 0x1234_5678
```

Field	Size
OP	6'bits
\$rs	5'bits
\$rt	5'bits
Immediate Value	16'bits

10_1101	S_Addr	T_Addr	IO_Memory Address Offset
OP	\$rs	\$rt	16'bit Immediate Value

Return from ISR, Enable Interrupts R-Type

RETI: pops Flags
pops Addr, IE = 1

RETI is the instruction to use where the Interrupt Sub Routine (ISR) ends. The last instruction in the ISR should be RETI, because if a Jump or another instruction is used to leave the ISR, the Interrupt Enable (IE) will stay low.

The IE is turned on only through SETIE, and turns itself off when inside of the ISR. This precaution is taken to avoid an endless influx of Interrupts that prevent instructions from being executed. The RETI instruction is to let the processor know that the ISR is finished, and that it can accept more External Interrupts to handle.

The Processor will always see the External Interrupts, but only act upon them if the IE is also HIGH.

When Returning from the ISR back to the previous instruction/routine, the Stack pointer (\$sp) will pop the previous instruction address and load the Program Counter (PC). The \$sp is automatically decremented by 4, and pops the Flags off the stack and into the Flags register. Finally the \$sp is decremented by 4 one last time, and then points to the ISR address again.

RETI

```
PC = dMEM[$sp] //pop Addr
$sp = $sp -4    //dec4 $sp
FLAGS = dMEM[$sp]//pop flags
$sp = $sp -4    //dec4 $sp
$sp = 0x3FC // $sp points ISR
Interrupt_Enable = 1'bit 1
```

Field	Size
OP	6'bits
empty	26'bits

01_1110	00_0000_0000_0000_0000_0000_0000
OP	26'bits 0

F. Instruction Format

1. R Type

Register Type instructions contain all operands pertaining to the instruction in a register. Typically these instructions formats contain the instruction and three registers, two of which are source registers and one being a destination register where the result of the instruction will be stored. However there are some exceptions to this format, such as shift instructions that only have 2 registers and the shift amount in the instruction.

MIPS ASSEMBLY CODE

Opcode rd,rs,rt

31	26	25	21	20	16	15	11	10	6	5	0
0x0000	rs	rt	rd	Shamt	Function						


When forming this instruction type notice that the designated bits for the opcode are to be set to 0x00. This is because the actual opcode value will be stored in the function section on the right hand side.

Examples.

MIPS ASSEMBLY: ADD \$at,\$v0,\$t0

BINARY VALUES:

31	26	25	21	20	16	15	11	10	6	5	0
0x0000	\$v0	\$t0	\$at	0x000	ADD						
OPCODE	RS	Rt	Rd	Shamt	Function						



31	26	25	21	20	16	15	11	10	6	5	0
000000	00010	01000	00001	00000	000101						
OPCODE	RS	Rt	Rd	Shamt	Function						

MIPS ASSEMBLY: SLL \$at,\$v0,4

BINARY VALUES:

31	26	25	21	20	16	15	11	10	6	5	0
0x0000	\$v0	0x0000	\$at	0x0004	SLL						
OPCODE	RS	Rt	Rd	Shamt	Function						



31	26	25	21	20	16	15	11	10	6	5	0
000000	000010	000000	000001	000100	000000						
OPCODE	RS	Rt	Rd	Shamt	Function						

-

MIPS ASSEMBLY: AND \$t0,\$v0,\$t0

BINARY VALUES:

31	26	25	21	20	16	15	11	10	6	5	0
0x0000	\$v0	\$t0	\$t0	0x000	AND						
OPCODE	RS	Rt	Rd	Shamt	Function						



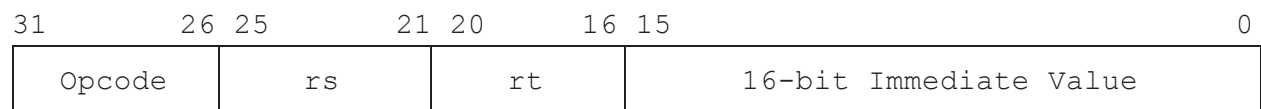
31	26	25	21	20	16	15	11	10	6	5	0
000000	00010	00001	00001	00000	011000						
OPCODE	RS	Rt	Rd	Shamt	Function						

2. I Type

Immediate Type instructions work just as the name implies by having an immediate value with the instruction. Typically this instruction format will still have two registers, but just one is a source and one is a destination. The immediate value is used instead of a second source register. This allows the user to use a value that has not yet been saved into a source register.

Opcode *rt,rs, #0xHHHH*

Note: In Immediate type the destination register is RT.

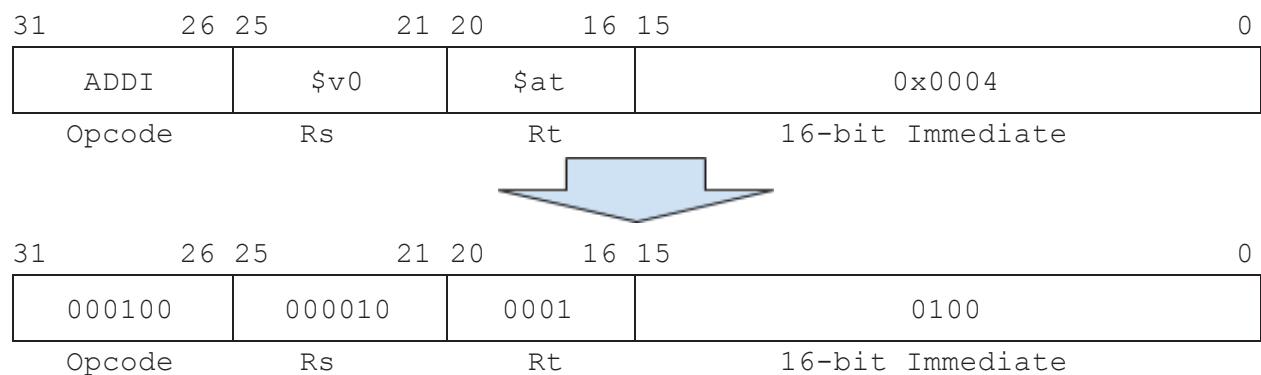


Immediate instruction can serve a variety of purposes, for immediate arithmetic functions it can add an immediate value to a stored value. Other purposes can be to use different addressing modes, such as a branch instruction where the immediate value can be used as an offset to the Program Counter.

Examples.

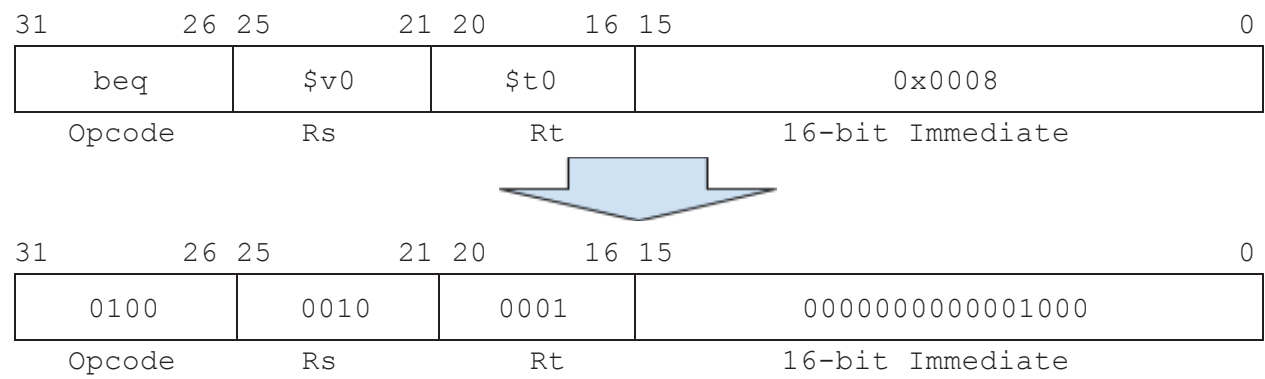
MIPS ASSEMBLY: *ADDI \$at,\$v0,0x0004*

BINARY VALUES:



```
// PC = 0xBE23
```

BINARY VALUES:

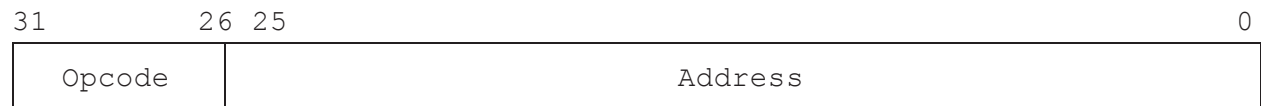


3. J type

Jump Instructions are the least used but they are still powerful in that they allow you to jump to any address location in the memory. Jump instructions only have two fields which are the opcode for jump and the address the program wants to jump to.

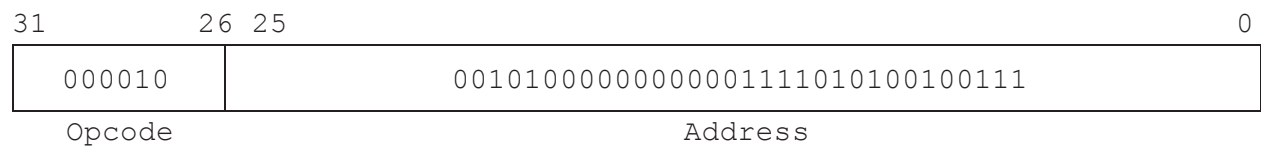
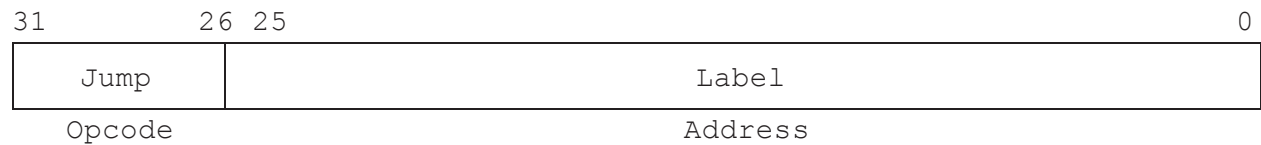
MIPS Assembly:

Opcode Address



Examples.

MIPS Assembly: J Label // Label is a 26-bit address
Binary Values: //for the sake of the example
 //Label = 0x2800F527



III. Verilog Implementation / Design / Verification

A. TOP LEVEL Source Code - 2 Pages

```

1  `timescale 1ns / 1ps
2  /*****
3  * Authors:   Javier Garcia
4  *           Taylor Cochran
5  *
6  * Emails:    javis9526@yahoo.com
7  *           TayCochran123@gmail.com
8  *
9  * Filename:  CPU_tb.v
10 * Date:      November 27, 2018
11 * Version:   1.2
12 *
13 *                                                    (End of Page Width)
14 *****/
15 module CPU_tb;
16
17     // Inputs
18     reg clk;
19     reg rst;
20
21
22     wire      intr,intr_ack;
23     wire [31:0] EXMEM_ALU,pc4_mux,M_Data,io_out;
24     wire [15:0] EXMEM_M;
25     // Instantiate the Unit Under Test (UUT)
26     ///////////////////////////////////////////////////
27     //Central Processing Unit
28     ///////////////////////////////////////////////////
29     CPU uut (
30         .clk(clk),
31         .rst(rst),
32         .intr_ack(intr_ack),
33         .intr(io_intr),
34         .EXMEM_ALU(EXMEM_ALU),
35         .pc4_mux(pc4_mux),
36         .EXMEM_M(EXMEM_M),
37         .M_Data(M_Data),
38         .io_out(io_out)
39     );
40     ///////////////////////////////////////////////////
41     //Data memory module
42     ///////////////////////////////////////////////////
43     DataMemory uut2 (.clk(clk),
44                     .Address(EXMEM_ALU[11:0]),
45                     .D_In(pc4_mux),
46                     .dm_cs(EXMEM_M[6]),
47                     .dm_wr(EXMEM_M[5]),
48                     .dm_rd(EXMEM_M[4]),
49                     .D_Out(M_Data));
50
51     ///////////////////////////////////////////////////
52     //IO memory Module
53     ///////////////////////////////////////////////////
54     IO_Memory uut3 (.clk(clk),
55                    .rst(rst),
56                    .io_cs(EXMEM_M[9]),
57                    .io_wr(EXMEM_M[8]),

```

```
58         .io_rd(EXMEM_M[7]),
59         .Address(EXMEM_ALU[11:0]),
60         .D_In(pc4_mux),
61         .D_Out(io_out),
62         .intr_ack(intr_ack),
63         .io_intr(io_intr));
64
65     always #5 clk = ~clk;
66
67     initial begin
68         // Initialize Inputs
69         clk = 0;
70         rst = 1;
71
72         // Wait 100 ns for global reset to finish
73
74         //$readmemh("iM_01.dat", uut.IU.IM.M);
75         //$readmemh("dM_01.dat", uut2.M);
76         //$readmemh("dM_02.dat", uut2.M);
77         //$readmemh("iM_02.dat", uut.IU.IM.M);
78         //$readmemh("iM_03.dat", uut.IU.IM.M);
79         //$readmemh("dM_03.dat", uut2.M);
80         //$readmemh("iM_04.dat", uut.IU.IM.M);
81         //$readmemh("dM_04.dat", uut2.M);
82         //$readmemh("iM_05.dat", uut.IU.IM.M);
83         //$readmemh("dM_05.dat", uut2.M);
84         //$readmemh("iM_06.dat", uut.IU.IM.M);
85         //$readmemh("dM_06.dat", uut2.M);
86         //$readmemh("iM_07.dat", uut.IU.IM.M);
87         //$readmemh("dM_07.dat", uut2.M);
88         //$readmemh("iM_08.dat", uut.IU.IM.M);
89         //$readmemh("dM_08.dat", uut2.M);
90         //$readmemh("iM_09.dat", uut.IU.IM.M);
91         //$readmemh("dM_09.dat", uut2.M);
92         //$readmemh("iM_10.dat", uut.IU.IM.M);
93         //$readmemh("dM_10.dat", uut2.M);
94         //$readmemh("iM_11.dat", uut.IU.IM.M);
95         //$readmemh("dM_11.dat", uut2.M);
96         //$readmemh("iM_12.dat", uut.IU.IM.M);
97         //$readmemh("dM_12.dat", uut2.M);
98         //$readmemh("iM_13.dat", uut.IU.IM.M);
99         //$readmemh("dM_13.dat", uut2.M);
100        $readmemh("iM_14.dat", uut.IU.IM.M);
101        $readmemh("dM_14.dat", uut2.M);
102        #10
103
104        rst = 0;
105
106    end
107
108 endmodule
109
110
```

B. Modules and Sub Modules - 43 Pages

```

1  `timescale 1ns / 1ps
2  /*****
3  * Authors:   Javier Garcia
4  *           Taylor Cochran
5  *
6  * Emails:    javis9526@yahoo.com
7  *           TayCochran123@gmail.com
8  *
9  * Filename:  CPU.v
10 * Date:      November 26, 2018
11 * Version:   1.2
12 *
13 *                                                    (End of Page Width)
14 *****/
15 module CPU(clk,rst,intr_ack,
16             intr,EXMEM_ALU,
17             pc4_mux,
18             EXMEM_M,
19             M_Data,io_out);
20
21     input      clk, rst;
22     input      intr; // interrupt
23     input [31:0] M_Data,io_out; //memory outputs
24
25     output      intr_ack; //interrupt acknowledge
26     output [31:0] EXMEM_ALU,pc4_mux; //ALU, data to mem
27     output [15:0] EXMEM_M; // Memory Control Word
28
29     wire        Branch_s,ISR,IE; //Branch, Interrupt service, Interrupt Enable
30     wire [31:0] IFID_IR; //instruction register
31
32     wire [2:0] LISR; //leave interrupt service routine
33     wire [15:0] EX; //execute control
34     wire [15:0] M; // memory control
35     wire [3:0] WB; // write back control
36
37
38     wire        pc_ld , pc_inc;
39     wire [31:0] PC_in;
40     wire        im_cs , im_wr , im_rd;
41     wire [31:0] PC ;
42     wire [15:0] IDEX_M;
43     wire [4:0] IDEX_TA;
44     wire [31:0] D_Out;
45     wire [31:0] PC_out,M_Data;
46     wire [15:0] EXMEM_M;
47     wire [31:0] pc4_mux, EXMEM_ALU;
48
49     wire Stall,Flush;
50
51     ///////////////////////////////////////////////////
52     //                                CONTROL UNIT
53     //The control unit handles setting up the control words for every
54     //instruction. It outputs new control words every clock cycle due to
55     //design being a pipeline. The control unit also handles setting up and
56     //leaving the ISR.
57     ///////////////////////////////////////////////////

```

```

58
59     Control_Unit CU(.clk(clk),           //clock
60                     .rst(rst),           //reset
61                     .IE(IE),             //Interrupt Enable
62                     .ISR(ISR),           //Interrupt Service Routine
63                     .LISR(LISR),         //Leave Interrupt Service Routine
64                     .EXMEM_M(EXMEM_M),   //Memory Control Word
65                     .IFID_IR(IFID_IR),   //Instruction Register
66                     .Branch_s(Branch_s), //Branch Signal
67                     .intr(intr),         //Interrupt
68                     .intr_ack(intr_ack), //Interrupt Acknowledg
69                     .EX(EX),             //Execute Control Words
70                     .M(M),               //Memory Control Words
71                     .WB(WB));            //Write Back Control Words
72
73     ///////////////////////////////////////////////////
74     //                      HAZARD CONTROL UNIT
75     //This unit takes care of any hazards in the pipeline based on an
76     //instruction. The main purpose is to flush or stall the pipeline when
77     //necessary.
78     ///////////////////////////////////////////////////
79     Control_Hazard CHU (.IFID_IR(IFID_IR), //Instruction Register
80                        .IDEX_TA(IDEX_TA),  //T Address
81                        .IDEX_M(IDEX_M),    //Memory Control Words(stage 3)
82                        .EXMEM_M(EXMEM_M),  //Memory Control Words(stage 4)
83                        .Branch_s(Branch_s), //Branch Signal
84                        .Stall(Stall),      //Stall
85                        .Flush(Flush));     //Flush
86
87     ///////////////////////////////////////////////////
88     //                      PROGRAM COUNTER CONTROL UNIT
89     //This control unit simple takes care of setting the appropriate signals
90     //for the program counter.
91     ///////////////////////////////////////////////////
92
93     PC_CU PCU (.EXMEM_M(EXMEM_M), //Memory Control Words
94               .Branch_s(Branch_s), //Branch Signal
95               .Stall(Stall),       //Stall
96               .pc_ld(pc_ld),       //PC Load
97               .pc_inc(pc_inc),     //PC Increment
98               .im_cs(im_cs),       //im chip select
99               .im_wr(im_wr),       //im write
100              .im_rd(im_rd));      //im read
101
102     ///////////////////////////////////////////////////
103     //                      INSTRUCTION UNIT
104     //Conatains both the PC and the instruction memory. This is also the
105     //first stage to the pipeline where the instructions are fetched.
106     ///////////////////////////////////////////////////
107     Instruction_Unit IU (.clk(clk),      //clock
108                        .rst(rst),        //reset
109                        .ISR(ISR),        //Interrupt Service routine
110                        .LISR(LISR),      //Leave ISR
111                        .PC_in(PC_in),    //Program Counter Input
112                        .pc_ld(pc_ld),    //Program Counter Load
113                        .pc_inc(pc_inc),  //PC Increment
114                        .im_cs(im_cs),    //im chip select

```

