# 

**CECS 440 –MIPS Instruction Set Architecture**

Benjamin Santos - 015780126

Naoaki Takatsu - 015746144

November 27, 2018

*This page intentionally left blank*

**Table of Contents**

Introduction ………...……………………………………………………………………… 6

A. Memory Structure ……………………………………………………………………… 7

B. Processor Register Set ………………………………………………………………….. 9

C. Data Types …...…………………………………………………………………………. 10

D. Addressing Modes ……………………………………………………………………… 11

E. Instruction Set ………………………………………………………………………….. 14  
 1. R-Type Instructions …………………………………………………………….. 14

a. sll ………………………………………………………………………... 14

b. srl ………………………………………………………………………....16

c. sra ……………………………………………………………………….. 18

d. jr .………………………………………………………………………... 20

e. mfhi ……………………………………………………………………... 22

f. mflo ……………………………………………………………………... 24

g. mult ……………………………………………………………………….26

h. div ………………………………………………………………………. 28

i. add ……………………………………………………………………….. 30

j. addu ……………………………………………………………………… 32

k. sub ………………………………………………………………………. 34

l. subu ……………………………………………………………………… 36

m. and ……………………………………………………………………… 38

n. or ………………………………………………………………………... 40

o. xor ………………………………………………………………………. 42

p. nor ………………………………………………………………………. 44

q. slt ……………………………………………………………………….. 46

r. sltu ………………………………………………………………………. 48

s. break …………………………………………………………………….. 50

t. setie ……………………………………………………………………… 51

2. I-Type Instructions ……………………………………………………………… 52

a. beq ……………………………………………………………………… 52

b. bne ……………………………………………………………………… 54

c. blez ……………………………………………………………………… 56

d. bgtz ……………………………………………………………………… 58

e. addi ……………………………………………………………………… 60

f. slti ……………………………………………………………………….. 62

g. sltiu ……………………………………………………………………… 64

h. andi ……………………………………………………………………… 66

i. ori ………………………………………………………………………... 68

j. xori ………………………………………………………………………. 70

k. lui ……………………………………………………………………….. 72

l. lw ………………………………………………………………………... 74

m. sw ………………………………………………………………………. 76

3. J-Type Instructions ……………………………………………………………... 78

a. j …………………………………………………………………………. 83

b. jal ……………………………………………………………………….. 80

4. Enhanced Instructions …………………………………………………..……..... 82

a. input …………………………………………………………………….. 82

b. output …………………………………………………………………… 84

c. reti ………………………………………………………………………. 86

d. rotleft .. …………………….……………………………………………. 87

e. rotright ………………………………………………………………….. 89

F. Verilog Source Code …………………………………………………………..……...... 91

a. Top Level Module ……………….……………………………………………… 91

b. CPU ……………...……………………………………………………………… 94

c. MIPS Control Unit (MCU)……………………………………………………… 97

d. Instruction Unit (IU) ……………………………………………………………. 121

e. Instruction Memory (iM) ……………………………………………………….. 123

f. Integer Datapath (IDP) ………………………………………………………….. 124

g. Regfile\_32 ………………………………………………………………. ……... 126

h. alu\_32 …………………………………………………………………………… 127

i. MIPS\_32 ………………………………………………………………………… 130

j. MPY\_32 …………………………………………………………………………. 135

k. DIV\_32 …………………………………………………………………………. 136

l. Barrel Shift ………………………………………………………………. ……... 137

m. Rotate\_32 ………………………………………………………………………. 139

n. DataMemory ……………………………………………………………………. 141

o. InputOutput ……………………………………………………………………... 142

G. Verilog Annotation ………………………………………………………..……............ 143

a. Instruction Memory Module 1 ………………………………………………….. 143

b. Instruction Memory Module 2 ………………………………………………….. 144

c. Instruction Memory Module 3 ………………………………………………….. 145