```
115             .im_wr(im_wr),           //im write
116             .im_rd(im_rd),           //im read
117             .Stall(Stall),            // Stall
118             .Flush(Flush),            //Flush
119             .PC_out(PC_out),           //Program Counter Output
120             .IFID_IR(IFID_IR));       //Instruction Register
121
122     //////////////////////////////////////
123     //                                DATAPATH
124     //The rest of the pipeline stages are located here. As well as all other
125     // operations are performed in the datapath.
126     //////////////////////////////////////
127     DataPath DP (.clk(clk),            //clock
128                 .rst(rst),             //reset
129                 .IE(IE),               //Interrupt Enable
130                 .ISR(ISR),             //Interrupt Service Routin
131                 .LISR(LISR),           //Leave ISR
132                 .EX(EX),               //Execute Control Words
133                 .M(M),                 //Memory Control Words
134                 .WB(WB),               //Write Back Control Words
135                 .Flush(Flush),         //Flush
136                 .Stall(Stall),         //Stall
137                 .IFID_IR(IFID_IR),     //Instruction Register
138                 .PC_out(PC_out),        //Program Counter
139                 .M_Data(M_Data),        //Memory Data
140                 .io_out(io_out),        //IO Module out
141                 .IDEX_TA(IDEX_TA),      //T Address
142                 .IDEX_M(IDEX_M),        //Memory Control Words(stage3)
143                 .EXMEM_M(EXMEM_M),      //Memory Control Words(stage34)
144                 .pc4_mux(pc4_mux),      //data to memory
145                 .EXMEM_ALU(EXMEM_ALU),  //ALU
146                 .PC_in(PC_in),          //Program Counter Input
147                 .Branch_s(Branch_s));   //Branch Signal
148
149
150
151
152
153
154     endmodule
155
```



```

1  `timescale 1ns / 1ps
2  /*****
3  * Authors:  Javier Garcia
4  *           Taylor Cochran
5  *
6  * Emails:   javis9526@yahoo.com
7  *           TayCochran123@gmail.com
8  *
9  * Filename: Control_Unit.v
10 * Date:      November 27, 2018
11 * Version:   1.3 Added RTL
12 *
13 *
14 *****/
15 module Control_Unit(clk,rst,IE,ISR,LISR,EXMEM_M,IFID_IR,Branch_s,intr,intr_ack,
16                     EX,M,WB );
17
18     input          clk,rst; // clock, reset
19     input          Branch_s; // branch signal
20     input          intr;    // interrupt
21     input [15:0] EXMEM_M; //EXMEM memory control word
22     input [31:0] IFID_IR; // instruction register
23
24     output [2:0] LISR;    // leave interrupt service routine
25     output reg   intr_ack; // interrupt acknowledge
26     output reg [15:0] EX;    // execute control word - origin
27     output reg [15:0] M;     // memory control word - origin
28     output reg [3:0] WB;     // write back control word - origin
29
30 //-----//
31 ////////////////////////////////////////////////////
32 //      Control Word BreakDown
33 //
34 // EX: | HILO_L | T_mux | Addr_mux | Y_mux | Shamt | Function |
35 //      15      14 13      12 11      10 9      5 4      0
36 //
37 //      IO      M
38 // M: | L | PC4 | Flag | IF | PC_in | JF | (cs,wr,rd) | (cs,wr,rd) | Branch |
39 //      15  14   13  12   11  10 9      7 6      4 3      0
40 //
41 // WB: | Break | WB_Data | D_En |
42 //      3 2      1      0
43 ////////////////////////////////////////////////////
44 output wire ISR; // interupt sevice routine (used for setup)
45
46 output reg IE; // interrupt enable
47
48 reg [2:0] LISR; // leave interrupt service routine
49
50 ////////////////////////////////////////////////////
51 // All the R-type instructions are given parameter names in order to improve
52 // legibility in the control block.
53 ////////////////////////////////////////////////////
54 parameter r_type = 6'h00, sll  = 6'h00, srl  = 6'h02, sra = 6'h03,
55            jr    = 6'h08, mfhi = 6'h10, mflo = 6'h12, mul = 6'h18,
56            div   = 6'h1A, add  = 6'h20, addu = 6'h21, sub = 6'h22,
57            subu  = 6'h23, And  = 6'h24, Or   = 6'h25, Xor = 6'h26,

```

```

58      Nor      = 6'h27,  slt  = 6'h2A,  sltu = 6'h2B,  Break = 6'h0D,
59      setie = 6'h1F;
60
61      //////////////////////////////////////////////////
62      // Just as the above instructions, all other instructions are also given a
63      // parameter name.
64      //////////////////////////////////////////////////
65      parameter beq = 6'h04, bne = 6'h05, blez = 6'h06, bgtz = 6'h07,
66      addi = 6'h08, slti = 6'h0A, sltiu = 6'h0B, andi = 6'h0C,
67      ori = 6'h0D, xori = 6'h0E, lui = 6'h0F, lw = 6'h23,
68      sw = 6'h2B, j = 6'h02, jal = 6'h03, Input = 6'h1C,
69      Output = 6'h1D, reti = 6'h1E, e_key = 6'h1F;
70
71
72      // Start the interrupt service routine when the interrupt enable is active,
73      // interrupt is active high, and it has not been acknowledged yet.
74      assign ISR = intr & IE & !intr_ack;
75
76
77
78      reg [3:0] ISR_StepWire; // These registers are used as counters in order
79      reg [3:0] ISR_StepReg; // to allow the ISR to properly set up.
80
81      //////////////////////////////////////////////////
82      // Here the counter for the ISR is incremented on every clock tick
83      // allowing for proper stage to stage setup.
84      //////////////////////////////////////////////////
85      always@(posedge clk, posedge rst)
86          if(rst) ISR_StepReg <= 4'b0;
87          else     ISR_StepReg <= ISR_StepWire;
88
89      //////////////////////////////////////////////////
90      // This counter is used to allow the cpu to exit the ISR allowing it
91      // enough time to properly pop the pc and flags.
92      //////////////////////////////////////////////////
93      always@(posedge clk, posedge rst)
94          if(rst) LISR = 0;else
95          if(LISR) LISR = LISR +1 ;
96
97      //////////////////////////////////////////////////
98      // Synchronous block that acknowledges the interrupt once the setup has been
99      // completed. Acknowledging the interrupt ends the setup and enters the ISR.
100     //////////////////////////////////////////////////
101     always@(posedge clk, posedge rst)
102         if(rst) intr_ack = 0;else
103         if(EXMEM_M[12]) intr_ack = 1'b1;else // when int. flag is high ack = 1
104         if(intr_ack & (intr == 0)) intr_ack = 0;else // turn of ack after
105             intr_ack = intr_ack; // idle state
106
107     //////////////////////////////////////////////////
108     // The control unit, the control words for each instructions coming into the
109     // pipeline are determined here based on the type of instruction as well
110     // as the op codes and values in the instruction register(IR).
111     //////////////////////////////////////////////////
112     always @(*)begin
113         if(rst)begin
114             EX = 16'b0_0_11_00_00000_10101; // $sp setup mod 13

```

```
115      M = 16'b0_0_0_0_0_0_000_000_0000; //
116      WB = 4'b0_0_0_1; //
117
118
119      IE = 1'b0;
120      ISR_StepWire = 0;
121
122  end
123  else if (ISR) begin
124      case (ISR_StepReg)
125
126          //store flags
127          4'h0: begin //Step 1 stores current PC at $ra *NOTE: This is just for
128                  //module 13, 14 will do the $sp pointer store
129                  //also increment the ISR_Step
130                  //Control_EX = Set up the YMUX to pass in the PC4 Value, and
131                  //toggle the WB Dest. to be $ra
132                  //Control_M = Do nothing in this stage
133                  //Control_WB = WRITE to R[31] the $ra
134                  EX = 16'b0_0_11_00_00000_10001;
135                  M = 16'b0_0_1_0_0_0_000_110_0000;
136                  WB = 4'b0_00_1;
137                  ISR_StepWire = 4'h1;
138              end
139
140          4'h1: begin
141                  //store pc
142                  //Step Loads the PC with the the value from dMEM[0x3FC]
143                  //need to add in the appropriate hardware for this to work
144                  //Control_EX = use appropriate setup for getting RS to be
145                  //$31 = $ra
146                  //Control_M = Take the Value read from dMEM[0x3FC] and jump
147                  //to it *add hardware
148                  //Control_WB = do nothing
149                  EX = 16'b0_0_11_00_00000_10001;
150                  M = 16'b0_1_0_0_0_0_000_110_0000;
151                  WB = 4'b0_00_1;
152                  ISR_StepWire = 4'h2;
153
154
155
156              end
157          4'h2: begin
158              // jump to isr
159              EX = 16'b0_0_00_00_00000_10101;
160              M = 16'b0_0_0_1_0_0_000_101_0000;
161              WB = 4'b0_00_0;
162              ISR_StepWire = 4'h3;
163          end
164      endcase
165
166
167
168  end
169  else
170      if (LISR) //
171          case (LISR)
```

```
172      //
173      3'b001:begin
174          EX = 16'b0_0_11_00_00000_10010;
175          M = 16'b0_0_0_0_0_0_000_101_0000;
176          WB = 4'b0_00_1; end
177      //
178      3'b010:begin
179          EX = 16'b0_0_11_00_00000_10010;
180          M = 16'b0_0_0_0_0_0_000_101_0000;
181          WB = 4'b0_00_1;end
182      //
183      3'b011:begin
184          EX = 16'b0_0_11_00_00000_10010;
185          M = 16'b0_0_0_0_0_0_000_101_0000;
186          WB = 4'b0_00_1;end
187      //
188      3'b100:begin
189          EX = 16'b0_0_11_00_00000_10010;
190          M = 16'b0_0_0_0_0_0_000_101_0000;
191          WB = 4'b0_00_0;end
192      endcase
193  else
194      case(IFID_IR[31:26])
195          r_type:
196              case(IFID_IR[5:0])
197                  add: begin
198                      EX = 16'b0_0_00_00_00000_00010;
199                      M = 16'b0_0_0_0_0_0_000_000_0000;
200                      WB = 4'b0_0_0_1; end
201
202                  sll: begin
203                      EX = {6'b0_0_00_00,IFID_IR[10:6],5'b01100};
204                      M = 16'b0_0_0_0_0_0_000_000_0000;
205                      WB = 4'b0_0_0_1; end
206
207                  srl: begin
208                      EX = {6'b0_0_00_00,IFID_IR[10:6],5'b01101};
209                      M = 16'b0_0_0_0_0_0_000_000_0000;
210                      WB = 4'b0_0_0_1; end
211
212                  sra: begin
213                      EX = {6'b0_0_00_00,IFID_IR[10:6],5'b001110};
214                      M = 16'b0_0_0_0_0_0_000_000_0000;
215                      WB = 4'b0_0_0_1; end
216
217                  //////////////////////////////////////////
218                  jr: begin
219                      EX = 16'b0_0_00_00_00000_00000;
220                      M = 16'b0_0_0_0_1_0_000_000_0000;
221                      WB = 4'b0_0_0_0; end
222
223                  mfhi: begin
224                      EX = 16'b0_0_00_10_00000_00000;
225                      M = 16'b0_0_0_0_0_0_000_000_0000;
226                      WB = 4'b0_0_0_1; end
227
228                  mflo: begin
229                      EX = 16'b0_0_00_01_00000_00000;
230                      M = 16'b0_0_0_0_0_0_000_000_0000;
231                      WB = 4'b0_0_0_1; end
232
233                  mul: begin
234                      EX = 16'b1_0_00_00_00000_11110;
```

```
229      M = 16'b0_0_0_0_0_0_000_000_0000;
230      WB = 4'b0_0_0_0; end
231  div: begin
232      EX = 16'b1_0_00_00_00000_11111;
233      M = 16'b0_0_0_0_0_0_000_000_0000;
234      WB = 4'b0_0_0_0; end
235  //////////////////////////////////
236
237  addu: begin
238      EX = 16'b0_0_00_00_00000_00100;
239      M = 16'b0_0_0_0_0_0_000_000_0000;
240      WB = 4'b0_0_0_1; end
241  sub: begin
242      EX = 16'b0_0_00_00_00000_00011;
243      M = 16'b0_0_0_0_0_0_000_000_0000;
244      WB = 4'b0_0_0_1; end
245  subu: begin
246      EX = 16'b0_0_00_00_00000_00101;
247      M = 16'b0_0_0_0_0_0_000_000_0000;
248      WB = 4'b0_0_0_1; end
249  And: begin
250      EX = 16'b0_0_00_00_00000_01000;
251      M = 16'b0_0_0_0_0_0_000_000_0000;
252      WB = 4'b0_0_0_1; end
253  Or: begin
254      EX = 16'b0_0_00_00_00000_01001;
255      M = 16'b0_0_0_0_0_0_000_000_0000;
256      WB = 4'b0_0_0_1; end
257  Xor: begin
258      EX = 16'b0_0_00_00_00000_01010;
259      M = 16'b0_0_0_0_0_0_000_000_0000;
260      WB = 4'b0_0_0_1; end
261  Nor: begin
262      EX = 16'b0_0_00_00_00000_01011;
263      M = 16'b0_0_0_0_0_0_000_000_0000;
264      WB = 4'b0_0_0_1; end
265  slt: begin
266      EX = 16'b0_0_00_00_00000_00110;
267      M = 16'b0_0_0_0_0_0_000_000_0000;
268      WB = 4'b0_0_0_1; end
269  sltu: begin
270      EX = 16'b0_0_00_00_00000_00111;
271      M = 16'b0_0_0_0_0_0_000_000_0000;
272      WB = 4'b0_0_0_1; end
273
274  Break: begin
275      EX = 16'b0_0_00_00_00000_00000;
276      M = 16'b0_0_0_0_0_0_000_000_0000;
277      WB = 4'b1_0_0_0; end
278
279  setie: begin
280      EX = 16'b0_0_00_00_00000_00000;
281      M = 16'b0_0_0_0_0_0_000_000_0000;
282      WB = 4'b0_0_0_0;
283      IE = 1'b1;end
284
285  endcase
```

```
286      addi: begin
287          EX = 16'b0_1_01_00_00000_00010;
288          M = 16'b0_0_0_0_0_0_000_000_0000;
289          WB = 4'b0_0_0_1; end
290      slti: begin
291          EX = 16'b0_1_01_00_00000_00110;
292          M = 16'b0_0_0_0_0_0_000_000_0000;
293          WB = 4'b0_0_0_1; end
294      sltiu: begin
295          EX = 16'b0_1_01_00_00000_00111;
296          M = 16'b0_0_0_0_0_0_000_000_0000;
297          WB = 4'b0_0_0_1; end
298      andi: begin
299          EX = 16'b0_1_01_00_00000_10110;
300          M = 16'b0_0_0_0_0_0_000_000_0000;
301          WB = 4'b0_0_0_1; end
302      lui: begin
303          EX = 16'b0_1_01_00_00000_11000;
304          M = 16'b0_0_0_0_0_0_000_000_0000;
305          WB = 4'b0_0_0_1; end
306      ori: begin
307          EX = 16'b0_1_01_00_00000_10111;
308          M = 16'b0_0_0_0_0_0_000_000_0000;
309          WB = 4'b0_0_0_1; end
310      xori: begin
311          EX = 16'b0_1_01_00_00000_11001;
312          M = 16'b0_0_0_0_0_0_000_000_0000;
313          WB = 4'b0_0_0_1; end
314      sw: begin
315          EX = 16'b0_1_00_00_00000_00010;
316          M = 16'b0_0_0_0_0_0_000_110_0000;
317          WB = 4'b0_0_0_0; end
318      lw: begin
319          EX = 16'b0_1_01_00_00000_00010;
320          M = 16'b0_0_0_0_0_0_000_101_0000;
321          WB = 4'b0_0_1_1; end
322
323      beq: begin
324          EX = 16'b0_0_00_00_00000_00011;
325          M = 16'b0_0_0_0_0_0_000_000_0001;
326          WB = 4'b0_0_0_0; end
327
328      bne: begin
329          EX = 16'b0_0_00_00_00000_00011;;
330          M = 16'b0_0_0_0_0_0_000_000_0010;
331          WB = 4'b0_00_0; end
332      blez: begin
333          EX = 16'b0_0_00_00_00000_00011;
334          M = 16'b0_0_0_0_0_0_000_000_0100;
335          WB = 4'b0_00_0; end
336      bgtz: begin
337          EX = 16'b0_0_00_00_00000_00011;
338          M = 16'b0_0_0_0_0_0_000_000_1000;
339          WB = 4'b0_00_0; end
340
341      j: begin
342          EX = 16'b0_0_10_11_00000_00000;
```

```
343             M = 16'b0_0_0_0_0_1_000_000_0000;
344             WB = 4'b0_00_0; end
345         jal: begin
346             EX = 16'b0_0_10_11_00000_00010;
347             M = 16'b0_0_0_0_0_1_000_000_0000;
348             WB = 4'b0_00_1; end
349         Output: begin
350             EX = 16'b0_1_00_00_00000_00010;
351             M = 16'b0_0_0_0_0_0_110_000_0000;
352             WB = 4'b0_00_0; end
353         Input: begin
354             EX = 16'b0_1_01_00_00000_00010;
355             M = 16'b0_0_0_0_0_0_101_000_0000;
356             WB = 4'b0_10_1; end
357
358         reti: begin
359             EX = 16'b0_0_11_00_00000_00000;
360             M = 16'b1_0_0_0_0_0_000_101_0000;
361             WB = 4'b0_00_1;
362             LISR = 1; end
363     endcase
364 end
365
366 endmodule
367
```

```
1  `timescale 1ns / 1ps
2  /*****
3  * Authors:   Javier Garcia
4  *           Taylor Cochran
5  *
6  * Emails:    javis9526@yahoo.com
7  *           TayCochran123@gmail.com
8  *
9  * Filename:  Control_Hazard.v
10 * Date:      November 24, 2018
11 * Version:   1.2
12 *
13 *                                                    (End of Page Width)
14 *****/
15 module Control_Hazard(IFID_IR, IDEX_TA, IDEX_M, EXMEM_M,
16                      Branch_s,
17                      Stall, Flush);
18
19     input [31:0] IFID_IR; // instruction register
20     input [4:0]  IDEX_TA; // t register address
21     input [15:0] IDEX_M, EXMEM_M; // control words
22     input        Branch_s; // branch signal
23
24     output reg Stall;
25     output reg Flush;
26     //////////////////////////////////////////////////
27     // This combo logic block determines wether the pipeline
28     // needs to be flushed. Flushes usually occur when there
29     // is a hazard due to a branch/jump.
30     //////////////////////////////////////////////////
31     always @(*) begin
32         // if there is a jump_r,jump,or branch instruction
33         // flush the pipeline.
34         if(EXMEM_M[11]|EXMEM_M[10]| Branch_s)
35             Flush = 1;
36         else
37             Flush = 0;end
38
39     //////////////////////////////////////////////////
40     // This combo logic determines wether we need to stall the
41     // pipeline.
42     //////////////////////////////////////////////////
43     always @(*) begin
44         Stall = 0;
45         // when a load word is detected
46         if(IDEX_M[4])
47             // if the destination equals the next instructions source
48             // then stall.
49             if((IFID_IR[25:21] == IDEX_TA))
50                 Stall = 1;
51         else
52             // if the destination equals the next instructions source
53             // then stall.
54             if(IFID_IR[20:16] == IDEX_TA)
55                 Stall = 1;
56     end
57 endmodule
```



```
1  `timescale 1ns / 1ps
2  /*****
3  * Authors:   Javier Garcia
4  *           Taylor Cochran
5  *
6  * Emails:    javis9526@yahoo.com
7  *           TayCochran123@gmail.com
8  *
9  * Filename:  PC_CU.v
10 * Date:      November 26, 2018
11 * Version:   1.2
12 *
13 *                                                    (End of Page Width)
14 *****/
15 module PC_CU(EXMEM_M,Branch_s,Stall,
16             pc_ld, pc_inc, im_cs, im_wr, im_rd);
17 //-----//
18     input [15:0] EXMEM_M; // memory control word
19     input      Branch_s; // branch signal
20     input      Stall;
21
22     output pc_ld,pc_inc; // load,increment
23     output im_cs,im_wr,im_rd;// chip select, write, read
24
25     ///////////////////////////////////////////////////
26     // The pc load and increment signals are determined by 3 cases:
27     // •if entering/leaving the ISR or jumping or branching : load the new PC
28     // •if there is a stall signal neither load or increment
29     // •otherwise continuously increment the PC
30     ///////////////////////////////////////////////////
31     assign {pc_ld,pc_inc} =(EXMEM_M[15]|EXMEM_M[12]| EXMEM_M[11] |
32                             EXMEM_M[10] | Branch_s)? 2'b10:
33                             (Stall)? 2'b00:
34                             2'b01;
35
36     // When ever there is a stall stop reading from the instruction memory
37     // otherwise continuously read.
38     assign {im_cs,im_wr,im_rd} = (Stall)? 3'b0: 3'b101;
39
40
41
42 endmodule
43
```

```

1  `timescale 1ns / 1ps
2  /***** C E C S  4 4 0 *****/
3  *
4  * File Name:  Instruction_Unit.v
5  * Project:    Lab_Assignment_5
6  * Designer:   Javier Garcia & Taylor Cochran
7  * Email:      javis9526@yahoo.com,TayCochran123@gmail.com
8  * Rev. No.:   Version 1.2
9  * Rev. Date:  10-20-18
10 *
11 * Purpose: This module keeps track of the instructions. PC is found here and
12 *          updated after every instruction is executed. The instructions are
13 *          stored in the instruction memory module.
14 *****/
15 module Instruction_Unit(clk,rst,ISR,LISR,
16                        PC_in,pc_ld,pc_inc,
17                        im_cs,im_wr,im_rd,
18                        Stall,Flush,
19                        PC_out,
20                        IFID_IR);
21
22     input      clk , rst;
23     input      ISR; // interrupt service routine
24     input      [2:0] LISR; // leave interrupt
25     input      pc_ld , pc_inc; // load, increment
26     input      [31:0] PC_in; // input
27     input      im_cs , im_wr , im_rd; //chip select, write, read
28     input      Stall,Flush;
29
30     output reg  [31:0] IFID_IR ; // pc, instruction register
31
32     output      [31:0] PC_out;
33
34     reg         [31:0] PC;
35     wire        [31:0] D_Out; // instruction memory output
36
37     ///////////////////////////////////////////////////
38     // The program counter is determines synchronously in order to access a new
39     // instruction every clock tick.
40     ///////////////////////////////////////////////////
41     always@(posedge clk,posedge rst) begin
42         if(rst) PC = 32'b0; else // reset the program counter
43         if(pc_ld) PC = PC_in;else // when the load is high load the new pc
44         if(ISR|LISR) PC = PC;else // if entering/leaving the ISR, stall
45         if(pc_inc) PC = PC + 4;else // increment the pc by 4
46             PC = PC; // Stall
47     end
48
49     // 32-bit wire that carries the pc value out
50     // of the module.
51     assign PC_out = PC;
52
53     ///////////////////////////////////////////////////
54     // This block is the memory that contains all the instructions. Because
55     // the memory is byte addressable the PC is in increments of 4.
56     // We never write to this memory, we only read.
57     // Note: the address only uses 12-bits because out biggest accesible

```

```
58      //      address is 0xFFF.
59      //////////////////////////////////////////////////
60      InstructionMemory IM (.clk(clk),
61                           .Address(PC_out[11:0]), .D_In(32'h0),
62                           .im_cs(im_cs), .im_wr(im_wr), .im_rd(im_rd),
63                           .D_Out(D_Out));
64
65      //////////////////////////////////////////////////
66      // Synchronously load the first pipeline stage register that contains the
67      // instruction. This instruction will determine all the following pipeline
68      // stage control words.
69      //////////////////////////////////////////////////
70      always@(posedge clk, posedge rst)begin
71          if(rst) IFID_IR = 32'b0;else      // reset the instruction register
72          if(Flush) IFID_IR = 32'b0;else    // flush the instruction register
73          if(Stall) IFID_IR = IFID_IR;else  // stall the instruction register
74              IFID_IR = D_Out;             // load the new instruction
75      end
76
77 endmodule
78
```

```

1  `timescale 1ns / 1ps
2  /***** C E C S  4 4 0 *****/
3  *
4  * File Name:  InstructionMemory.v
5  * Project:    Lab_Assignment_5
6  * Designer:   Javier Garcia & Taylor Cochran
7  * Email:      javis9526@yahoo.com,TayCochran123@gmail.com
8  * Rev. No.:   Version 1.2
9  * Rev. Date:  10-20-18
10 *
11 * Purpose:     The instructions that are used in this system are stored here
12 *              in a memory register whos size is  [7:0] and depth is[4095].
13 *****/
14 module InstructionMemory(clk,
15                          Address, D_In,
16                          im_cs,im_wr,im_rd,
17                          D_Out);
18
19     input                clk;
20
21     input  [31:0]        D_In; // iMem input
22
23     input  [11:0]        Address;
24     input                im_cs,im_wr,im_rd;
25
26     output [31:0]        D_Out; // iMem output
27
28     reg  [7:0]           M  [4095:0]; // memory of width: 8, depth: 4096
29
30     wire  [31:0]         D_Out;
31
32     ///////////////////////////////////////////////////
33     // Synchronous block that writes to the memory. An instruction takes 32-bits,
34     // but since the memory is byte adressible we must access 4-memory location
35     // per instruction.
36     ///////////////////////////////////////////////////
37     always@(posedge clk)
38         if(im_wr & im_cs){M[Address],M[Address+1],M[Address+2],M[Address+3]} = D_In;
39
40     ///////////////////////////////////////////////////
41     //Asynchronous block that reads from the memory.
42     ///////////////////////////////////////////////////
43     assign D_Out =(im_rd&im_cs)? {M[Address],M[Address+1],M[Address+2],M[Address+3]}:
44                                     32'bz;
45
46 endmodule
47

```

```

1  `timescale 1ns / 1ps
2  /*****
3  * Authors:  Javier Garcia
4  *           Taylor Cochran
5  *
6  * Emails:   javis9526@yahoo.com
7  *           TayCochran123@gmail.com
8  *
9  * Filename: DataPath.v
10 * Date:      November 26, 2018
11 * Version:   1.3 Added the Stall Fix
12 *
13 *                                                    (End of Page Width)
14 *****/
15 module DataPath(clk,rst,IE,ISR,LISR,
16                 EX,M,WB,Flush,Stall,
17                 IFID_IR,PC_out, M_Data,io_out, IDEX_TA, IDEX_M,
18                 EXMEM_M,pc4_mux, EXMEM_ALU,PC_in,Branch_s);
19
20     input clk,rst,IE,ISR; // clock, reset, interrupt enable, interr. servie routine
21     input [2:0] LISR; // leave interrupt service
22     input [15:0] EX; // execute control words
23     input [15:0] M ; // memory control words
24     input [3:0] WB; // write back control words
25 //-----//
26     input      Flush,Stall;
27
28     input      [31:0] IFID_IR,PC_out,M_Data,io_out; //instruction,pc, memory, io
29
30     output     [4:0] IDEX_TA; // t register address
31
32     output     [15:0] IDEX_M,EXMEM_M; // memory control words
33
34     output     [31:0] pc4_mux, EXMEM_ALU,PC_in; //data to memory, alu, pc
35
36     output     Branch_s; //branch signal
37
38     wire [31:0] PC_in, IFID_IR ; //pc , instruction register
39
40     wire [15:0] IDEX_EX;
41     wire [15:0] IDEX_M ;
42     wire [3:0]  IDEX_WB;
43     wire [31:0] IDEX_SE, IDEX_PC, IDEX_S, IDEX_T, IDEX_JA;
44     wire [4:0]  IDEX_SA, IDEX_TA, IDEX_DA;
45
46     wire [15:0] EXMEM_M;
47     wire [3:0]  EXMEM_WB;
48     wire [31:0] EXMEM_Baddr,EXMEM_Jaddr;
49     wire [3:0]  EXMEM_FLAGS;
50     wire [31:0] EXMEM_ALU,EXMEM_MData;
51     wire [4:0]  EXMEM_Waddr;
52
53     wire [3:0]  MEMWB_WB;
54     wire [31:0] MEMWB_MData, MEMWB_IO, MEMWB_ALU;
55     wire [4:0]  MEMWB_Waddr;
56
57     wire [31:0] WB_Data;

```

```

58     wire    [4:0] WB_Addr;
59     wire          WB_D_EN;
60
61
62     ////////////////////////////////////////////////////
63     // Note: many of the wires have an index at the beginning of their      //
64     //       name to signify their origin. Their origin being the pipeline  //
65     //       stage registers that they come from.                          //
66     //                                                                    //
67     // • IFID   : Instruction Fetch/ Instruction Decode registers            //
68     // • IDEX   : Instruction Decode/ Execution registers                    //
69     // • EXMEM  : Execution/ Memory registers                              //
70     // • MEMWB  : Memory/ Write Back registers                            //
71     ////////////////////////////////////////////////////
72
73     ////////////////////////////////////////////////////
74     // Stage 2 of Pipeline is the Decode stage where the instruction is read and
75     // the pertaining control words are set. This stage also houses the register files
76     // containing the source/destination registers.
77     ////////////////////////////////////////////////////
78     PipeLine_Stage_2  PS2    (.clk(clk), .rst(rst), // clock and reset
79                               .IFID_IR(IFID_IR),   // Instruction Register
80                               .PC(PC_out),         // Program Counter
81                               .ISR(ISR),           // Interrupt Service Routine
82                               .LISR(LISR),         // Leave Interrupt Service Routine
83                               .WB_Data(WB_Data),   // Write Back Data
84                               .WB_Addr(WB_Addr),   // Write Back Address
85                               .WB_D_EN(WB_D_EN),   // Write Back Enable
86                               .Flush(Flush),       // Flush
87                               .Stall(Stall),       // Stall
88                               .EX(EX),             // Execute Control Word
89                               .M(M),               // Memory Control Word
90                               .WB(WB),             // Write Back Control Word
91                               .IDEX_EX(IDEX_EX),   // Execute Control Word
92                               .IDEX_M(IDEX_M),     // Memory Control Word
93                               .IDEX_WB(IDEX_WB),   // Write Back Control Word
94                               .IDEX_SA(IDEX_SA),   // Source Register S Address
95                               .IDEX_TA(IDEX_TA),   // Source/Dest. Register T Address
96                               .IDEX_DA(IDEX_DA),   // Destination Register D Address
97                               .IDEX_SE(IDEX_SE),   // Sign Extended 16
98                               .IDEX_PC(IDEX_PC),   // Program counter
99                               .IDEX_S(IDEX_S),     // Source S contents
100                              .IDEX_T(IDEX_T),     // Source T contents
101                              .IDEX_JA(IDEX_JA));   // Jump Address
102
103     ////////////////////////////////////////////////////
104     // Stage 3 of the pipeline is the execution stage. After decoding and setting the
105     // appropriate key words the alu will perform the appropriate operations.
106     // This stage also houses the forwarding unit that handles any data dependencies.
107     ////////////////////////////////////////////////////
108     PipeLine_Stage_3  PS3    (.clk(clk), .rst(rst), // clock and reset
109                               .Flush(Flush),       // Flush
110                               .IDEX_EX(IDEX_EX),   // Execute Control Word
111                               .IDEX_M(IDEX_M),     // Memory Control Words
112                               .IDEX_WB(IDEX_WB),   // Write Back Control Words
113                               .IDEX_SA(IDEX_SA),   // Source S Address
114                               .IDEX_TA(IDEX_TA),   // Source/Dest. T Address

```

```

115      .IDEX_DA(IDEX_DA),           // Destination D Address
116      .IDEX_SE(IDEX_SE),           // Sign Extended 16
117      .IDEX_PC(IDEX_PC),           // Program Counter
118      .IDEX_S(IDEX_S),             // Source S Contents
119      .IDEX_T(IDEX_T),             // Source T Contents
120      .IDEX_JA(IDEX_JA),           // Jump Address
121      .M_Data(M_Data),             // Memory Data
122      .MEMWB_WB(MEMWB_WB[0]),       // Write Back Data Enable
123      .WB_Data(WB_Data),           // Write Back Data
124      .MEMWB_Waddr(MEMWB_Waddr),    // Write Back Address
125      .EXMEM_M(EXMEM_M),           // Memory Control Words
126      .EXMEM_WB(EXMEM_WB),         // Write Back Control Words
127      .EXMEM_Baddr(EXMEM_Baddr),    // Branch Address
128      .EXMEM_Jaddr(EXMEM_Jaddr),    // Jump Address
129      .EXMEM_FLAGS(EXMEM_FLAGS),    // ALU Flags
130      .EXMEM_ALU(EXMEM_ALU),        // ALU output
131      .EXMEM_MData(EXMEM_MData),    // Memory Data
132      .EXMEM_Waddr(EXMEM_Waddr));   // Write Back Address
133
134      //////////////////////////////////////
135      // Stage 4 of the pipeline is the memory stage. This stage will access both the
136      // memory and the IO. However they are not located in here the wires for the modules
137      // are passed out to the cpu. This State also handles the branch and jump
138      // instructions.
139      //////////////////////////////////////
140      PipeLine_Stage_4 PS4 (.clk(clk), .rst(rst),           // clock, reset
141      .IE(IE),           // Interupt Enable
142      .IDEX_PC(IDEX_PC), // Program Counter
143      .EXMEM_M(EXMEM_M), // Memory Control Words
144      .EXMEM_WB(EXMEM_WB), // Write Back Control Words
145      .EXMEM_Baddr(EXMEM_Baddr), // Branch Address
146      .EXMEM_Jaddr(EXMEM_Jaddr), // Jump Address
147      .EXMEM_FLAGS(EXMEM_FLAGS), // Flags
148      .M_Data(M_Data), // Memory Data
149      .io_out(io_out), // IO Data
150      .EXMEM_ALU(EXMEM_ALU), // ALU
151      .EXMEM_MData(EXMEM_MData), // Memory Data
152      .EXMEM_Waddr(EXMEM_Waddr), // Write Back Address
153      .MEMWB_WB(MEMWB_WB), // Write Back Control Words
154      .MEMWB_MData(MEMWB_MData), // Memory Data
155      .MEMWB_IO(MEMWB_IO), // IO Data
156      .MEMWB_ALU(MEMWB_ALU), // ALU
157      .MEMWB_Waddr(MEMWB_Waddr), // Write Back Address
158      .PC_out(PC_in), // Program Counter input
159      .Branch_s(Branch_s), // Branch signal
160      .pc4_mux(pc4_mux)); // Data going to memory
161
162      //////////////////////////////////////
163      // Stage 5 of the pipeline is the write back to the register file stage. However
164      // this is more of a pseudo-module that takes the control words and passes the
165      // correct values back to the state 2 module for storing. This is because the
166      // register files are only found int Stage 2.
167      //////////////////////////////////////
168      PipeLine_Stage_5 PS5 (.clk(clk), .rst(rst),           // clock , reset
169      .MEMWB_WB(MEMWB_WB), // Write Back
170      .MEMWB_MData(MEMWB_MData), // Memory Data
171      .MEMWB_IO(MEMWB_IO), // IO Data

```

```
172      .MEMWB_ALU (MEMWB_ALU),      // ALU
173      .MEMWB_Waddr (MEMWB_Waddr),  // Write Back Address
174      .WB_Data (WB_Data),          // Write Back Data
175      .WB_Addr (WB_Addr),          // Write Back Address
176      .WB_D_EN (WB_D_EN));        // Write Back Data Enable
177
178  endmodule
179
```



```

1  `timescale 1ns / 1ps
2  /*****
3  * Authors:   Javier Garcia
4  *           Taylor Cochran
5  *
6  * Emails:    javis9526@yahoo.com
7  *           TayCochran123@gmail.com
8  *
9  * Filename:  PipeLine_Stage_2.v
10 * Date:      November 24, 2018
11 * Version:   1.5 fixed Stalling error
12 *
13 *                                                    (End of Page Width)
14 *****/
15 module PipeLine_Stage_2 (clk,rst,IFID_IR,PC,ISR,LISR,
16                          WB_Data,WB_Addr,WB_D_EN,
17                          Flush,Stall,EX,M,WB,
18                          IDEX_EX, IDEX_M, IDEX_WB,
19                          IDEX_SA, IDEX_TA, IDEX_DA,
20                          IDEX_SE, IDEX_PC, IDEX_S, IDEX_T, IDEX_JA);
21
22  //////////////////////////////////////////
23  // Inputs
24  //////////////////////////////////////////
25  input      clk,rst,Flush,Stall;
26  input [2:0] LISR; // Leave Interrupt Service Routine
27  input      ISR;  // Interrupt Service Routine
28  input [31:0] IFID_IR,PC,WB_Data;
29  input [4:0]  WB_Addr; // Write Back Address
30  input      WB_D_EN; // Write Back Data Enable
31
32  input [15:0] EX;    // Execute control word
33  input [15:0] M ;    // Memory Control Word
34  input [3:0]  WB;    // Write Back Contro Word
35  //////////////////////////////////////////
36  // Outputs for the IDEX pipeline stage
37  //////////////////////////////////////////
38  output reg [15:0] IDEX_EX; // Execute Control Word
39  output reg [15:0] IDEX_M ; // Memory Control Word
40  output reg [3:0]  IDEX_WB; // Write Back Control Word
41                      //signExtend,  PC,    S,      T,    Jump Address
42  output reg [31:0] IDEX_SE, IDEX_PC, IDEX_S, IDEX_T, IDEX_JA;
43  output reg [4:0]  IDEX_SA, IDEX_TA, IDEX_DA;
44                      // Saddr , Tadd ,  Daddr
45
46
47  //mux for the source address
48  wire [4:0]  Saddr_mux;
49
50  // wires for the register file outputs
51  wire [31:0] S, T;
52
53  //////////////////////////////////////////
54  //                                REGISTER FILE
55  // Internal memory that the Datapath uses to access operands.
56  //////////////////////////////////////////
57

```

```

58      // When Entering/Leaving the ISR use the stack pointer register
59      // as the source register, else use the S source register.
60      assign Saddr_mux = (ISR | LISR)? 5'd29: IFID_IR[25:21];
61
62      // 32 32-bit registers that are used as the sources and destination
63      // for most of the instructions.
64      regfile32 RF (.clk(clk), .rst(rst), .D(WB_Data),
65                  .S_Addr(Saddr_mux), .T_Addr(IFID_IR[20:16]),
66                  .D_En(WB_D_EN), .D_Addr(WB_Addr),
67                  .S(S), .T(T));
68
69      ////////////////////////////////////////////////////
70      // Pipeline Registers are synchronously loaded.
71      ////////////////////////////////////////////////////
72      always @ ( posedge clk, posedge rst) begin
73          if(rst)begin                // upon reset reset all the registers
74              IDEX_EX = EX;           // except the control words, because they
75              IDEX_M  = M;           // initialize the stack pointer on reset.
76              IDEX_WB = WB;
77              IDEX_SA = 5'b0;
78              IDEX_TA = 5'b0;
79              IDEX_DA = 5'b0;
80              IDEX_SE = 32'b0;
81              IDEX_PC = 32'b0;
82              IDEX_S  = 32'b0;
83              IDEX_T  = 32'b0;
84              IDEX_JA = 32'b0;end
85      // When flushing, clear all the registers completetly.
86      else if(Flush | Stall) begin
87          IDEX_EX = 17'b0;
88          IDEX_M  = 12'b0;
89          IDEX_WB = 4'b0;
90          IDEX_SA = 5'b0;
91          IDEX_TA = 5'b0;
92          IDEX_DA = 5'b0;
93          IDEX_SE = 32'b0;
94          IDEX_PC = 32'b0;
95          IDEX_S  = 32'b0;
96          IDEX_T  = 32'b0;
97          IDEX_JA = 32'b0;
98      end
99      else begin
100         IDEX_EX = EX; // Execute Control
101         IDEX_M  = M; // Memory Control
102         IDEX_WB = WB; // Write Back Control
103         IDEX_SA = Saddr_mux; // Source address
104         IDEX_TA = IFID_IR[20:16]; // T Address
105         IDEX_DA = IFID_IR[15:11]; // D Address
106         IDEX_SE = {{16{IFID_IR[15]}}, IFID_IR[15:0]}; // sign extend 16
107         IDEX_PC = (ISR)? PC-8:PC; // ISR clears 2 inst. so pc-8, else pc
108         IDEX_S  = S; // S register
109         IDEX_T  = T; // T register
110         IDEX_JA = {PC[31:28], IFID_IR[25:0], 2'b00}; // jump address
111     end
112 end
113 endmodule
114

```