d. Instruction Memory Module 4 ………………………………………………….. 146

e. Instruction Memory Module 5 ………………………………………………….. 147

f. Instruction Memory Module 6 ………………………………………………….. 148

g. Instruction Memory Module 7 ………………………………………………….. 149

h. Instruction Memory Module 8 ………………………………………………….. 150

i. Instruction Memory Module 9 …………………………………………………... 152

j. Instruction Memory Module 10 …………………………………………………. 154

k. Instruction Memory Module 11 ………………………………………………… 156

l. Instruction Memory Module 12 …………………………………………………. 158

m. Instruction Memory Module 13 ………………………………………………... 159

n. Instruction Memory Module 14 ………………………………………………… 161

o. Instruction Memory Module Enhanced ………………………………………… 163

H. Hardware Block Diagram ………………………………………………..…….………. 165

a. CPU\_test3 ………………………………………………………………………. 165

b. CPU …………………………………………………………………………….. 166

1. Integer Datapath ………………………………………………………… 167

2. Instruction Unit …………………………………………………………. 168

c. ALU\_32 ………………………………………………………………………… 169

d. Regfile\_32 ………………………………………………………………………. 170

e.barrelSHIFT\_32 …………………………………………………………………. 171

f. rotate\_32 ………………………………………………………………………... 172

**Introduction**

This MIPS reduced instruction set architecture is specifically designed and created for educational purposes. The MIPS includes the following characteristics:

* non-pipeline multicycle data path
* 32 general purpose registers and 4 flag registers
* Harvard memory architecture granting bidirectional communication through 3 separate memory blocks
* 32-bit size instructions
* 5 major addressing modes to fetch operands at runtime
* Original enhanced key instructions

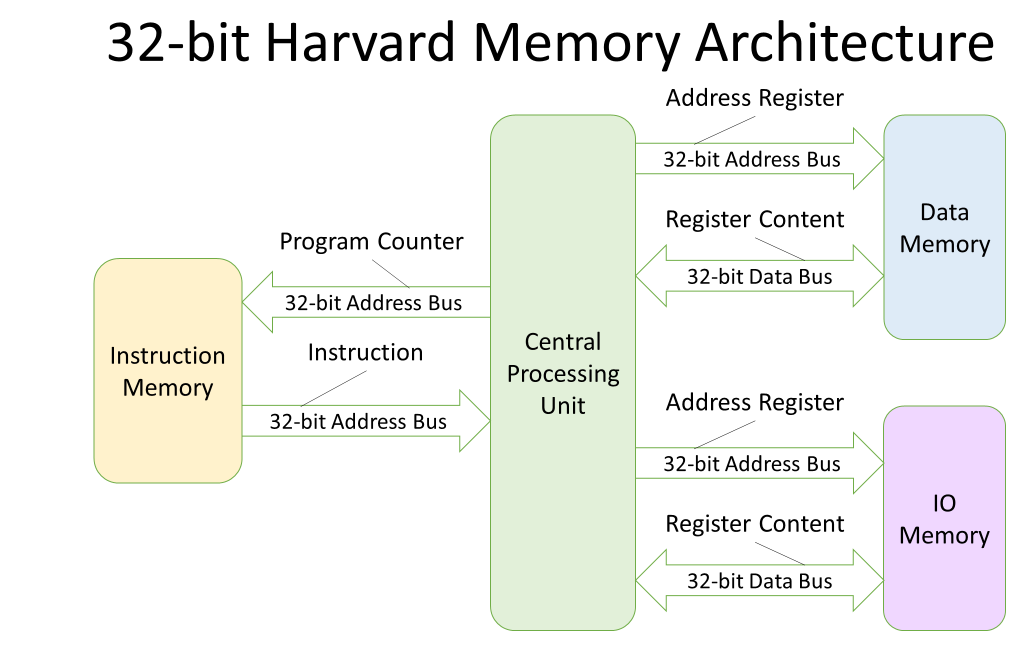
**A. Memory Structure**

This MIPS uses a 32-bit Harvard memory architecture. The Central Processing Unit (CPU) establishes separate bus communication between three memory units: Instruction Memory, Data Memory, and Input Output Memory. All memory sizes are set to 4098 x 8. Addresses are byte addressable (4-bit) with each address containing 8-bit data values, creating 32-bit data per address byte. The memory placement follows a big-endian format.

Ex.

0x1234\_ABCD stored at 0x2258

|  |  |
| --- | --- |
| Address | Content |
| … | … |
| 0x225B | 0x12 |
| 0x226A | 0x34 |
| 0x2259 | 0xAB |
| 0x2258 | 0xCD |
| … | … |

****

**B. Processor Register Set**

32-bit Register File

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Number | Use | Preserved Across a Call |
| $zero | 0 | Constant 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 | Temporary | No |
| $s0-$s7 | 16-23 | Saved Temporary | Yes |
| $t8-$t9 | 24-25 | Temporary | 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 | No |

Other Registers

|  |  |  |
| --- | --- | --- |
| Register Name | Symbol | Size (bit) |
| Carry | C | 1 |
| Negative | N | 1 |
| Zero | Z | 1 |
| Overflow | V | 1 |
| Program Counter | PC | 32 |
| Instruction Register | IR | 32 |
| rt Register | RT | 32 |
| rs Register | RS | 32 |
| Higher Register | HI | 32 |
| Lower Register | LO | 32 |
| Arithmetic Logic Unit Register | ALU\_reg | 32 |
| Data In | D\_in | 32 |

**C. Data Types**

Single Word Integer

* 16-bit integer for register and immediate type operands/results

Double Word Integer

* 32-bit integer for or immediate, multiplication, and division, results

**D. Addressing Modes**

Register

Use:

Register type instructions

Operand:

rs and rt

Ex.