```

1  `timescale 1ns / 1ps
2  /***** C E C S  4 4 0 *****/
3  *
4  * File Name:  regfile32.v
5  * Project:    Lab_Assignment_5
6  * Designer:   Javier Garcia, Taylor Cochran
7  * Email:      javis9526@yahoo.com,TayCochran123@gmail.com
8  * Rev. No.:   Version 1.7
9  * Rev. Date:  10-22-2018
10 *
11 * Purpose: This module contains the memory registers that hold the operands
12 *          for the integer datapath. Results from the ALU are also written
13 *          back into these registers.
14 *
15 * Notes: reg32 is a 32x32 bit matrix, that contains 32 registers all of
16 *        which are 32-bits wide. Register 0 will always contain 0.
17 *****/
18 module regfile32( clk, rst, D,
19                  S_Addr, T_Addr,
20                  D_En, D_Addr,
21                  S,T);
22
23     input          clk, rst, D_En;
24     input  [31:0]  D; // Data In
25     input  [4:0]   S_Addr, T_Addr, D_Addr;
26     output [31:0]  S , T;
27     wire  [31:0]  S, T;
28
29     reg [31:0] reg32 [31:0]; // register file
30     ///////////////////////////////////////////////////
31     //WRITE the incoming bits to the designated register.
32     ///////////////////////////////////////////////////
33
34     always @ (posedge clk, posedge rst) begin
35         if(rst)
36             reg32[0] = 32'b0;
37         else if(D_En)
38             if(D_Addr!=0) reg32[D_Addr]= D ;
39
40
41
42     end
43
44     ///////////////////////////////////////////////////
45     //READ the contents of a register based on an address.
46     // Writing to the source register causes a forward in
47     // data.
48     ///////////////////////////////////////////////////
49
50     assign S = (D_En & (D_Addr == S_Addr)) ? D : reg32[S_Addr];
51
52     assign T = (D_En & (D_Addr == T_Addr)) ? D : reg32[T_Addr];
53
54
55
56
57 endmodule

```

```
1  `timescale 1ns / 1ps
2  /*****
3  * Authors:   Javier Garcia
4  *           Taylor Cochran
5  *
6  * Emails:    javis9526@yahoo.com
7  *           TayCochran123@gmail.com
8  *
9  * Filename:  PipeLine_Stage_3.v
10 * Date:      November 25, 2018
11 * Version:   1.6 Cleaned up Wiring
12 *
13 *                                                    (End of Page Width)
14 *****/
15 module PipeLine_Stage_3(clk, rst, Flush, IDEX_EX, IDEX_M, IDEX_WB,
16                        IDEX_SA, IDEX_TA, IDEX_DA,
17                        IDEX_SE, IDEX_PC, IDEX_S, IDEX_T, IDEX_JA,
18                        M_Data,
19                        MEMWB_WB,
20                        WB_Data,
21                        MEMWB_Waddr,
22                        EXMEM_M, EXMEM_WB,
23                        EXMEM_Baddr, EXMEM_Jaddr,
24                        EXMEM_FLAGS,
25                        EXMEM_ALU, EXMEM_MData, EXMEM_Waddr);
26
27     input clk, rst, Flush;
28     input [15:0] IDEX_EX;
29     input [15:0] IDEX_M ;
30     input [3:0] IDEX_WB;
31     input MEMWB_WB;
32     input [31:0] WB_Data, M_Data;
33     input [4:0] MEMWB_Waddr;
34     input [31:0] IDEX_SE, IDEX_PC, IDEX_S, IDEX_T, IDEX_JA;
35     input [4:0] IDEX_SA, IDEX_TA, IDEX_DA;
36
37     output reg [15:0] EXMEM_M;
38     output reg [3:0] EXMEM_WB;
39     output reg [31:0] EXMEM_Baddr, EXMEM_Jaddr;
40     output reg [3:0] EXMEM_FLAGS;
41     output reg [31:0] EXMEM_ALU, EXMEM_MData;
42     output reg [4:0] EXMEM_Waddr;
43
44
45     reg [31:0] HI, LO;
46
47
48     wire [31:0] B_addr, Y_hi, Y_lo, ALU_OUT;
49
50     wire [31:0] For_s, For_t;
51
52     wire [31:0] T_Mux;
53
54     wire [4:0] W_Addr;
55
56     wire [1:0] S_sel, T_sel;
57
```

```

58     wire C,N,V,Z;
59
60     //////////////////////////////////////////////////
61     // Forwarding Unit
62     // Handles any dependencies when executing instructions, by
63     // forwarding the appropriate data.
64     //////////////////////////////////////////////////
65     Fowarding FU (
66         .IDEX_SA(IDEX_SA), //S address
67         .IDEX_TA(IDEX_TA), //T address
68         .EXMEM_WB(EXMEM_WB[0]), //Write Back Enable
69         .EXMEM_Waddr(EXMEM_Waddr), //Write Back Address
70         .MEMWB_WB(MEMWB_WB), //Write back control
71         .MEMWB_Waddr(MEMWB_Waddr), //Write back address
72         .S_sel(S_sel), //Select the s forward
73         .T_sel(T_sel)); // select the t forward
74
75
76     //////////////////////////////////////////////////
77     //      For_s & For_t
78     // mux that forwards the appropriate data based on the results
79     // of the forwarding unit.
80     //////////////////////////////////////////////////
81     assign For_s = (S_sel == 1)? EXMEM_ALU://forward alu result
82                     (S_sel == 2)? WB_Data: // forward write back
83                     IDEX_S; // dont forward
84
85     assign For_t = (T_sel == 1)? EXMEM_ALU: //forward alu result
86                     (T_sel == 2)? WB_Data: // forward write back
87                     IDEX_T; // dont forward
88
89     //////////////////////////////////////////////////
90     //      T MUX
91     // Mux decides if the opernad comes from the register files
92     // or the external memory.
93     //////////////////////////////////////////////////
94
95     assign T_Mux =(IDEX_EX[14])? IDEX_SE : For_t;
96
97     //////////////////////////////////////////////////
98     //      W_Addr
99     // Selects the write back address based on the control word for
100    // the execute stage.
101    //////////////////////////////////////////////////
102    assign W_Addr = (IDEX_EX[13:12] == 0)? IDEX_DA:
103                    (IDEX_EX[13:12] == 1)? IDEX_TA:
104                    (IDEX_EX[13:12] == 2)? 5'd31:
105                    (IDEX_EX[13:12] == 3)? 5'd29:
106                    IDEX_DA;
107
108
109    //////////////////////////////////////////////////
110    //      ARITHMETIC LOGIC UNIT
111    // Logic unit that process all the basic arithmetic functions.
112    //
113    //////////////////////////////////////////////////
114

```

```

115     ALU_32    ALU    (.S(For_s),.T(T_Mux),.FS(IDEX_EX[9:0]),
116                .Y_hi(Y_hi),.Y_lo(Y_lo),
117                .C(C),.V(V),.N(N),.Z(Z));
118
119     ///////////////////////////////////////////////////
120     //                      HILO REGISTERS
121     // Temporary registers that are only used to hold the outcomes
122     // of the division and multiplication functions.
123     ///////////////////////////////////////////////////
124
125     always @(posedge clk, posedge rst) begin
126         if(rst)      {HI,LO} = 64'b0;else
127         if(IDEX_EX[15]) {HI,LO} = {Y_hi,Y_lo};
128     end
129
130
131     ///////////////////////////////////////////////////
132     //                      Y MUX
133     // This mux decides what will be output from the ALU.
134     //
135     ///////////////////////////////////////////////////
136
137     assign ALU_OUT = (IDEX_EX[11:10] == 2'd0)?  Y_lo:
138                    (IDEX_EX[11:10] == 2'd1)?  L_O:
139                    (IDEX_EX[11:10] == 2'd2)?  HI:
140                    (IDEX_EX[11:10] == 2'd3)?  IDEX_PC:
141                    Y_lo;
142
143     assign B_addr = IDEX_PC + {IDEX_SE[29:0],2'b0};
144     ///////////////////////////////////////////////////
145     // Pipeline Registers are synchronously loaded.
146     ///////////////////////////////////////////////////
147
148     always @(posedge clk, posedge rst) begin
149         // upon reset completely clear the all the pipeline registers.
150         if (rst)begin
151             EXMEM_M = 13'b0;
152             EXMEM_WB = 4'b0;
153             EXMEM_Baddr = 32'b0;
154             EXMEM_Jaddr = 32'b0;
155             EXMEM_FLAGS = 4'b0;
156             EXMEM_ALU = 32'b0;
157             EXMEM_MData = 32'b0;
158             EXMEM_Waddr = 32'b0; end
159         else
160             // upon flush, clear all the pipeline registes.
161             if(Flush) begin
162                 EXMEM_M = 13'b0;
163                 EXMEM_WB = 4'b0;
164                 EXMEM_Baddr = 32'b0;
165                 EXMEM_Jaddr = 32'b0;
166                 EXMEM_FLAGS = 4'b0;
167                 EXMEM_ALU = 32'b0;
168                 EXMEM_MData = 32'b0;
169                 EXMEM_Waddr = 32'b0;
170             end
171             //synchronously update the pipelien registers

```

```
172         else begin
173             EXMEM_M = IDEX_M; // Memory Control
174             EXMEM_WB = IDEX_WB; // Write Back Control
175             EXMEM_Baddr = B_addr; // Branch Address
176             EXMEM_Jaddr = IDEX_JA ; // Jump Addrss
177             // when popping get flags from memory otherwise get from alu
178             EXMEM_FLAGS = ((EXMEM_ALU == 31'h400) & EXMEM_M[4])? M_Data[3:0]:{C,V,N,Z};
179             EXMEM_ALU = ALU_OUT; // ALU Output
180             EXMEM_MData = For_t; // T data from the mux
181             EXMEM_Waddr = W_Addr; // Write Address
182         end
183
184     end
185
186 endmodule
187
```

```

1  `timescale 1ns / 1ps
2  /*****
3  * Authors:  Javier Garcia
4  *           Taylor Cochran
5  *
6  * Emails:   javis9526@yahoo.com
7  *           TayCochran123@gmail.com
8  *
9  * Filename: Fowarding.v
10 * Date:      November 26, 2018
11 * Version:   1.4 fixed the problem with MEM
12 *
13 *                                                    (End of Page Width)
14 *****/
15 module Fowarding(IDEX_SA, IDEX_TA,
16                 EXMEM_WB, EXMEM_Waddr,
17                 MEMWB_WB, MEMWB_Waddr, S_sel, T_sel);
18     input [4:0] IDEX_SA, IDEX_TA;
19     input      EXMEM_WB, MEMWB_WB;
20
21     input [4:0] EXMEM_Waddr, MEMWB_Waddr;
22
23     output reg [1:0] S_sel, T_sel;
24
25     ////////////////////////////////////////////////////
26     // The combo block represents the forwarding logic. Note that this only
27     // handles data dependencies that are not latches. Those are handled by the
28     // register file itself.
29     ////////////////////////////////////////////////////
30     always @(*) begin
31         // if source S matches destination from EXMEM and Write Enable
32         if((IDEX_SA == EXMEM_Waddr) & EXMEM_WB)
33             S_sel = 2'b01; // forward the data from EXMEM
34         else
35             // if source S matches destination from EXMEM and Write Enable
36             if((IDEX_SA == MEMWB_Waddr) & MEMWB_WB)
37                 S_sel = 2'b10; // forward the data from MEMWB
38             else
39                 S_sel = 0; // dont forward any data
40         ////////////////////////////////////////////////////
41         // if source T matches destination from EXMEM and Write Enable
42         if((IDEX_TA == EXMEM_Waddr) & EXMEM_WB)
43             T_sel = 2'b01; // forward the data from EXMEM
44         else
45             // if source T matches destination from EXMEM and Write Enable
46             if((IDEX_TA == MEMWB_Waddr) & MEMWB_WB)
47                 T_sel = 2'b10; // forward the data from MEMWB
48             else
49                 T_sel = 0; // dont forward any data
50     end
51
52     end
53
54
55 endmodule
56

```



```

1  `timescale 1ns / 1ps
2  /***** C E C S 4 4 0 *****/
3  *
4  * File Name:  ALU.v
5  * Project:    Lab_Assignment_1
6  * Designer:   Javier Garcia
7  * Email:      javis9526@yahoo.com
8  * Rev. No.:   Version 1.4
9  * Rev. Date:  9-12-18
10 *
11 * Purpose: Wrapper function that contains the 3 modules needed to implement
12 *          an ALU. The 3 modules take care of defining all results from an
13 *          operation as well as the flags for said operations.
14 *****/
15 module ALU_32( S,T,FS,
16               Y_hi,Y_lo,
17               C,V,N,Z);
18
19     input  [31:0]  S,T;
20     input  [9:0]   FS;
21     output          N,Z,V,C;
22     output [31:0]  Y_hi,Y_lo;
23
24     wire          N,Z,V,C;    //flags
25     wire [31:0]   Y_hi,Y_lo;  // outputs
26
27     wire [63:0]   prod;       //wire for multiplication
28     wire [31:0]   Q,R,OP_R;   //quotient,remainder,operation_result
29     wire [31:0]   bsl,bsr,bsa;
30     wire [3:0]    sllF,srlF,sraF;
31     wire [3:0]    DFLAGS,MFLAGS,FLAGS; // flags for the modules
32
33
34     ///////////////////////////////////////////////////
35     // Modules that are used to perform the arithmetic
36     // functions of the alu.
37     ///////////////////////////////////////////////////
38     DIV_32  DIVISION      (S,T,R,Q,DFLAGS);
39     MPY_32  MULTIPTLI     (S,T,prod,MFLAGS);
40     Shift_left  SLL       (T,FS[9:5],bsl,sllF);
41     Shif_right_logic  SRL   (T,FS[9:5],bsr,srlF);
42     Shift_right_arithmetic  SRA   (T,FS[9:5],bsa,sraF);
43     MIPS_32  MIPS        (S,T,FS,OP_R,FLAGS);
44
45     // mux that assigns the outputs based on the select FS.
46     assign {Y_hi,Y_lo,C,V,N,Z} = (FS[4:0] ==5'h1F)? {R,Q,DFLAGS}://div
47                                     (FS[4:0] ==5'h1E)? {prod,MFLAGS}://mul
48                                     (FS[4:0] == 5'h0C)? {32'b0,bsl,sllF}://sll
49                                     (FS[4:0] == 5'h0D)? {32'b0,bsr,srlF}://srl
50                                     (FS[4:0] == 5'h0E)? {32'b0,bsa,sraF}://sra
51                                     {32'b0,OP_R,FLAGS};//basic alu
52
53
54 endmodule
55

```

```
1  `timescale 1ns / 1ps
2  /***** C E C S  4 4 0 *****/
3  *
4  * File Name:   DIV_32.v
5  * Project:    Lab_Assignment_1
6  * Designer:   Javier Garcia
7  * Email:     javis9526@yahoo.com
8  * Rev. No.:  Version 1.6
9  * Rev. Date:  9-29-18
10 *
11 * Purpose: Divide the S operand by T and then store the quotient in Y_hi
12 *           and the remainder in Y_lo.
13 *
14 *****/
15 module DIV_32( S,T,
16               R,Q,
17               {C,V,N,Z});
18
19     input [31:0] S,T;
20
21     output reg [31:0] Q,R; // quotient, remainder
22
23     output reg N,V,Z,C; // flags
24
25     integer si,ti; // temporary integer
26
27     always @(*)begin
28         si = S; // pass s to temp integer
29         ti = T; // pass t to temp integer
30         Q = si/ti; // divide s by t; Q is quotient R is the remainder
31         R = si%ti; // r gets s mod t
32         N = Q[31]; //negativ flag
33         V = 1'bx; //overflow not affected
34         C = 1'bx; //carry not affected
35         Z = ~(32'hFFFFFFFF & Q); // check for zero
36     end
37
38 endmodule
39
```

```
1  `timescale 1ns / 1ps
2  /***** C E C S  4 4 0 *****/
3  *
4  * File Name:  MPY_32.v
5  * Project:    Lab_Assignment_1
6  * Designer:   Javier Garcia
7  * Email:     javis9526@yahoo.com
8  * Rev. No.:  Version 1.5
9  * Rev. Date:  9-30-18
10 *
11 * Purpose: Multiply the 32-bit operands S and T and store the 64-bit result
12 *           in {Y_hi,Y_lo}.
13 *****/
14 module MPY_32( S,T,
15               {Y_Hi,Y_Lo},
16               {C,V,N,Z});
17
18     input [31:0] S,T;
19
20     output reg [31:0] Y_Hi,Y_Lo;
21
22     output reg      N,V,Z,C;
23
24     integer si,ti; // temporary integers
25
26     always @(*) begin
27         si = S;      //pass s to temp integer
28         ti = T;      // pass t to temp integer
29         {Y_Hi,Y_Lo} = si*ti; // store product of S&T
30         N = 0;       // set negative flag
31         V = 1'bx;    // overflow not affected
32         C = 1'bx;    //carry not affected
33         Z = ~(32'hFFFFFFF & {Y_Hi,Y_Lo}); // find zero flag
34     end
35 endmodule
36
```

```
1  `timescale 1ns / 1ps
2  /*****
3  * Authors:  Javier Garcia
4  *           Taylor Cochran
5  *
6  * Emails:   javis9526@yahoo.com
7  *           TayCochran123@gmail.com
8  *
9  * Filename: Shift_left.v
10 * Date:      November 25, 2018
11 * Version:   1.2 More comments and cleaner spacing
12 *
13 *                                                    (End of Page Width)
14 *****/
15 module Shift_left(T,shamt,Y_lo,{C,V,N,Z});
16
17     input [31:0] T;
18     input [4:0] shamt;
19     output reg [31:0] Y_lo;
20     output reg C,V,N,Z;
21
22     ///////////////////////////////////////////////////
23     // Barrel Shifter allows to shift more than one bit to the
24     // left.
25     ///////////////////////////////////////////////////
26     always @(*) begin
27         C = T[31];  N = Y_lo[31];  V =1'bx ;
28         case(shamt) // shift amount
29             5'd0:  Y_lo = T; // dont shift
30             5'd1:  Y_lo = {T[30:0], 1'b0}; // shift left 1-bit
31             5'd2:  Y_lo = {T[29:0], 2'b0}; // shift left 2-bit
32             5'd3:  Y_lo = {T[28:0], 3'b0}; // shift left 3-bit
33             5'd4:  Y_lo = {T[27:0], 4'b0}; // shift left 4-bit
34             5'd5:  Y_lo = {T[26:0], 5'b0}; // shift left 5-bit
35             5'd6:  Y_lo = {T[25:0], 6'b0}; // shift left 6-bit
36             5'd7:  Y_lo = {T[24:0], 7'b0}; // shift left 7-bit
37             5'd8:  Y_lo = {T[23:0], 8'b0}; // shift left 8-bit
38             5'd9:  Y_lo = {T[22:0], 9'b0}; // shift left 9-bit
39             5'd10: Y_lo = {T[21:0],10'b0}; // shift left 10-bit
40             5'd11: Y_lo = {T[20:0],11'b0}; // shift left 11-bit
41             5'd12: Y_lo = {T[19:0],12'b0}; // shift left 12-bit
42             5'd13: Y_lo = {T[18:0],13'b0}; // shift left 13-bit
43             5'd14: Y_lo = {T[17:0],14'b0}; // shift left 14-bit
44             5'd15: Y_lo = {T[16:0],15'b0}; // shift left 15-bit
45             5'd16: Y_lo = {T[15:0],16'b0}; // shift left 16-bit
46             5'd17: Y_lo = {T[14:0],17'b0}; // shift left 17-bit
47             5'd18: Y_lo = {T[13:0],18'b0}; // shift left 18-bit
48             5'd19: Y_lo = {T[12:0],19'b0}; // shift left 19-bit
49             5'd20: Y_lo = {T[11:0],20'b0}; // shift left 20-bit
50             5'd21: Y_lo = {T[10:0],21'b0}; // shift left 21-bit
51             5'd22: Y_lo = {T [9:0],22'b0}; // shift left 22-bit
52             5'd23: Y_lo = {T [8:0],23'b0}; // shift left 23-bit
53             5'd24: Y_lo = {T [7:0],24'b0}; // shift left 24-bit
54             5'd25: Y_lo = {T [6:0],25'b0}; // shift left 25-bit
55             5'd26: Y_lo = {T [5:0],26'b0}; // shift left 26-bit
56             5'd27: Y_lo = {T [4:0],27'b0}; // shift left 27-bit
57             5'd28: Y_lo = {T [3:0],28'b0}; // shift left 28-bit
```

```
58         5'd29: Y_lo = {T [2:0],29'b0};// shift left 29-bit
59         5'd30: Y_lo = {T [1:0],30'b0};// shift left 30-bit
60         5'd31: Y_lo = {T [0],31'b0};// shift left 31-bit
61         default: Y_lo = 0;// dont shift
62     endcase
63 end
64 endmodule
65
```

```
1  `timescale 1ns / 1ps
2  /*****
3  * Authors:  Javier Garcia
4  *           Taylor Cochran
5  *
6  * Emails:   javis9526@yahoo.com
7  *           TayCochran123@gmail.com
8  *
9  * Filename: Shift_right_logic.v
10 * Date:      November 25, 2018
11 * Version:   1.2 More comments and cleaner spacing
12 *
13 *                                                    (End of Page Width)
14 *****/
15 module Shif_right_logic(T,shamt,Y_lo,{C,V,N,Z});
16
17     input [31:0] T;
18     input [4:0] shamt;
19     output reg [31:0]Y_lo;
20     output reg C,V,N,Z;
21
22     ////////////////////////////////////////////////////
23     // Barrel shifter allows to shift more than one bit to the right.
24     ////////////////////////////////////////////////////
25     always @(*) begin
26         C = T[0];      N = Y_lo[31];  V =1'bx ;
27         case(shamt) // shift amount;
28             5'd0: Y_lo = T;// dont shift
29             5'd1: Y_lo = {1'b0,T[31:1]};// shift right 1-bit
30             5'd2: Y_lo = {2'b0,T[31:2]};// shift right 2-bit
31             5'd3: Y_lo = {3'b0,T[31:3]};// shift right 3-bit
32             5'd4: Y_lo = {4'b0,T[31:4]};// shift right 4-bit
33             5'd5: Y_lo = {5'b0,T[31:5]};// shift right 5-bit
34             5'd6: Y_lo = {6'b0,T[31:6]};// shift right 6-bit
35             5'd7: Y_lo = {7'b0,T[31:7]};// shift right 7-bit
36             5'd8: Y_lo = {8'b0,T[31:8]};// shift right 8-bit
37             5'd9: Y_lo = {9'b0,T[31:9]};// shift right 9-bit
38             5'd10: Y_lo = {10'b0,T[31:10]};// shift right 10-bit
39             5'd11: Y_lo = {11'b0,T[31:11]};// shift right 11-bit
40             5'd12: Y_lo = {12'b0,T[31:12]};// shift right 12-bit
41             5'd13: Y_lo = {13'b0,T[31:13]};// shift right 13-bit
42             5'd14: Y_lo = {14'b0,T[31:14]};// shift right 14-bit
43             5'd15: Y_lo = {15'b0,T[31:15]};// shift right 15-bit
44             5'd16: Y_lo = {16'b0,T[31:16]};// shift right 16-bit
45             5'd17: Y_lo = {17'b0,T[31:17]};// shift right 17-bit
46             5'd18: Y_lo = {18'b0,T[31:18]};// shift right 18-bit
47             5'd19: Y_lo = {19'b0,T[31:19]};// shift right 19-bit
48             5'd20: Y_lo = {20'b0,T[31:20]};// shift right 20-bit
49             5'd21: Y_lo = {21'b0,T[31:21]};// shift right 21-bit
50             5'd22: Y_lo = {22'b0,T[31:22]};// shift right 22-bit
51             5'd23: Y_lo = {23'b0,T[31:23]};// shift right 23-bit
52             5'd24: Y_lo = {24'b0,T[31:24]};// shift right 24-bit
53             5'd25: Y_lo = {25'b0,T[31:25]};// shift right 25-bit
54             5'd26: Y_lo = {26'b0,T[31:26]};// shift right 26-bit
55             5'd27: Y_lo = {27'b0,T[31:27]};// shift right 27-bit
56             5'd28: Y_lo = {28'b0,T[31:28]};// shift right 28-bit
57             5'd29: Y_lo = {29'b0,T[31:29]};// shift right 29-bit
```

```
58         5'd30: Y_lo = {30'b0,T[31:30]}; // shift right 30-bit
59         5'd31: Y_lo = {31'b0,T[31]}; // shift right 31-bit
60         default: Y_lo = T; // dont shift
61     endcase
62 end
63 endmodule
64
```

```
1  `timescale 1ns / 1ps
2  /*****
3  * Authors:  Javier Garcia
4  *           Taylor Cochran
5  *
6  * Emails:   javis9526@yahoo.com
7  *           TayCochran123@gmail.com
8  *
9  * Filename: Shift_right_arithmetic.v
10 * Date:      November 25, 2018
11 * Version:   1.2 More comments and cleaner spacing
12 *
13 *                                                    (End of Page Width)
14 *****/
15 module Shift_right_arithmetic(T,shamt,Y_lo,{C,V,N,Z});
16
17     input [31:0] T;
18     input [4:0] shamt;
19     output reg[31:0] Y_lo;
20     output reg C,V,N,Z;
21
22     ////////////////////////////////////////////////////
23     // Barrel Shifter allows to shift more than one bit to the
24     // right while still maintaining the sign bit.
25     ////////////////////////////////////////////////////
26     always @(*)begin
27         C = T[0];    N = Y_lo[31];    V =1'bx ;
28         case(shamt) // shift amount
29             5'd1: Y_lo = {{1{T[31]}},T[31:1]}; // shift right 1-bit
30             5'd2: Y_lo = {{2{T[31]}},T[31:2]}; // shift right 2-bit
31             5'd3: Y_lo = {{3{T[31]}},T[31:3]}; // shift right 3-bit
32             5'd4: Y_lo = {{4{T[31]}},T[31:4]}; // shift right 4-bit
33             5'd5: Y_lo = {{5{T[31]}},T[31:5]}; // shift right 5-bit
34             5'd6: Y_lo = {{6{T[31]}},T[31:6]}; // shift right 6-bit
35             5'd7: Y_lo = {{7{T[31]}},T[31:7]}; // shift right 7-bit
36             5'd8: Y_lo = {{8{T[31]}},T[31:8]}; // shift right 8-bit
37             5'd9: Y_lo = {{9{T[31]}},T[31:9]}; // shift right 9-bit
38             5'd10: Y_lo = {{10{T[31]}},T[31:10]}; // shift right 10-bit
39             5'd11: Y_lo = {{11{T[31]}},T[31:11]}; // shift right 11-bit
40             5'd12: Y_lo = {{12{T[31]}},T[31:12]}; // shift right 12-bit
41             5'd13: Y_lo = {{13{T[31]}},T[31:13]}; // shift right 13-bit
42             5'd14: Y_lo = {{14{T[31]}},T[31:14]}; // shift right 14-bit
43             5'd15: Y_lo = {{15{T[31]}},T[31:15]}; // shift right 15-bit
44             5'd16: Y_lo = {{16{T[31]}},T[31:16]}; // shift right 16-bit
45             5'd17: Y_lo = {{17{T[31]}},T[31:17]}; // shift right 17-bit
46             5'd18: Y_lo = {{18{T[31]}},T[31:18]}; // shift right 18-bit
47             5'd19: Y_lo = {{19{T[31]}},T[31:19]}; // shift right 19-bit
48             5'd20: Y_lo = {{20{T[31]}},T[31:20]}; // shift right 20-bit
49             5'd21: Y_lo = {{21{T[31]}},T[31:21]}; // shift right 21-bit
50             5'd22: Y_lo = {{22{T[31]}},T[31:22]}; // shift right 22-bit
51             5'd23: Y_lo = {{23{T[31]}},T[31:23]}; // shift right 23-bit
52             5'd24: Y_lo = {{24{T[31]}},T[31:24]}; // shift right 24-bit
53             5'd25: Y_lo = {{25{T[31]}},T[31:25]}; // shift right 25-bit
54             5'd26: Y_lo = {{26{T[31]}},T[31:26]}; // shift right 26-bit
55             5'd27: Y_lo = {{27{T[31]}},T[31:27]}; // shift right 27-bit
56             5'd28: Y_lo = {{28{T[31]}},T[31:28]}; // shift right 28-bit
57             5'd29: Y_lo = {{29{T[31]}},T[31:29]}; // shift right 29-bit
```



```
58         5'd30: Y_lo = {{30{T[31]}},T[31:30]}; // shift right 30-bit
59         5'd31: Y_lo = {{31{T[31]}},T[31]};  // shift right 31-bit
60         5'd0: Y_lo = T; // dont shift
61         default: Y_lo = T;
62     endcase
63 end
64 endmodule
65
```

```

1  `timescale 1ns / 1ps
2  /***** C E C S 4 4 0 *****/
3  *
4  * File Name:  MIPS_32.v
5  * Project:    Lab_Assignment_1
6  * Designer:   Javier Garcia
7  * Email:      javis9526@yahoo.com
8  * Rev. No.:   Version 1.8
9  * Rev. Date:  9-12-18
10 *
11 * Purpose: Perform the basic operations for the ALU. Here simple arithmetic
12 *          functions are performed as well as logic, and shifts. The respective
13 *          flags for each function are also set.
14 * Notes: All operations here are signed except for ADDU, SUBU, STLU.
15 *
16 *****/
17 module MIPS_32( S,T,FS,
18                 Y_lo,
19                 {C,V,N,Z});
20
21     input  [31:0]  S,T;
22     input   [9:0]   FS;
23     output          N,Z,V,C;
24     output [31:0]  Y_lo;
25
26     reg          N,Z,V,C;
27     reg [31:0]  Y_lo;
28
29     integer ts,tt;
30
31
32
33     always @(S,T,FS) begin
34         {N,Z,V,C} = 4'b0;
35         {ts,tt} = {S,T};
36         casex(FS[4:0])
37             5'h00: begin//*****Pass S*****
38                 Y_lo = S; C = 1'bx;      N = Y_lo[31];      V = 1'bx;
39             end
40             5'h01: begin //*****Pass T*****
41                 Y_lo = T; C = 1'bx;      N = Y_lo[31];      V = 1'bx;
42             end
43             5'h02: begin//*****ADD*****
44                 {C,Y_lo} = S+T;      N = Y_lo[31];      V = (S[31] == Y_lo[31])?0:1;
45             end
46             5'h03: begin //*****SUB*****
47                 {C,Y_lo} = S-T;      N = Y_lo[31];      V = (S[31] == Y_lo[31])?0:1;
48             end
49             5'h04: begin //*****ADDU*****
50                 {C,Y_lo} = S+T;      N = 0;      V = (S[31] == Y_lo[31])?0:1;
51             end
52             5'h05: begin //*****SUBU*****
53                 {C,Y_lo} = S-T;      N = 0;      V = (S[31] == Y_lo[31])?0:1;
54             end
55             5'h06: begin //*****SLT*****
56                 Y_lo = (ts<tt)? 1: 0; C = 1'bx;      N = Y_lo[31];      V = 1'bx;
57             end

```

```

58      5'h07: begin //*****SLTU*****
59          Y_lo = (S<T)? 1:0; C = 1'bx;          N = 0;          V = 1'bx ;
60      end
61      5'h08: begin //*****AND*****
62          Y_lo = S&T; C = 1'bx;          N = Y_lo[31];          V = 1'bx;
63      end
64      5'h09: begin //*****OR*****
65          Y_lo = S|T; C = 1'bx;          N = Y_lo[31];          V = 1'bx;
66      end
67      5'h0A: begin //*****XOR*****
68          Y_lo = S^T; C = 1'bx;          N = Y_lo[31];          V = 1'bx ;
69      end
70      5'h0B: begin //*****NOR*****
71          Y_lo = ~(S | T); C = 1'bx;          N = Y_lo[31];          V = 1'bx;
72      end
73      5'h0F: begin //*****INC*****
74          {C,Y_lo} = S + 1;          N = Y_lo[31];          V = (S[31] == Y_lo[31])?0:1;
75      end
76      5'h10: begin //*****DEC*****
77          {C,Y_lo} = S-1;          N = Y_lo[31];          V = (S[31] == Y_lo[31])?0:1;
78      end
79      5'h11: begin //*****INC4*****
80          {C,Y_lo} = S+4;          N = Y_lo[31];          V = (S[31] == Y_lo[31])?0:1;
81      end
82      5'h12: begin //*****DEC4*****
83          {C,Y_lo} = S-4;          N = Y_lo[31];          V = (S[31] == Y_lo[31])?0:1;
84      end
85      5'h13: begin //*****ZEROS*****
86          Y_lo = 32'h0; C = 1'bx;          N = Y_lo[31];          V = 1'bx ;
87      end
88      5'h14: begin //*****ONES*****
89          Y_lo = 32'hFFFFFFF;          C = 1'bx;          N = Y_lo[31];          V = 1'bx ;
90      end
91      5'h15: begin //*****SP_INIT*****
92          Y_lo = 32'h3FC;          C = 1'bx;          N = Y_lo[31];          V = 1'bx;
93      end
94      5'h16: begin //*****ANDI*****
95          Y_lo = S&{16'h0,T[15:0]}; C = 1'bx;          N = Y_lo[31];          V = 1'bx;
96      end
97      5'h17: begin //*****ORI*****
98          Y_lo = S|{16'h0,T[15:0]}; C = 1'bx; N = Y_lo[31];          V = 1'bx;
99      end
100     5'h18: begin //*****LUI*****
101         Y_lo = {T[15:0],16'h0}; C = 1'bx; N = Y_lo[31]; V = 1'bx;
102     end
103     5'h19: begin //*****XORI*****
104         Y_lo = S^{16'h0,T[15:0]}; C = 1'bx; N = Y_lo[31];          V = 1'bx;
105     end
106
107     default:begin //*****Default*****
108         Y_lo = Y_lo;          C=C;          N = Y_lo[31];          V = V;
109     end
110 endcase
111 // and the output with zero and store flag
112 Z = !(32'hFFFFFFF & Y_lo);
113 end
114

```

```
115     endmodule
116
```

```

1  `timescale 1ns / 1ps
2  /*****
3  * Authors:   Javier Garcia
4  *           Taylor Cochran
5  *
6  * Emails:    javis9526@yahoo.com
7  *           TayCochran123@gmail.com
8  *
9  * Filename:  PipeLine_Stage_4.v
10 * Date:      November 26, 2018
11 * Version:   1.3 More comments and cleaner spacing, also added the Stall Fix
12 *
13 *                                                    (End of Page Width)
14 *****/
15 module PipeLine_Stage_4(clk,rst,IE,IDEX_PC,
16                         EXMEM_M,EXMEM_WB,
17                         EXMEM_Baddr,EXMEM_Jaddr,
18                         EXMEM_FLAGS,
19                         EXMEM_ALU, EXMEM_MData,EXMEM_Waddr,M_Data,io_out,
20                         MEMWB_WB,
21                         MEMWB_MData,MEMWB_IO,MEMWB_ALU,MEMWB_Waddr,PC_out,
22                         Branch_s,pc4_mux);
23
24     input  clk,rst,IE;
25     input  [31:0]  IDEX_PC;
26     input   [15:0]  EXMEM_M;
27     input   [3:0]   EXMEM_WB;
28     input  [31:0]  EXMEM_Baddr,EXMEM_Jaddr;
29     input   [3:0]   EXMEM_FLAGS;
30     input  [4:0]   EXMEM_Waddr;
31     input  [31:0]  EXMEM_ALU,EXMEM_MData,M_Data,io_out;
32
33
34     output reg  [3:0]  MEMWB_WB;
35     output reg  [31:0] MEMWB_MData,MEMWB_ALU,MEMWB_IO;
36     output reg  [4:0]  MEMWB_Waddr;
37     output      [31:0] PC_out,pc4_mux;
38     output      Branch_s;
39
40     reg  [4:0]  t_Flags;
41     wire [31:0] flag_mux,pc4_mux;
42     wire [31:0] PC_out,b_mux,j_mux;
43     wire      Branch_s;
44
45     ////////////////////////////////////////////////////
46     // Branch logic that raises a branch signal whenever a branch is to occur.
47     // This signal notifies the pc_cu that it needs to load the new PC address.
48     ////////////////////////////////////////////////////
49     //Branch Equal flag and zero flag are high
50     assign Branch_s = (EXMEM_M[0] & EXMEM_FLAGS[0])? 1'b1: //branch
51                       //Branch Not Equal Falg and Not zero flag
52                       (EXMEM_M[1] & ~EXMEM_FLAGS[0])? 1'b1: // branch
53                       //B Less/equal zero flag and the or of zero, negative
54                       (EXMEM_M[2] & (EXMEM_FLAGS[1] | EXMEM_FLAGS[0]))? 1'b1:
55                       //B greater than zero flag and not negative not zero
56                       (EXMEM_M[3] & ~EXMEM_FLAGS[1] & ~EXMEM_FLAGS[0])? 1'b1:
57                       0;//dont branch

```

```

58
59
60 //synchronous register that takes in the interrupt enable flag and the alu flags
61 always @(posedge clk, posedge rst)begin
62     if(rst) t_Flags = 5'b0;else
63         t_Flags = {IE,EXMEM_FLAGS};
64 end
65
66 // when flags select pass the flass, else pass the register data
67 assign flag_mux = (EXMEM_M[13])? t_Flags:EXMEM_MData;
68 // when storing pc pass the pc else pass the above result
69 assign pc4_mux = (EXMEM_M[14])? IDEX_PC: flag_mux;
70
71 ////////////////////////////////////////////////////
72 // The below muxes take care of passing the correct pc to stage 1.
73 // These values are based on the jump and branch instructions.
74 ////////////////////////////////////////////////////
75
76 // when a branch signal pass the branch addr. else pas the jump reg(ALU)
77 assign b_mux = (Branch_s)? EXMEM_Baddr: EXMEM_ALU;
78 // when the jump flag is high pass the jump addr. else pass above result
79 assign j_mux = (EXMEM_M[10])? EXMEM_Jaddr:b_mux;
80 // when entering/leaving ISR get the PC from the stack else from above result
81 assign PC_out = (EXMEM_M[15] | EXMEM_M[12])? M_Data:j_mux;
82
83
84
85 ////////////////////////////////////////////////////
86 // Pipeline Registers are synchronously loaded.
87 ////////////////////////////////////////////////////
88 always @(posedge clk,posedge rst)begin
89     if(rst)begin
90         MEMWB_WB =4'b0;
91         MEMWB_MData = 32'b0;
92         MEMWB_IO= 32'b0;
93         MEMWB_ALU = 32'b0;
94         MEMWB_Waddr = 5'b0;
95     end
96     else begin
97         MEMWB_WB = EXMEM_WB;    //Write Back control
98         MEMWB_MData = M_Data;    // Memory module output
99         MEMWB_IO = io_out;    // IO module output
100        MEMWB_ALU = EXMEM_ALU;    // ALU output
101        MEMWB_Waddr = EXMEM_Waddr;// Write Address
102    end
103 end
104 endmodule
105

```

```

1  `timescale 1ns / 1ps
2  /*****
3  * Authors:   Javier Garcia
4  *           Taylor Cochran
5  *
6  * Emails:    javis9526@yahoo.com
7  *           TayCochran123@gmail.com
8  *
9  * Filename:  PipeLine_Stage_5.v
10 * Date:      November 25, 2018
11 * Version:   1.4 renamed wires to match diagram
12 *
13 *                                                    (End of Page Width)
14 *****/
15 module PipeLine_Stage_5(clk,rst,MEMWB_WB,MEMWB_MData,MEMWB_IO,
16                        MEMWB_ALU,MEMWB_Waddr,WB_Data,WB_Addr,WB_D_EN);
17
18     input clk,rst;
19
20     input  [3:0]    MEMWB_WB;
21     input  [31:0]   MEMWB_MData,MEMWB_IO,MEMWB_ALU;
22     input  [4:0]    MEMWB_Waddr;
23
24     output wire [31:0] WB_Data;//Write Back Data
25     output wire [4:0]  WB_Addr;//Write Back Address
26     output wire        WB_D_EN; // Write Back Data Enable
27
28     ////////////////////////////////////////////
29     // When the Break flag is detected terminate the program.
30     ////////////////////////////////////////////
31     integer i;
32     always@(*) begin
33
34         if(MEMWB_WB[3])begin
35             $display("-----Register Dump-----");
36             for(i = 0;i<16; i = i+1)begin
37
38                 $display("t= %t   : R[%h] = %h   \t\t R[%h] = %h",$time,i[11:0],
39                         CPU_tb.uut.DP.PS2.RF.reg32[i[11:0]],i[11:0]+16,
40                         CPU_tb.uut.DP.PS2.RF.reg32[i[11:0]+16]);
41
42             end
43             $display("-----Memory Dump-----");
44             for(i = 192;i<255; i = i+4)begin
45
46                 $display("t= %t   : M[%h] = %h   ",$time,i[11:0],
47                         {CPU_tb.uut2.M[i[11:0]],CPU_tb.uut2.M[i[11:0]+1],
48                         CPU_tb.uut2.M[i[11:0]+2],CPU_tb.uut2.M[i[11:0]+3]});
49
50             end
51             //$display("t= %t   : M[%h] = %h   ",$time,12'h3f0,
52             //CPU_tb.uut3.Memory[12'h3f0]);
53             $finish;
54         end
55     end
56
57     //pass the data from IO

```