If r2’s content is 0x0080 and   
r3’s content is 0x000F

r1 gets 0x0080 – 0x000F = 0x0071

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Memory Addr | Instruction |  | Addr | Contents |
|  | 0x084C | sub r1, r2, r3 |  | 0x0844 | ----- |
|  |  |  |  | 0x0848 | ----- |
|  |  |  |  | 0x084C | opc r1 r9 r3 |
|  |  |  |  | 0x0850 | ----- |
|  |  |  |  | 0x0854 | ----- |
|  |  |  |  | 0x0858 | ----- |

Base (Indexed)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Memory Addr | Instruction |  | Addr | Contents |
|  | 0x1108 | load r6, 0x0328(r2) |  | 0x1100 | ----- |
|  |  |  |  | 0x1104 | ----- |
|  |  |  |  | 0x1108 | opc r6 r2 base |
|  |  | Use:  Load and store  Operand:  Result of adding 16-bit trailing word offset and content of rs register  Ex.  If r2’s content is 0x73C4:  0x0328 + 0x73C4 = 0x76EC  So r6 gets 0x0021 |  | 0x110C | 0x0328 |
|  |  |  |  | 0x1110 | ----- |
|  |  |  |  | 0x1114 | ----- |
|  |  |  |  | … | … |
|  |  |  |  | 0x73C0 | ----- |
|  |  |  |  | 0x76EC | 0x0021 |
|  |  |  |  | 0x73C8 | ----- |

PC Direct

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Memory Addr | Instruction |  | Addr | Contents |
|  | 0x0928 | j 0x1000 |  | 0x0920 | ----- |
| Use:  Jump type instructions  Operand:  26-bit trailing word  Ex.  PC gets 0x1000 |  |  |  | 0x0924 | ----- |
|  |  |  |  | 0x0928 | opc PC\_dir |
|  |  |  |  | 0x092C | 0x1000 |
|  |  |  |  | 0x0930 | ----- |
|  |  |  |  | 0x0934 | ----- |
|  |  |  |  | … | … |
|  |  |  |  | 0x0FFC | ----- |
|  |  |  |  | 0x1000 | instr we'll jump to |
|  |  |  |  | 0x1004 | ----- |

PC Relative

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Memory Addr | Instruction |  | Addr | Contents |
|  | 0x0330 | beq r3, r0, 0xFF00 |  | 0x0224 | ----- |
| Use:  Branch instructions  If condition is true:  PC 🡨 PC + 4 + 16-bit address  Otherwise:  PC 🡨 PC + 4  Ex.  If r3’s content is 0x0000  PC gets PC + 4 + 0xFF00 = 0x0238  If r3’s content is 0x0001 PC gets PC + 4 = 0x0338 |  |  |  | 0x0238 | instr we'll jump to |
|  |  |  |  | 0x023C | ----- |
|  |  |  |  | … | … |
|  |  |  |  | 0x0328 | ----- |
|  |  |  |  | 0x032C | ----- |
|  |  |  |  | 0x0330 | opc r3 r0 PC\_rel |
|  |  |  |  | 0x0334 | 0xFFF0 |
|  |  |  |  | 0x0338 | next instr |
|  |  |  |  | 0x033C | ----- |

Immediate

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Memory Addr | Instruction |  | Addr | Contents |
|  | 0x0230 | addi r0, r8, 0xA21C |  | 0x0228 | ----- |
|  |  |  |  | 0x022C | ----- |
|  |  | Use:  Immediate type instructions  Operand:  16-bit trailing word  Ex.  If r8’s content is 0x3019:  r0 gets 0x3019 + 0xA21C = 0xD235 |  | 0x0230 | opc r0 r8 imm |
|  |  |  |  | 0x0234 | 0xA21C |
|  |  |  |  | 0x0238 | ----- |
|  |  |  |  | 0x023C | ----- |

**E. Instruction Set**

Shift Left Logical sll

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| R  **0x00** | **0x00** | **rt** | **rd** | **shtamt** | **0x00** |

6 5 5 5 5 6

**Format:** sll $rd, $rt, shtamt

**Purpose:**

Execute logical shift left by specified shift amount.