```
58     assign WB_Data = (MEMWB_WB[2:1] == 2'b10)? MEMWB_IO:
59                     // pass the data from memory module
60                     (MEMWB_WB[2:1] == 2'b01)? MEMWB_MData :
61                     MEMWB_ALU ; // pass ALU
62
63     assign WB_Addr = MEMWB_Waddr; // write back address
64
65     assign WB_D_EN = MEMWB_WB[0]; // Write Data Enable
66
67 endmodule
68
```


C.MEMORY MODULES - 3 Pages

```

1  `timescale 1ns / 1ps
2  /*****
3  * Authors:   Javier Garcia
4  *           Taylor Cochran
5  *
6  * Emails:    javis9526@yahoo.com
7  *           TayCochran123@gmail.com
8  *
9  * Filename:  IO_Memory.v
10 * Date:      November 24, 2018
11 * Version:   1.2
12 *
13 *                                                    (End of Page Width)
14 *****/
15 module IO_Memory(clk, rst, io_cs, io_wr, io_rd, Address, D_In, D_Out,
16                  intr_ack, io_intr);
17
18     //Inputs
19     input  clk, rst;
20
21     input io_cs, //Chip Select control signal.  Used to either read or write
22            io_wr, //      Write control signal.  MUST BE ON TO WRITE
23            io_rd; //      Read control signal.  MUST BE ON TO READ
24
25     input intr_ack; //Interrupt Acknowledge for handhsake
26
27     input [11:0] Address; //12-bit address field for D_Out
28     input [31:0] D_In;    //32-bit Data Input
29
30     //Outputs
31     output reg    io_intr; //Psuedo external interrupt for accessing ISR.
32
33     output [31:0] D_Out; //32-bit Output field for Data Path to read
34                        //If not being read, D_Out will be in Tri-State Hi-Z mode
35 //-----//
36 ///////////////////////////////////////////////////
37 // 4096x8 Big Endian Memory //
38 //-----//
39 // Effective 1024x32 Memory //
40 ///////////////////////////////////////////////////
41
42     reg [7:0] Memory [4095:0]; //Width: 7 bits & Depth: 4096 bits
43
44     //Synchronous Write
45     always@(posedge clk)
46         if(io_cs & io_wr) {Memory[Address + 12'h0],
47                             Memory[Address + 12'h1],
48                             Memory[Address + 12'h2],
49                             Memory[Address + 12'h3]} <= D_In;
50
51     //Asynchronous Read
52     assign D_Out = (io_cs & io_rd) ? {Memory[Address + 12'h0],
53                                         Memory[Address + 12'h1],
54                                         Memory[Address + 12'h2],
55                                         Memory[Address + 12'h3]} : 32'bZ;
56 //-----//
57 ///////////////////////////////////////////////////

```

```
58      //      Interrupt  Handshake      //
59      //////////////////////////////////////
60
61
62      initial begin
63          io_intr = 0;
64
65          #200 io_intr = 1;
66          end
67
68
69      always @(posedge intr_ack) begin
70          io_intr = 0;
71      end
72
73      //-----//
74  endmodule
75
```

```
1  `timescale 1ns / 1ps
2  /*****
3  * Designer:   Javier Garcia & Taylor Cochran
4  * Email:     javis9526@yahoo.com,TayCochran123@gmail.com
5  * Filename:  Data_Memory.v
6  * Date:     October 18, 2018
7  * Version:   1.1
8  *
9  * Notes: The Data Memory of the MIPS Processor has the ability to write synchronously
10 *         with the clock. Reading is Asynchronous. A 'Chip Select' must be asserted to
11 *         either READ or WRITE, along with the 'Read' and 'Write' controls.
12 *
13 *                                                     (End of Page Width)
14 *****/
15 module DataMemory(clk,
16                   Address, D_In,
17                   dm_cs,dm_wr,dm_rd,
18                   D_Out);
19
20     input          clk;
21     input  [31:0]  D_In;
22     input  [11:0]  Address;
23     input          dm_cs,dm_wr,dm_rd;
24
25     output [31:0]  D_Out;
26
27     reg  [7:0]     M  [4095:0]; // memory of width: 8, depth: 4096
28
29
30     ///////////////////////////////////////////////////
31     //Synchronous block that writes to the memory
32     ///////////////////////////////////////////////////
33     always@(posedge clk)
34         if(dm_wr & dm_cs) {M[Address],M[Address+1],M[Address+2],M[Address+3]} = D_In;
35
36     ///////////////////////////////////////////////////
37     //Asynchronous block that reads from the memory
38     ///////////////////////////////////////////////////
39     assign D_Out =(dm_rd&dm_cs)? {M[Address],M[Address+1],M[Address+2],M[Address+3]}:
40                                     32'bz;
41
42 endmodule
43
```

D. ISIM Log Files - 32 Pages

iM 01

```

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

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

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

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

00 00 78 20 // yes_ne:     add  $15, $0, $0          # CLR   R15  "pass flag"
3c 0d 10 01 //            lui  $13, 0x1001
35 ad 00 c0 //            ori  $13, 0x00C0          # LI    R13,  0x100100C0
ad a1 00 00 //            sw   $01, 0($13)          # ST    [R13], R01
00 00 00 0d //            break

```

Finished circuit initialization process.

```

-----Register Dump-----
t= 245000.00ps : R[000] = 00000000      R[010] = xxxxxxxx
t= 245000.00ps : R[001] = 12345678      R[011] = xxxxxxxx
t= 245000.00ps : R[002] = 87654321      R[012] = xxxxxxxx
t= 245000.00ps : R[003] = 12345678      R[013] = xxxxxxxx
t= 245000.00ps : R[004] = xxxxxxxx      R[014] = xxxxxxxx
t= 245000.00ps : R[005] = xxxxxxxx      R[015] = xxxxxxxx
t= 245000.00ps : R[006] = xxxxxxxx      R[016] = xxxxxxxx
t= 245000.00ps : R[007] = xxxxxxxx      R[017] = xxxxxxxx
t= 245000.00ps : R[008] = xxxxxxxx      R[018] = xxxxxxxx
t= 245000.00ps : R[009] = xxxxxxxx      R[019] = xxxxxxxx
t= 245000.00ps : R[00a] = xxxxxxxx      R[01a] = xxxxxxxx
t= 245000.00ps : R[00b] = xxxxxxxx      R[01b] = xxxxxxxx
t= 245000.00ps : R[00c] = xxxxxxxx      R[01c] = xxxxxxxx
t= 245000.00ps : R[00d] = 100100c0      R[01d] = 000003fc
t= 245000.00ps : R[00e] = 00000000      R[01e] = xxxxxxxx
t= 245000.00ps : R[00f] = 00000000      R[01f] = xxxxxxxx
-----Memory Dump-----
t= 245000.00ps : M[0c0] = 12345678
t= 245000.00ps : M[0c4] = xxxxxxxx
t= 245000.00ps : M[0c8] = xxxxxxxx
t= 245000.00ps : M[0cc] = xxxxxxxx

```

```
t= 245000.00ps : M[0d0] = xxxxxxxx
t= 245000.00ps : M[0d4] = xxxxxxxx
t= 245000.00ps : M[0d8] = xxxxxxxx
t= 245000.00ps : M[0dc] = xxxxxxxx
t= 245000.00ps : M[0e0] = xxxxxxxx
t= 245000.00ps : M[0e4] = xxxxxxxx
t= 245000.00ps : M[0e8] = xxxxxxxx
t= 245000.00ps : M[0ec] = xxxxxxxx
t= 245000.00ps : M[0f0] = xxxxxxxx
t= 245000.00ps : M[0f4] = xxxxxxxx
t= 245000.00ps : M[0f8] = xxxxxxxx
t= 245000.00ps : M[0fc] = xxxxxxxx
```

Stopped at time : 245 ns : File

"C:/Users/javis/OneDrive/Documents/School/CECS

440/i_prefer_The_Real_Pipeline_14/PipeLine_Stage_5.v" Line 54

```

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

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

3c 0e 5a 5a // exit: lui $14, 0x5A5A
35 ce 3c 3c //      ori $14, 0x3C3C      # LI    R14,  0x5A5A3C3C
00 00 00 0d //      break

```

Finished circuit initialization process.

run 1.00us

```

-----Register Dump-----
t= 1405000.00ps : R[000] = 00000000      R[010] = xxxxxxxx
t= 1405000.00ps : R[001] = 0000ffff      R[011] = xxxxxxxx
t= 1405000.00ps : R[002] = 00000000      R[012] = xxxxxxxx
t= 1405000.00ps : R[003] = xxxxxxxx      R[013] = xxxxxxxx
t= 1405000.00ps : R[004] = xxxxxxxx      R[014] = xxxxxxxx
t= 1405000.00ps : R[005] = xxxxxxxx      R[015] = xxxxxxxx
t= 1405000.00ps : R[006] = xxxxxxxx      R[016] = xxxxxxxx
t= 1405000.00ps : R[007] = xxxxxxxx      R[017] = xxxxxxxx
t= 1405000.00ps : R[008] = xxxxxxxx      R[018] = xxxxxxxx
t= 1405000.00ps : R[009] = xxxxxxxx      R[019] = xxxxxxxx
t= 1405000.00ps : R[00a] = xxxxxxxx      R[01a] = xxxxxxxx
t= 1405000.00ps : R[00b] = xxxxxxxx      R[01b] = xxxxxxxx
t= 1405000.00ps : R[00c] = xxxxxxxx      R[01c] = xxxxxxxx
t= 1405000.00ps : R[00d] = xxxxxxxx      R[01d] = 000003fc
t= 1405000.00ps : R[00e] = 5a5a3c3c      R[01e] = xxxxxxxx
t= 1405000.00ps : R[00f] = 10010100      R[01f] = xxxxxxxx

```

```

-----Memory Dump-----

```

```

t= 1405000.00ps : M[0c0] = 7fffffff
t= 1405000.00ps : M[0c4] = 3fffffff
t= 1405000.00ps : M[0c8] = 1fffffff
t= 1405000.00ps : M[0cc] = 0fffffff
t= 1405000.00ps : M[0d0] = 07fffffff
t= 1405000.00ps : M[0d4] = 03fffffff
t= 1405000.00ps : M[0d8] = 01fffffff
t= 1405000.00ps : M[0dc] = 00fffffff
t= 1405000.00ps : M[0e0] = 007fffffff
t= 1405000.00ps : M[0e4] = 003fffffff

```



```
t= 1405000.00ps : M[0e8] = 001ffffff
t= 1405000.00ps : M[0ec] = 000ffffff
t= 1405000.00ps : M[0f0] = 0007ffff
t= 1405000.00ps : M[0f4] = 0003ffff
t= 1405000.00ps : M[0f8] = 0001ffff
t= 1405000.00ps : M[0fc] = 0000ffff
Stopped at time : 1405 ns : File
"C:/Users/javis/OneDrive/Documents/School/CECS
440/i_prefer_The_Real_Pipeline_14/PipeLine_Stage_5.v" Line 54
```

```

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

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

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

3c 0e 5a 5a // exit:      lui  $14, 0x5A5A
35 ce 3c 3c //            ori  $14, 0x3C3C      # LI    R14,  0x5A5A3C3C
00 00 00 0d //            break

```

Finished circuit initialization process.

run 1.00us

```

-----Register Dump-----
t= 1405000.00ps : R[000] = 00000000      R[010] = xxxxxxxx
t= 1405000.00ps : R[001] = ffff8000      R[011] = xxxxxxxx
t= 1405000.00ps : R[002] = 00000000      R[012] = xxxxxxxx
t= 1405000.00ps : R[003] = xxxxxxxx      R[013] = xxxxxxxx
t= 1405000.00ps : R[004] = xxxxxxxx      R[014] = xxxxxxxx
t= 1405000.00ps : R[005] = xxxxxxxx      R[015] = xxxxxxxx
t= 1405000.00ps : R[006] = xxxxxxxx      R[016] = xxxxxxxx
t= 1405000.00ps : R[007] = xxxxxxxx      R[017] = xxxxxxxx
t= 1405000.00ps : R[008] = xxxxxxxx      R[018] = xxxxxxxx
t= 1405000.00ps : R[009] = xxxxxxxx      R[019] = xxxxxxxx
t= 1405000.00ps : R[00a] = xxxxxxxx      R[01a] = xxxxxxxx
t= 1405000.00ps : R[00b] = xxxxxxxx      R[01b] = xxxxxxxx
t= 1405000.00ps : R[00c] = xxxxxxxx      R[01c] = xxxxxxxx
t= 1405000.00ps : R[00d] = xxxxxxxx      R[01d] = 000003fc
t= 1405000.00ps : R[00e] = 5a5a3c3c      R[01e] = xxxxxxxx
t= 1405000.00ps : R[00f] = 10010100      R[01f] = xxxxxxxx

```

```

-----Memory Dump-----
t= 1405000.00ps : M[0c0] = c0007fff
t= 1405000.00ps : M[0c4] = e0003fff
t= 1405000.00ps : M[0c8] = f0001fff
t= 1405000.00ps : M[0cc] = f8000fff
t= 1405000.00ps : M[0d0] = fc0007ff
t= 1405000.00ps : M[0d4] = fe0003ff
t= 1405000.00ps : M[0d8] = ff0001ff

```

```
t= 1405000.00ps : M[0dc] = ff8000ff
t= 1405000.00ps : M[0e0] = ffc0007f
t= 1405000.00ps : M[0e4] = ffe0003f
t= 1405000.00ps : M[0e8] = fff0001f
t= 1405000.00ps : M[0ec] = fff8000f
t= 1405000.00ps : M[0f0] = fffc0007
t= 1405000.00ps : M[0f4] = fffe0003
t= 1405000.00ps : M[0f8] = ffff0001
t= 1405000.00ps : M[0fc] = ffff8000
```

Stopped at time : 1405 ns : File

"C:/Users/javis/OneDrive/Documents/School/CECS

440/i_prefer_The_Real_Pipeline_14/PipeLine_Stage_5.v" Line 54

iM 04

```

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

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

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

3c 0e 5a 5a // exit:      lui  $14, 0x5A5A
35 ce 3c 3c //            ori  $14, 0x3C3C      # LI    R14,  0x5A5A3C3C
00 00 00 0d //            break

```

Finished circuit initialization process.

run 1.00us

```

-----Register Dump-----
t= 1565000.00ps : R[000] = 00000000      R[010] = xxxxxxxx
t= 1565000.00ps : R[001] = ffff0000      R[011] = xxxxxxxx
t= 1565000.00ps : R[002] = 00000000      R[012] = xxxxxxxx
t= 1565000.00ps : R[003] = 00000000      R[013] = xxxxxxxx
t= 1565000.00ps : R[004] = xxxxxxxx      R[014] = xxxxxxxx
t= 1565000.00ps : R[005] = xxxxxxxx      R[015] = xxxxxxxx
t= 1565000.00ps : R[006] = xxxxxxxx      R[016] = xxxxxxxx
t= 1565000.00ps : R[007] = xxxxxxxx      R[017] = xxxxxxxx
t= 1565000.00ps : R[008] = xxxxxxxx      R[018] = xxxxxxxx
t= 1565000.00ps : R[009] = xxxxxxxx      R[019] = xxxxxxxx
t= 1565000.00ps : R[00a] = xxxxxxxx      R[01a] = xxxxxxxx
t= 1565000.00ps : R[00b] = xxxxxxxx      R[01b] = xxxxxxxx
t= 1565000.00ps : R[00c] = xxxxxxxx      R[01c] = xxxxxxxx
t= 1565000.00ps : R[00d] = xxxxxxxx      R[01d] = 000003fc
t= 1565000.00ps : R[00e] = 5a5a3c3c      R[01e] = xxxxxxxx
t= 1565000.00ps : R[00f] = 10010100      R[01f] = xxxxxxxx

-----Memory Dump-----
t= 1565000.00ps : M[0c0] = ffffffff
t= 1565000.00ps : M[0c4] = ffffffff
t= 1565000.00ps : M[0c8] = ffffffff
t= 1565000.00ps : M[0cc] = ffffffff
t= 1565000.00ps : M[0d0] = ffffffff

```

```
t= 1565000.00ps : M[0d4] = ffffffff0
t= 1565000.00ps : M[0d8] = ffffffff80
t= 1565000.00ps : M[0dc] = ffffffff00
t= 1565000.00ps : M[0e0] = fffffffe00
t= 1565000.00ps : M[0e4] = fffffffc00
t= 1565000.00ps : M[0e8] = fffffff800
t= 1565000.00ps : M[0ec] = fffffff000
t= 1565000.00ps : M[0f0] = fffffe000
t= 1565000.00ps : M[0f4] = fffffc000
t= 1565000.00ps : M[0f8] = fffff8000
t= 1565000.00ps : M[0fc] = fffff0000
Stopped at time : 1565 ns : File
"C:/Users/javis/OneDrive/Documents/School/CECS
440/i_prefer_The_Real_Pipeline_14/PipeLine_Stage_5.v" Line 54
```

```

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

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

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

3c 0e 5a 5a // exit:      lui  $14, 0x5A5A
35 ce 3c 3c //            ori  $14, 0x3C3C      # LI    R14,  0x5A5A3C3C
00 00 00 0d //            break

```

Finished circuit initialization process.

run 1.00us

```

-----Register Dump-----
t= 1565000.00ps : R[000] = 00000000      R[010] = xxxxxxxx
t= 1565000.00ps : R[001] = ffff0000      R[011] = xxxxxxxx
t= 1565000.00ps : R[002] = 00000000      R[012] = xxxxxxxx
t= 1565000.00ps : R[003] = 00000000      R[013] = xxxxxxxx
t= 1565000.00ps : R[004] = xxxxxxxx      R[014] = xxxxxxxx
t= 1565000.00ps : R[005] = xxxxxxxx      R[015] = xxxxxxxx
t= 1565000.00ps : R[006] = xxxxxxxx      R[016] = xxxxxxxx
t= 1565000.00ps : R[007] = xxxxxxxx      R[017] = xxxxxxxx
t= 1565000.00ps : R[008] = xxxxxxxx      R[018] = xxxxxxxx
t= 1565000.00ps : R[009] = xxxxxxxx      R[019] = xxxxxxxx
t= 1565000.00ps : R[00a] = xxxxxxxx      R[01a] = xxxxxxxx
t= 1565000.00ps : R[00b] = xxxxxxxx      R[01b] = xxxxxxxx
t= 1565000.00ps : R[00c] = xxxxxxxx      R[01c] = xxxxxxxx
t= 1565000.00ps : R[00d] = xxxxxxxx      R[01d] = 000003fc
t= 1565000.00ps : R[00e] = 5a5a3c3c      R[01e] = xxxxxxxx
t= 1565000.00ps : R[00f] = 10010100      R[01f] = xxxxxxxx

-----Memory Dump-----
t= 1565000.00ps : M[0c0] = ffffffff
t= 1565000.00ps : M[0c4] = ffffffff
t= 1565000.00ps : M[0c8] = ffffffff
t= 1565000.00ps : M[0cc] = ffffffff
t= 1565000.00ps : M[0d0] = ffffffff

```

```
t= 1565000.00ps : M[0d4] = ffffffff0
t= 1565000.00ps : M[0d8] = ffffffff80
t= 1565000.00ps : M[0dc] = ffffffff00
t= 1565000.00ps : M[0e0] = fffffffe00
t= 1565000.00ps : M[0e4] = fffffffc00
t= 1565000.00ps : M[0e8] = fffffff800
t= 1565000.00ps : M[0ec] = ffffff000
t= 1565000.00ps : M[0f0] = fffffe000
t= 1565000.00ps : M[0f4] = fffffc000
t= 1565000.00ps : M[0f8] = fffff8000
t= 1565000.00ps : M[0fc] = fffff0000
Stopped at time : 1565 ns : File
"C:/Users/javis/OneDrive/Documents/School/CECS
440/i_prefer_The_Real_Pipeline_14/PipeLine_Stage_5.v" Line 5
```

```

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

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

```

Simulator is doing circuit initialization process.

Finished circuit initialization process.

run 1.00us

```

-----Register Dump-----
t= 1785000.00ps : R[000] = 00000000      R[010] = xxxxxxxx
t= 1785000.00ps : R[001] = 12345678      R[011] = 000075cc
t= 1785000.00ps : R[002] = 89abcdef      R[012] = xxxxxxxx
t= 1785000.00ps : R[003] = a5a5a5a5      R[013] = xxxxxxxx
t= 1785000.00ps : R[004] = 5a5a5a5a      R[014] = xxxxxxxx
t= 1785000.00ps : R[005] = 2468ace0      R[015] = xxxxxxxx
t= 1785000.00ps : R[006] = 13579bdf      R[016] = xxxxxxxx
t= 1785000.00ps : R[007] = 0f0f0f0f      R[017] = xxxxxxxx
t= 1785000.00ps : R[008] = f0f0f0f0      R[018] = xxxxxxxx
t= 1785000.00ps : R[009] = 00000009      R[019] = xxxxxxxx
t= 1785000.00ps : R[00a] = 0000000a      R[01a] = xxxxxxxx
t= 1785000.00ps : R[00b] = 0000000b      R[01b] = xxxxxxxx
t= 1785000.00ps : R[00c] = 0000000c      R[01c] = xxxxxxxx
t= 1785000.00ps : R[00d] = 00000000      R[01d] = 000003fc
t= 1785000.00ps : R[00e] = 10010100      R[01e] = xxxxxxxx
t= 1785000.00ps : R[00f] = 10010040      R[01f] = xxxxxxxx

```



```
-----Memory Dump-----
t= 1785000.00ps : M[0c0] = c3c3c3c3
t= 1785000.00ps : M[0c4] = 12345678
t= 1785000.00ps : M[0c8] = 89abcdef
t= 1785000.00ps : M[0cc] = a5a5a5a5
t= 1785000.00ps : M[0d0] = 5a5a5a5a
t= 1785000.00ps : M[0d4] = 2468ace0
t= 1785000.00ps : M[0d8] = 13579bdf
t= 1785000.00ps : M[0dc] = 0f0f0f0f
t= 1785000.00ps : M[0e0] = f0f0f0f0
t= 1785000.00ps : M[0e4] = 00000009
t= 1785000.00ps : M[0e8] = 0000000a
t= 1785000.00ps : M[0ec] = 0000000b
t= 1785000.00ps : M[0f0] = 0000000c
t= 1785000.00ps : M[0f4] = 0000000d
t= 1785000.00ps : M[0f8] = ffffffff8
t= 1785000.00ps : M[0fc] = 000075cc
Stopped at time : 1785 ns : File
"C:/Users/javis/OneDrive/Documents/School/CECS
440/i_prefer_The_Real_Pipeline_14/PipeLine_Stage_5.v" Line 54
```

```

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

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

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

```

Simulator is doing circuit initialization process.

Finished circuit initialization process.

run 1.00us

```

-----Register Dump-----
t= 1885000.00ps : R[000] = 00000000      R[010] = xxxxxxxx
t= 1885000.00ps : R[001] = 12345678      R[011] = 000075cc
t= 1885000.00ps : R[002] = 89abcdef      R[012] = xxxxxxxx
t= 1885000.00ps : R[003] = a5a5a5a5      R[013] = xxxxxxxx
t= 1885000.00ps : R[004] = 5a5a5a5a      R[014] = xxxxxxxx
t= 1885000.00ps : R[005] = 2468ace0      R[015] = xxxxxxxx
t= 1885000.00ps : R[006] = 13579bdf      R[016] = xxxxxxxx
t= 1885000.00ps : R[007] = 0f0f0f0f      R[017] = xxxxxxxx
t= 1885000.00ps : R[008] = f0f0f0f0      R[018] = xxxxxxxx

```

t= 1885000.00ps	:	R[009]	=	00000009		R[019]	=	xxxxxxxx
t= 1885000.00ps	:	R[00a]	=	0000000a		R[01a]	=	xxxxxxxx
t= 1885000.00ps	:	R[00b]	=	0000000b		R[01b]	=	xxxxxxxx
t= 1885000.00ps	:	R[00c]	=	0000000c		R[01c]	=	xxxxxxxx
t= 1885000.00ps	:	R[00d]	=	00000000		R[01d]	=	000003fc
t= 1885000.00ps	:	R[00e]	=	10010100		R[01e]	=	xxxxxxxx
t= 1885000.00ps	:	R[00f]	=	ffffffff		R[01f]	=	00000048

-----Memory Dump-----

t= 1885000.00ps	:	M[0c0]	=	c3c3c3c3
t= 1885000.00ps	:	M[0c4]	=	12345678
t= 1885000.00ps	:	M[0c8]	=	89abcdef
t= 1885000.00ps	:	M[0cc]	=	a5a5a5a5
t= 1885000.00ps	:	M[0d0]	=	5a5a5a5a
t= 1885000.00ps	:	M[0d4]	=	2468ace0
t= 1885000.00ps	:	M[0d8]	=	13579bdf
t= 1885000.00ps	:	M[0dc]	=	0f0f0f0f
t= 1885000.00ps	:	M[0e0]	=	f0f0f0f0
t= 1885000.00ps	:	M[0e4]	=	00000009
t= 1885000.00ps	:	M[0e8]	=	0000000a
t= 1885000.00ps	:	M[0ec]	=	0000000b
t= 1885000.00ps	:	M[0f0]	=	0000000c
t= 1885000.00ps	:	M[0f4]	=	0000000d
t= 1885000.00ps	:	M[0f8]	=	ffffffff8
t= 1885000.00ps	:	M[0fc]	=	000075cc

Stopped at time : 1885 ns : File

"C:/Users/javis/OneDrive/Documents/School/CECS

440/i_prefer_The_Real_Pipeline_14/PipeLine_Stage_5.v" Line 54

```

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

3c 0e 00 00 // pass:      lui  $14, 0x0000
35 ce 00 00 //            ori  $14, 0x0000      # $r14 <-- 0x00000000
(Pass flag)
00 00 00 0d //            break
3c 0e ff ff // fail1:      lui  $14, 0xFFFF
35 ce ff ff //            ori  $14, 0xFFFF      # $r14 <-- 0xFFFFFFFF
(Fail flag 1)
00 00 00 0d //            break
3c 0e ff ff // fail2L:      lui  $14, 0xFFFF
35 ce ff fe //            ori  $14, 0xFFFE      # $r14 <-- 0xFFFFFFFEE
(Fail flag 2L)
00 00 00 0d //            break
3c 0e ff ff // fail2H:      lui  $14, 0xFFFF
35 ce ff fd //            ori  $14, 0xFFFD      # $r14 <-- 0xFFFFFFFDD
(Fail flag 2H)
00 00 00 0d //            break
3c 0e ff ff // fail3L:      lui  $14, 0xFFFF
35 ce ff fc //            ori  $14, 0xFFFC      # $r14 <-- 0xFFFFFFFCC
(Fail flag 3L)
00 00 00 0d //            break
3c 0e ff ff // fail3H:      lui  $14, 0xFFFF
35 ce ff fb //            ori  $14, 0xFFFB      # $r14 <-- 0xFFFFFFFBB
(Fail flag 3H)
00 00 00 0d //            break
3c 0e ff ff // fail4:      lui  $14, 0xFFFF

```

```

35 ce ff fa //          ori $14, 0xFFFFA      # $r14 <-- 0xFFFFFFFF
(Fail flag 4)
00 00 00 0d //          break

```

Simulator is doing circuit initialization process.
Finished circuit initialization process.

```

-----Register Dump-----
t= 315000.00ps : R[000] = 00000000          R[010] = xxxxxxxx
t= 315000.00ps : R[001] = 00000019          R[011] = xxxxxxxx
t= 315000.00ps : R[002] = 000003e8          R[012] = xxxxxxxx
t= 315000.00ps : R[003] = ffffffff          R[013] = xxxxxxxx
t= 315000.00ps : R[004] = ffffffc18        R[014] = xxxxxxxx
t= 315000.00ps : R[005] = 000061a8          R[015] = xxxxxxxx
t= 315000.00ps : R[006] = ffff9e58          R[016] = xxxxxxxx
t= 315000.00ps : R[007] = ffffffff          R[017] = xxxxxxxx
t= 315000.00ps : R[008] = 000061a8          R[018] = xxxxxxxx
t= 315000.00ps : R[009] = ffff9e58          R[019] = xxxxxxxx
t= 315000.00ps : R[00a] = ffffffff          R[01a] = xxxxxxxx
t= 315000.00ps : R[00b] = ffff9e58          R[01b] = xxxxxxxx
t= 315000.00ps : R[00c] = ffffffff          R[01c] = xxxxxxxx
t= 315000.00ps : R[00d] = 000061a8          R[01d] = 000003fc
t= 315000.00ps : R[00e] = 00000000          R[01e] = xxxxxxxx
t= 315000.00ps : R[00f] = 10010000          R[01f] = xxxxxxxx

```

```

-----Memory Dump-----
t= 315000.00ps : M[0c0] = xxxxxxxx
t= 315000.00ps : M[0c4] = xxxxxxxx
t= 315000.00ps : M[0c8] = xxxxxxxx
t= 315000.00ps : M[0cc] = xxxxxxxx
t= 315000.00ps : M[0d0] = xxxxxxxx
t= 315000.00ps : M[0d4] = xxxxxxxx
t= 315000.00ps : M[0d8] = xxxxxxxx
t= 315000.00ps : M[0dc] = xxxxxxxx
t= 315000.00ps : M[0e0] = xxxxxxxx
t= 315000.00ps : M[0e4] = xxxxxxxx
t= 315000.00ps : M[0e8] = xxxxxxxx
t= 315000.00ps : M[0ec] = xxxxxxxx
t= 315000.00ps : M[0f0] = xxxxxxxx
t= 315000.00ps : M[0f4] = xxxxxxxx
t= 315000.00ps : M[0f8] = xxxxxxxx
t= 315000.00ps : M[0fc] = xxxxxxxx

```

Stopped at time : 315 ns : File

"C:/Users/javis/OneDrive/Documents/School/CECS

440/i_prefer_The_Real_Pipeline_14/PipeLine_Stage_5.v" Line 54

```

@0
3c 0f 10 01 // main:      lui  $15, 0x1001
35 ef 00 c0 //             ori  $15, 0x00C0      # $r15 <-- 0x100100C0
(dest pointer)
20 01 ff 8a //             addi $01, $00, -118      # $r01 <-- 0xFFFFFFFF8A
20 02 00 8a //             addi $02, $00, 138      # $r02 <-- 0x0000008A
0c 10 00 22 //             jal  slt_tests

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

01 ac 50 26 //             xor  $10, $13, $12      # $r10 <-- 0xFFFFFFFF
11 4b 00 02 //             beq  $10, $11, xor_pass
20 0e ff fb //             addi $14, $00, -5      # fail flag5 r14 <--
0xFFFF_FFFB
00 00 00 0d //             break
01 ac 48 24 // xor_pass:      and  $09, $13, $12      # $r09 <-- 0x00000000
11 20 00 02 //             beq  $09, $00, and_pass
20 0e ff fa //             addi $14, $00, -6      # fail flag6 r14 <--
0xFFFF_FFFA
00 00 00 0d //             break
01 e2 48 25 // and_pass:      or   $09, $15, $02      # $r09 <-- 0x100100CA
3c 08 10 01 //             lui  $08, 0x1001
35 08 00 ca //             ori  $08, 0x00CA      # $r08 <-- 0x100100CA
11 09 00 02 //             beq  $08, $09, or_pass
20 0e ff f9 //             addi $14, $00, -7      # fail flag7 r14 <--
0xFFFF_FFF9
00 00 00 0d //             break
01 e2 48 27 // or_pass:       nor  $09, $15, $02      # $r09 <-- 0xEFFFEFF35
3c 08 ef fe //             lui  $08, 0xEFFE
35 08 ff 35 //             ori  $08, 0xFF35      # $r08 <-- 0xEFFFEFF35
11 09 00 02 //             beq  $08, $09, nor_pass
20 0e ff f8 //             addi $14, $00, -8      # fail flag8 r14 <--
0xFFFF_FFF8
00 00 00 0d //             break
ad e8 00 10 // nor_pass:      sw   $08, 0x10($15)    # M[D0] <-- 0xEFFFEFF35
00 00 70 20 //             add  $14, $00, $00      # clear r14 indicating
"passed all"
00 00 00 0d //             break                  # should stop here, having
//                                     # completed all the

tests

00 22 18 2a // slt_tests:     slt  $03, $01, $02      # for signed# r01 < r02
14 60 00 02 //             bne  $03, $00, slt1      # thus, we should branch

```

```

20 0e ff ff //          addi $14, $00, -1          # fail flag1 r14 <--
FFFF_FFFF
00 00 00 0d //          break
20 04 00 c0 // slt1:    addi $04, $00, 0xC0        # pass flag1 M[C0] <-- C0
ad e4 00 00 //          sw    $04, 0x00($15)

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

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

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

```

Simulator is doing circuit initialization process.

Finished circuit initialization process.

```

-----Register Dump-----
t= 765000.00ps : R[000] = 00000000          R[010] = xxxxxxxx
t= 765000.00ps : R[001] = ffffffff8a        R[011] = xxxxxxxx
t= 765000.00ps : R[002] = 0000008a        R[012] = xxxxxxxx
t= 765000.00ps : R[003] = 00000000        R[013] = xxxxxxxx
t= 765000.00ps : R[004] = 000000c0        R[014] = xxxxxxxx
t= 765000.00ps : R[005] = 000000c4        R[015] = xxxxxxxx
t= 765000.00ps : R[006] = 000000c8        R[016] = xxxxxxxx
t= 765000.00ps : R[007] = 000000cc        R[017] = xxxxxxxx
t= 765000.00ps : R[008] = effeff35        R[018] = xxxxxxxx
t= 765000.00ps : R[009] = effeff35        R[019] = xxxxxxxx
t= 765000.00ps : R[00a] = ffffffff        R[01a] = xxxxxxxx
t= 765000.00ps : R[00b] = ffffffff        R[01b] = xxxxxxxx
t= 765000.00ps : R[00c] = 88778877        R[01c] = xxxxxxxx
t= 765000.00ps : R[00d] = 77887788        R[01d] = 000003fc
t= 765000.00ps : R[00e] = 00000000        R[01e] = xxxxxxxx
t= 765000.00ps : R[00f] = 100100c0        R[01f] = 00000014
-----Memory Dump-----
t= 765000.00ps : M[0c0] = 000000c0
t= 765000.00ps : M[0c4] = 000000c4
t= 765000.00ps : M[0c8] = 000000c8

```

```
t= 765000.00ps : M[0cc] = 000000cc
t= 765000.00ps : M[0d0] = effeff35
t= 765000.00ps : M[0d4] = xxxxxxxx
t= 765000.00ps : M[0d8] = xxxxxxxx
t= 765000.00ps : M[0dc] = xxxxxxxx
t= 765000.00ps : M[0e0] = xxxxxxxx
t= 765000.00ps : M[0e4] = xxxxxxxx
t= 765000.00ps : M[0e8] = xxxxxxxx
t= 765000.00ps : M[0ec] = xxxxxxxx
t= 765000.00ps : M[0f0] = xxxxxxxx
t= 765000.00ps : M[0f4] = xxxxxxxx
t= 765000.00ps : M[0f8] = xxxxxxxx
t= 765000.00ps : M[0fc] = xxxxxxxx
```

Stopped at time : 765 ns : File

"C:/Users/javis/OneDrive/Documents/School/CECS

440/i_prefer_The_Real_Pipeline_14/PipeLine_Stage_5.v" Line 54


```

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

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

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

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

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

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

```

```

3c 0e ff ff // fail1Q:    lui  $14, 0xFFFF
35 ce ff ff //           ori  $14, 0xFFFF          # $r14 <-- 0xFFFFFFFF
(Fail flag 1 Quot)
00 00 00 0d //           break
3c 0e ff ff // fail1R:    lui  $14, 0xFFFF
35 ce ff fe //           ori  $14, 0xFFFE          # $r14 <-- 0xFFFFFFFFE
(Fail flag 1 Rem)
00 00 00 0d //           break
3c 0e ff ff // fail2Q:    lui  $14, 0xFFFF
35 ce ff fd //           ori  $14, 0xFFFFD         # $r14 <-- 0xFFFFFFFDD
(Fail flag 2 Quot)
00 00 00 0d //           break
3c 0e ff ff // fail2R:    lui  $14, 0xFFFF
35 ce ff fc //           ori  $14, 0xFFFFC         # $r14 <-- 0xFFFFFFFCC
(Fail flag 2 Rem)
00 00 00 0d //           break
3c 0e ff ff // fail3Q:    lui  $14, 0xFFFF
35 ce ff fb //           ori  $14, 0xFFFFB         # $r14 <-- 0xFFFFFFFBB
(Fail flag 3 Quot)
00 00 00 0d //           break
3c 0e ff ff // fail3R:    lui  $14, 0xFFFF
35 ce ff fa //           ori  $14, 0xFFFFA         # $r14 <-- 0xFFFFFFFBA
(Fail flag 3 Rem)
00 00 00 0d //           break
3c 0e ff ff // fail4Q:    lui  $14, 0xFFFF
35 ce ff f9 //           ori  $14, 0xFFFF9         # $r14 <-- 0xFFFFFFF9
(Fail flag 4 Quot)
00 00 00 0d //           break
3c 0e ff ff // fail4R:    lui  $14, 0xFFFF
35 ce ff f8 //           ori  $14, 0xFFFF8         # $r14 <-- 0xFFFFFFF8
(Fail flag 4 Rem)

```

Simulator is doing circuit initialization process.
Finished circuit initialization process.