**Description:** rd 🡨 rt << shtamt

This operation will execute a zero-fill logical shift left to a register $rt by shift amount shtamt and store the result into a destination register $rd.

**Restrictions:**

* rd cannot be a zero register

**Operation:**

rd 🡨 rt << shtamt

**Exceptions:**

* Negative flag will be affected by $rd’s parity bit
* Zero flag will be set if $rd is 0x0

**Programming Notes:**

None

Shift Left Logical - Example sll

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| R  **00\_0000** | **0\_0000** | **1\_0010** | **1\_0001** | **0\_0010** | **00\_0000** |

6 5 5 5 5 6

$s1 = 0x0000\_0000

$s2 = 0x1234\_5678

sll $s1, $s2, 2

0001\_0010\_0011\_0100\_0101\_0110\_0111\_1000 00

$s1 🡨 0x48D1\_59E0

Shift Right Logical srl

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| R  **0x00** | **0x00** | **rt** | **rd** | **shtamt** | **0x02** |

6 5 5 5 5 6

**Format:** srl $rd, $rt, shtamt

**Purpose:**

Execute logical shift right by specified shift amount

**Description:** rd 🡨 rt >> shtamt

This operation will execute a zero-fill logical shift right to a register $rt by shift amount shtamt and store the result into a destination register $rd.

**Restrictions:**

* rd cannot be a zero register

**Operation:**

rd 🡨 rt >> shtamt

**Exceptions:**

* Zero flag will be set if $rd is 0x0

**Programming Notes:**

None

Shift Right Logical - Example srl

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| R  **00\_0000** | **0\_0000** | **1\_0001** | **1\_0100** | **0\_0001** | **00\_0001** |

6 5 5 5 5 6

$s4 = 0x0000\_0000

$s1 = 0x1010\_1010

srl $s4, $s1, 1

0 0001\_0000\_0001\_0000\_0001\_0000\_0001\_0000

$s4 🡨 0x0808\_0808

Shift Right Arithmetic sra

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| R  **0x00** | **0x00** | **rt** | **rd** | **shtamt** | **0x03** |

6 5 5 5 5 6

**Format:** sra $rd, $rt, shtamt

**Purpose:**

Execute arithmetic shift right by specified shift amount

**Description:** rd 🡨 ( rt >> shtamt ) || 0x10000

This operation will execute an arithmetic logical shift right to a register $rt by shift amount shtamt and store the result into a destination register $rd.

**Restrictions:**

* rd cannot be a zero register

**Operation:**

at 🡨 rt;

for( int i = 0; i < shtamt; i++ )

at 🡨 { rt(31), rt(31), at(30:1) };

rd 🡨 at;

**Exceptions:**

* Negative flag will be affected by $rd’s parity bit
* Zero flag will be set if $rd is 0x0

**Programming Notes:**

None

Shift Right Arithmetic - Example sra

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| R  **00\_0000** | **0\_0000** | **1\_0010** | **1\_0011** | **0\_0011** | **00\_0011** |

6 5 5 5 5 6

$s3 = 0x0000\_0000

$s2 = 0xFEFA\_1934

sra $s3, $s2, 3

1 111 111\_1110\_1111\_1010\_0001\_1001\_0011\_0100

$s3 🡨 0xFFDF\_4326

Jump Register jr

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| R  **0x00** | **rs** | **0x00** | **0x00** | **0x00** | **0x08** |

6 5 5 5 5 6

**Format:** jr $rs

**Purpose:**

Jump to specified address

**Description:** PC 🡨 rs

This operation will execute a jump with its target address set to $rs.

**Restrictions:**

* None

**Operation:**

PC 🡨 rs

**Exceptions:**

* None

**Programming Notes:**

None

Jump Register - Example jr

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| R  **00\_0000** | **1\_0110** | **0\_0000** | **0\_0000** | **0\_0000** | **00\_1000** |

6 5 5 5 5 6

$s6 = 0xABBA\_CCDD

jr $s6

PC 🡨 0xABBA\_CCDD

Move from HI mfhi

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| R  **0x00** | **0x00** | **0x00** | **rd** | **0x00** | **0x10** |

6 5 5 5 5 6

**Format:** mfhi $rd

**Purpose:**

Copy content of special register HI and store it to a register

**Description:** rd 🡨 HI

This operation will copy data from special register $HI and store it into destination register $rd.