```

-----Register Dump-----
t= 395000.00ps : R[000] = 00000000          R[010] = xxxxxxxx
t= 395000.00ps : R[001] = 00040911          R[011] = xxxxxxxx
t= 395000.00ps : R[002] = 000003e8          R[012] = xxxxxxxx
t= 395000.00ps : R[003] = fffbf6ef          R[013] = xxxxxxxx
t= 395000.00ps : R[004] = fffffc18          R[014] = xxxxxxxx
t= 395000.00ps : R[005] = 00000108          R[015] = xxxxxxxx
t= 395000.00ps : R[006] = 000001d1          R[016] = xxxxxxxx
t= 395000.00ps : R[007] = fffffef8          R[017] = xxxxxxxx
t= 395000.00ps : R[008] = fffffe2f          R[018] = xxxxxxxx
t= 395000.00ps : R[009] = 00000108          R[019] = xxxxxxxx
t= 395000.00ps : R[00a] = fffffe2f          R[01a] = xxxxxxxx
t= 395000.00ps : R[00b] = 00000000          R[01b] = xxxxxxxx
t= 395000.00ps : R[00c] = 00000000          R[01c] = xxxxxxxx
t= 395000.00ps : R[00d] = 00000000          R[01d] = 000003fc
t= 395000.00ps : R[00e] = 00000000          R[01e] = xxxxxxxx
t= 395000.00ps : R[00f] = 10010000          R[01f] = xxxxxxxx
-----Memory Dump-----
t= 395000.00ps : M[0c0] = xxxxxxxx

```

```
t= 395000.00ps : M[0c4] = xxxxxxxx
t= 395000.00ps : M[0c8] = xxxxxxxx
t= 395000.00ps : M[0cc] = xxxxxxxx
t= 395000.00ps : M[0d0] = xxxxxxxx
t= 395000.00ps : M[0d4] = xxxxxxxx
t= 395000.00ps : M[0d8] = xxxxxxxx
t= 395000.00ps : M[0dc] = xxxxxxxx
t= 395000.00ps : M[0e0] = xxxxxxxx
t= 395000.00ps : M[0e4] = xxxxxxxx
t= 395000.00ps : M[0e8] = xxxxxxxx
t= 395000.00ps : M[0ec] = xxxxxxxx
t= 395000.00ps : M[0f0] = xxxxxxxx
t= 395000.00ps : M[0f4] = xxxxxxxx
t= 395000.00ps : M[0f8] = xxxxxxxx
t= 395000.00ps : M[0fc] = xxxxxxxx
```

Stopped at time : 395 ns : File

"C:/Users/javis/OneDrive/Documents/School/CECS

440/i_prefer_The_Real_Pipeline_14/PipeLine_Stage_5.v" Line 54

```

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

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

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

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

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

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

```

```

3c 0e ff ff // fail1Q:    lui  $14, 0xFFFF
35 ce ff ff //           ori  $14, 0xFFFF          # $r14 <-- 0xFFFFFFFF
(Fail flag 1 Quot)
00 00 00 0d //           break
3c 0e ff ff // fail1R:    lui  $14, 0xFFFF
35 ce ff fe //           ori  $14, 0xFFFE          # $r14 <-- 0xFFFFFFFFE
(Fail flag 1 Rem)
00 00 00 0d //           break
3c 0e ff ff // fail2Q:    lui  $14, 0xFFFF
35 ce ff fd //           ori  $14, 0xFFFFD         # $r14 <-- 0xFFFFFFFDD
(Fail flag 2 Quot)
00 00 00 0d //           break
3c 0e ff ff // fail2R:    lui  $14, 0xFFFF
35 ce ff fc //           ori  $14, 0xFFFFC         # $r14 <-- 0xFFFFFFFCC
(Fail flag 2 Rem)
00 00 00 0d //           break
3c 0e ff ff // fail3Q:    lui  $14, 0xFFFF
35 ce ff fb //           ori  $14, 0xFFFFB         # $r14 <-- 0xFFFFFFFBB
(Fail flag 3 Quot)
00 00 00 0d //           break
3c 0e ff ff // fail3R:    lui  $14, 0xFFFF
35 ce ff fa //           ori  $14, 0xFFFFA         # $r14 <-- 0xFFFFFFFBA
(Fail flag 3 Rem)
00 00 00 0d //           break
3c 0e ff ff // fail4Q:    lui  $14, 0xFFFF
35 ce ff f9 //           ori  $14, 0xFFFF9         # $r14 <-- 0xFFFFFFF9
(Fail flag 4 Quot)
00 00 00 0d //           break
3c 0e ff ff // fail4R:    lui  $14, 0xFFFF
35 ce ff f8 //           ori  $14, 0xFFFF8         # $r14 <-- 0xFFFFFFF8
(Fail flag 4 Rem)

```

Simulator is doing circuit initialization process.
Finished circuit initialization process.

```

-----Register Dump-----
t= 805000.00ps : R[000] = 00000000          R[010] = xxxxxxxx
t= 805000.00ps : R[001] = ffffffff8a          R[011] =
xxxxxxxx
t= 805000.00ps : R[002] = 0000008a          R[012] = xxxxxxxx
t= 805000.00ps : R[003] = 00000000          R[013] = xxxxxxxx
t= 805000.00ps : R[004] = 000000c0          R[014] = xxxxxxxx
t= 805000.00ps : R[005] = 000000c4          R[015] = xxxxxxxx
t= 805000.00ps : R[006] = 000000d4          R[016] = xxxxxxxx
t= 805000.00ps : R[007] = 0000f0f0          R[017] = xxxxxxxx
t= 805000.00ps : R[008] = 00000000          R[018] = xxxxxxxx
t= 805000.00ps : R[009] = ffffffff          R[019] = xxxxxxxx
t= 805000.00ps : R[00a] = 0000f0f0          R[01a] = xxxxxxxx
t= 805000.00ps : R[00b] = ffffffff          R[01b] = xxxxxxxx
t= 805000.00ps : R[00c] = fffffaf5          R[01c] = xxxxxxxx
t= 805000.00ps : R[00d] = ffff5555          R[01d] = 000003fc
t= 805000.00ps : R[00e] = 00000000          R[01e] = xxxxxxxx
t= 805000.00ps : R[00f] = 100100c0          R[01f] = 00000014
-----Memory Dump-----
t= 805000.00ps : M[0c0] = 000000c0

```

```
t= 805000.00ps : M[0c4] = 000000c4
t= 805000.00ps : M[0c8] = 000000c8
t= 805000.00ps : M[0cc] = 000000cc
t= 805000.00ps : M[0d0] = 000000d0
t= 805000.00ps : M[0d4] = 000000d4
t= 805000.00ps : M[0d8] = ffffffff8a
t= 805000.00ps : M[0dc] = xxxxxxxx
t= 805000.00ps : M[0e0] = xxxxxxxx
t= 805000.00ps : M[0e4] = xxxxxxxx
t= 805000.00ps : M[0e8] = xxxxxxxx
t= 805000.00ps : M[0ec] = xxxxxxxx
t= 805000.00ps : M[0f0] = xxxxxxxx
t= 805000.00ps : M[0f4] = xxxxxxxx
t= 805000.00ps : M[0f8] = xxxxxxxx
t= 805000.00ps : M[0fc] = xxxxxxxx
```

Stopped at time : 805 ns : File

"C:/Users/javis/OneDrive/Documents/School/CECS

440/i_prefer_The_Real_Pipeline_14/PipeLine_Stage_5.v" Line 54

```

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

20 01 ff 8a //              addi $01, $00, -118    # $r01 <-- 0xFFFFFFFF8A
20 02 00 8a //              addi $02, $00, 138          # $r02 <-- 0x0000008A
0c 10 00 08 //              jal  blt_tests
ad e1 00 18 //              sw   $01, 0x18($15)          # M[D8] <-- 0xFFFFFFFF8A
ad e2 00 1c //              sw   $02, 0x1C($15)          # M[DC] <-- 0x0000008A
00 00 00 0d //              break

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

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

```

```

20 0e 00 00 // bgtz_p3:   addi $14, $00, 0          # set $r14 to 0000_0000
03 e0 00 08 //           jr   $31                 # return from subroutine

```

Simulator is doing circuit initialization process.
Finished circuit initialization process.

```

-----Register Dump-----
t= 295000.00ps : R[000] = 00000000          R[010] = xxxxxxxx
t= 295000.00ps : R[001] = 12345678          R[011] = xxxxxxxx
t= 295000.00ps : R[002] = 87654321          R[012] = xxxxxxxx
t= 295000.00ps : R[003] = abcdef01          R[013] = xxxxxxxx
t= 295000.00ps : R[004] = 01fedcba          R[014] = xxxxxxxx
t= 295000.00ps : R[005] = 5a5a5a5a          R[015] = xxxxxxxx
t= 295000.00ps : R[006] = ffffffff          R[016] = xxxxxxxx
t= 295000.00ps : R[007] = 100103f0          R[017] = xxxxxxxx
t= 295000.00ps : R[008] = fffffeff          R[018] = xxxxxxxx
t= 295000.00ps : R[009] = fffffefe          R[019] = xxxxxxxx
t= 295000.00ps : R[00a] = fffffefd          R[01a] = xxxxxxxx
t= 295000.00ps : R[00b] = fffffefc          R[01b] = xxxxxxxx
t= 295000.00ps : R[00c] = fffffefb          R[01c] = xxxxxxxx
t= 295000.00ps : R[00d] = fffffefa          R[01d] = 000003fc
t= 295000.00ps : R[00e] = fffffef9          R[01e] = xxxxxxxx
t= 295000.00ps : R[00f] = fffffef8          R[01f] = xxxxxxxx

```

```

-----Memory Dump-----
t= 295000.00ps : M[0c0] = xxxxxxxx
t= 295000.00ps : M[0c4] = xxxxxxxx
t= 295000.00ps : M[0c8] = xxxxxxxx
t= 295000.00ps : M[0cc] = xxxxxxxx
t= 295000.00ps : M[0d0] = xxxxxxxx
t= 295000.00ps : M[0d4] = xxxxxxxx
t= 295000.00ps : M[0d8] = xxxxxxxx
t= 295000.00ps : M[0dc] = xxxxxxxx
t= 295000.00ps : M[0e0] = xxxxxxxx
t= 295000.00ps : M[0e4] = xxxxxxxx
t= 295000.00ps : M[0e8] = xxxxxxxx
t= 295000.00ps : M[0ec] = xxxxxxxx
t= 295000.00ps : M[0f0] = xxxxxxxx
t= 295000.00ps : M[0f4] = xxxxxxxx
t= 295000.00ps : M[0f8] = xxxxxxxx
t= 295000.00ps : M[0fc] = xxxxxxxx

```

Stopped at time : 295 ns : File

"C:/Users/javis/OneDrive/Documents/School/CECS

440/i_prefer_The_Real_Pipeline_14/PipeLine_Stage_5.v" Line 54


```

@0
00 00 00 1f // main:      setie
3c 01 12 34 //            lui   $01, 0x1234
34 21 56 78 //            ori   $01, 0x5678      # LI   R01,  0x12345678
3c 02 87 65 //            lui   $02, 0x8765
34 42 43 21 //            ori   $02, 0x4321      # LI   R02,  0x87654321
3c 03 ab cd //            lui   $03, 0xABCD
34 63 ef 01 //            ori   $03, 0xEF01      # LI   R03,  0xABCDEF01
3c 04 01 fe //            lui   $04, 0x01FE
34 84 dc ba //            ori   $04, 0xDCBA      # LI   R04,  0x01FEDCBA
3c 05 5a 5a //            lui   $05, 0x5A5A
34 a5 5a 5a //            ori   $05, 0x5A5A      # LI   R05,  0x5A5A5A5A
3c 06 ff ff //            lui   $06, 0xFFFF
34 c6 ff ff //            ori   $06, 0xFFFF      # LI   R06,  0xFFFFFFFF
3c 07 ff ff //            lui   $07, 0xFFFF
34 e7 ff 00 //            ori   $07, 0xFF00      # LI   R07,  0xFFFFFFF0

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

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

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

76 11 00 00 // out_IO:   output $17, 0($16)      # output  [R16], R17
00 11 88 83 //            sra    $17, $17, 2      # change the pattern by
shifting twice
22 10 00 04 //            addi  $16, $16, 4      # increment the memory
pointer 4 bytes
22 52 ff ff //            addi  $18, $18, -1     # decrement the loop
counter

```

```

16 40 ff fb //          bne    $18, $00, out_IO #   and jmp to top if not
finished

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

```

(v1, using \$ra)

Finished circuit initialization process.

run 1.00us

-----Register Dump-----

```

t= 1805000.00ps : R[000] = 00000000          R[010] = 100100c0
t= 1805000.00ps : R[001] = 12345678          R[011] = ffffffff
t= 1805000.00ps : R[002] = 87654321          R[012] = 00000000
t= 1805000.00ps : R[003] = abcdef01          R[013] = 8000ffff
t= 1805000.00ps : R[004] = 01fedcba          R[014] = e0003fff
t= 1805000.00ps : R[005] = 5a5a5a5a          R[015] = f8000fff
t= 1805000.00ps : R[006] = ffffffff          R[016] = fe0003ff
t= 1805000.00ps : R[007] = 100103f0          R[017] = ff8000ff
t= 1805000.00ps : R[008] = ffffffff          R[018] = ffe0003f
t= 1805000.00ps : R[009] = fffffefe          R[019] = xxxxxxxx
t= 1805000.00ps : R[00a] = fffffefd          R[01a] = xxxxxxxx
t= 1805000.00ps : R[00b] = fffffefc          R[01b] = xxxxxxxx
t= 1805000.00ps : R[00c] = fffffefb          R[01c] = xxxxxxxx
t= 1805000.00ps : R[00d] = fffffefa          R[01d] = xxxxxxxx
t= 1805000.00ps : R[00e] = fffffef9          R[01e] = xxxxxxxx
t= 1805000.00ps : R[00f] = fffffef8          R[01f] = 00000044

```

-----IO Memory Dump-----

```

t= 1805000.00ps : M[0c0] = 8000ffff
t= 1805000.00ps : M[0c4] = e0003fff
t= 1805000.00ps : M[0c8] = f8000fff
t= 1805000.00ps : M[0cc] = fe0003ff
t= 1805000.00ps : M[0d0] = ff8000ff
t= 1805000.00ps : M[0d4] = ffe0003f
t= 1805000.00ps : M[0d8] = fff8000f
t= 1805000.00ps : M[0dc] = fffe0003
t= 1805000.00ps : M[0e0] = ffff8000
t= 1805000.00ps : M[0e4] = ffffe000
t= 1805000.00ps : M[0e8] = fffff800
t= 1805000.00ps : M[0ec] = fffffe00
t= 1805000.00ps : M[0f0] = fffffff8
t= 1805000.00ps : M[0f4] = fffffffe
t= 1805000.00ps : M[0f8] = ffffffff
t= 1805000.00ps : M[0fc] = ffffffff

```

-----Memory Dump-----

```

t= 1805000.00ps : M[3f0] = fffffef8

```

Stopped at time : 1805 ns : File

"C:/Users/javis/Desktop/FINAL_PIPELINE_13/PipeLine_Stage_5.v" Line 54

iM 14

```

@0
00 00 00 1f // main:      setie
3c 01 12 34 //            lui   $01, 0x1234
34 21 56 78 //            ori   $01, 0x5678      # LI   R01,  0x12345678
3c 02 87 65 //            lui   $02, 0x8765
34 42 43 21 //            ori   $02, 0x4321      # LI   R02,  0x87654321
3c 03 ab cd //            lui   $03, 0xABCD
34 63 ef 01 //            ori   $03, 0xEF01      # LI   R03,  0xABCDEF01
3c 04 01 fe //            lui   $04, 0x01FE
34 84 dc ba //            ori   $04, 0xDCBA      # LI   R04,  0x01FEDCBA
3c 05 5a 5a //            lui   $05, 0x5A5A
34 a5 5a 5a //            ori   $05, 0x5A5A      # LI   R05,  0x5A5A5A5A
3c 06 ff ff //            lui   $06, 0xFFFF
34 c6 ff ff //            ori   $06, 0xFFFF      # LI   R06,  0xFFFFFFFF
3c 07 ff ff //            lui   $07, 0xFFFF
34 e7 ff 00 //            ori   $07, 0xFF00      # LI   R07,  0xFFFFFFF0

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

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

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

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

```

```

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

# return from interrupt

# Note opcode=0x1E and

$rs=0x1D, i.e. $sp

```

Finished circuit initialization process.

run 1.00us

```

-----Register Dump-----
t= 1865000.00ps : R[000] = 00000000          R[010] = 100100c0
t= 1865000.00ps : R[001] = 12345678          R[011] = ffffffff
t= 1865000.00ps : R[002] = 87654321          R[012] = 00000000
t= 1865000.00ps : R[003] = abcdef01          R[013] = 8000ffff
t= 1865000.00ps : R[004] = 01fedcba          R[014] = e0003fff
t= 1865000.00ps : R[005] = 5a5a5a5a          R[015] = f8000fff
t= 1865000.00ps : R[006] = ffffffff          R[016] = fe0003ff
t= 1865000.00ps : R[007] = 100103f0          R[017] = ff8000ff
t= 1865000.00ps : R[008] = ffffffff          R[018] = ffe0003f
t= 1865000.00ps : R[009] = ffffffff          R[019] = xxxxxxxx
t= 1865000.00ps : R[00a] = ffffffff          R[01a] = xxxxxxxx
t= 1865000.00ps : R[00b] = ffffffff          R[01b] = xxxxxxxx
t= 1865000.00ps : R[00c] = ffffffff          R[01c] = xxxxxxxx
t= 1865000.00ps : R[00d] = ffffffff          R[01d] = 000003fc
t= 1865000.00ps : R[00e] = ffffffff          R[01e] = xxxxxxxx
t= 1865000.00ps : R[00f] = ffffffff          R[01f] = xxxxxxxx
-----IO Memory Dump-----
t= 1865000.00ps : M[0c0] = 8000ffff
t= 1865000.00ps : M[0c4] = e0003fff
t= 1865000.00ps : M[0c8] = f8000fff
t= 1865000.00ps : M[0cc] = fe0003ff
t= 1865000.00ps : M[0d0] = ff8000ff
t= 1865000.00ps : M[0d4] = ffe0003f
t= 1865000.00ps : M[0d8] = fff8000f
t= 1865000.00ps : M[0dc] = fffe0003
t= 1865000.00ps : M[0e0] = ffff8000
t= 1865000.00ps : M[0e4] = ffffe000
t= 1865000.00ps : M[0e8] = fffff800
t= 1865000.00ps : M[0ec] = fffffe00
t= 1865000.00ps : M[0f0] = ffffffff80
t= 1865000.00ps : M[0f4] = fffffffe0
t= 1865000.00ps : M[0f8] = ffffffff8
t= 1865000.00ps : M[0fc] = ffffffffefe
-----Memory Dump-----
t= 1865000.00ps : M[3f0] = fffffef8

```

Stopped at time : 1865 ns : File
"C:/Users/javis/OneDrive/Documents/School/CECS
440/i_prefer_The_Real_Pipeline_14/PipeLine_Stage_5.v" Line 55

IV. Hardware Implementation (10 Diagrams)

V. Additional Discussion/Comments

1. A look at the ISR

The Interrupt Service Routine is currently designed to be able to handle multiple Interrupts, and successfully move the flow of instructions from the current routine, save the PC, the flags, and go to the ISR. In order to get the Pipeline to properly implement a quick ISR without adding lots of scratch registers, there are added muxes and additional busses that carry the Flags and the PC to the dMEM IN and OUT.

First when the External Interrupt goes high, and the Interrupt Enable is High, the Pipeline Control Unit will issue a ISR flag that causes a Halt on the PC and we stop reading from the Instruction Memory. There is a single 5'bit temporary register for the Flags because the Control word generated from the Interrupt will be 1 clock behind the previous Instruction to pass through the ALU. By using a scratch register, the flags stay 'fresh' for 1 additional clock cycle, and allow a couple of MUX's to direct the Flags to memory. They will be stored by the current \$sp which was pre incremented by 4. After the Write to Data Memory occurs, we Write back to the RegFile in stage 5 to update the \$sp. However, the Control words after the Flag push, are designed to Push the current PC value (not PC+4, that would cause an instruction to skip) into the Stack. To do so the Forwarding Unit takes care of the updated \$sp value that was already incremented by 4. The PC push will also increment the \$sp by 4 again so it points to the current location of the PC Address. Finally, the last control word of the go_to_ISR section will be the SP_Init ALU command to get the location of the ISR and we perform a PC load directly from the output of Data Memory using a 2 to 1 MUX with the ISR Flag controlling the HIGH select. Then the ISR Stall is lifted, the Interrupt Enable is set to 1'b0 LOW, and the PC will now execute instructions at the ISR Address (in our case line 200 of the Instruction Memory).

After finishing the ISR, as of Module 14, the RETI instruction is issued and it will allow the processor to leave the ISR and return to the previous routine of instructions. In order to do this, the Control Unit will issue a Leaving ISR Stall, and the Return Address on the top of the \$sp Stack will be popped off. Using the MUX at the back of the Data Memory that we jumped to the ISR with, we can also use it to load the PC with a DMEM Value if the leave_ISR flag is HIGH. The PC will be loaded because the PC has a priority for Reset>Load>Stall>PC+4, and still stay stalled after we load. The Control Unit will have the next set of control words pre decrement the \$sp by 4 for the first time in the leave_ISR section. The Flags will be popped from the Data Memory, but pass retain their value in a register until the ISR is done. Finally, the \$sp will do a single run through the ALU to do a post decrement by 4 and write back to the \$sp in the RegFile. The Interrupt Enable Flag will go HIGH once again, the Stall is lifted, and the Processor resumes normal flow.

Works Cited

- 1) Professor Alison's CECS 440 Addressing Mode Slides
- 2) MIPS Green Sheet
- 3) Abstract vs Introduction (guidelines)
<https://www.enago.com/academy/abstract-versus-introduction-difference/>
- 4) Texas Instruments TM4C123GH6PM Data Sheet (as a format and style reference)
- 5) Computer Organization and Design - The Hardware/Software Interface 5th Ed.
By David A Patterson and John L. Hennessy