**Restrictions:**

* rd cannot be a zero register

**Operation:**

rd 🡨 HI

**Exceptions:**

* Negative flag will be affected by $rd’s parity bit
* Zero flag will be set if $rd is 0x0

**Programming Notes:**

None

Move from HI - Example mfhi

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| R  **00\_0000** | **0\_0000** | **0\_0000** | **0\_1111** | **0\_0000** | **01\_0000** |

6 5 5 5 5 6

$t7 = 0x0000\_0000

$HI = 0x0095\_FFE5

mfhi $t7

$t7 🡨 0x0095\_FFE5

Move from LO mflo

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| R  **0x00** | **0x00** | **0x00** | **rd** | **0x00** | **0x12** |

6 5 5 5 5 6

**Format:** mflo $rd

**Purpose:**

Copy content of special register LO and store it to a register

**Description:** rd 🡨 LO

This operation will copy data from special register $LO and store it into destination register $rd.

**Restrictions:**

* rd cannot be a zero register

**Operation:**

rd 🡨 LO

**Exceptions:**

* Negative flag will be affected by $rd’s parity bit
* Zero flag will be set if $rd is 0x0

**Programming Notes:**

None

Move from LO - Example mflo

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| R  **00\_0000** | **0\_0000** | **0\_0000** | **1\_0011** | **0\_0000** | **01\_0010** |

6 5 5 5 5 6

$s3 = 0x0000\_0000

$LO = 0xCD12\_8876

mflo $s3

$s3 🡨 0xCD12\_8876

Multiply mult

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| R  **0x00** | **rs** | **rt** | **0x00** | **0x00** | **0x18** |

6 5 5 5 5 6

**Format:** mult $rs, $rt

**Purpose:**

Multiply two registers with each other and store result

**Description:** { HI, LO } 🡨 rs \* rt

This operation will multiply $rs by $rt and store the upper 32 bit result to special register $HI and lower 32 bit result to special register $LO.

**Restrictions:**

* None

**Operation:**

**{** HI, LO } 🡨 rs \* rt

**Exceptions:**

* Negative flag will be affected by $HI’s parity bit
* Zero flag will be set if $HI and $LO are 0x0

**Programming Notes:**

Performs multiplication on signed values

Multiply - Example mult

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| R  **00\_0000** | **1\_0001** | **1\_0101** | **0\_0000** | **0\_0000** | **01\_1000** |

6 5 5 5 5 6

$s1 = 0x3D65\_9900

$s5 = 0x113B\_0008

mult $s1, $s5

$s1 \* $s5 = 0x0421\_E595\_2E2C\_C800

$HI 🡨 0x0421\_E595

$LO 🡨 0x2E2C\_C800

Divide div

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| R  **0x00** | **rs** | **rt** | **0x00** | **0x00** | **0x1A** |

6 5 5 5 5 6

**Format:** div $rs, $rt

**Purpose:**

Divide a dividend register by a divisor register and store the result to a register

**Description: {** HI, LO } 🡨 rs / rt

This operation will divide $rs by $rt and store the result into register $rd.

**Restrictions:**

* rt cannot be 0x0

**Operation:**

LO 🡨 rs / rt

HI 🡨 rs % rt

**Exceptions:**

* Negative flag will be affected by $LO’s parity bit
* Overflow flag will be set if rt is 0x0
* Zero flag will be set if $HI and $LO are 0x0

**Programming Notes:**

Performs division on signed values. Division by zero will set an overflow flag and may result in unpredictable value for HI and LO.

Divide - Example div

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| R  **00\_0000** | **1\_0001** | **1\_0111** | **0\_0000** | **0\_0000** | **01\_1010** |

6 5 5 5 5 6

$s1 = 0x0228\_1FBC

$s7 = 0x000C\_1FA4

div $s1, $s7

$s1 % $s7 = 0x0006\_8FE8

$s1 / $s7 = 0x0000\_002D

$HI 🡨 0x0006\_8FE8

$LO 🡨 0x0000\_002D

Signed Add - Example add

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| R  **0x00** | **rs** | **rt** | **rd** | **0x00** | **0x20** |

6 5 5 5 5 6

**Format:** add $rd, $rs, $rt

**Purpose:**

Signed add two registers with each other and store the result to a register

**Description:** rd 🡨 rs + rt

This operation will signed add $rs to $rt and store the result into register $rd.

**Restrictions:**

* rd cannot be a zero register

**Operation:**

{ cf, rd } 🡨 rs + rt

**Exceptions:**

* Negative flag will be affected by $rd’s parity bit
* Carry flag will be set if cf is a 1’b1
* Overflow flag will be set for the following two cases:

1. positive + positive = negative
2. negative + negative = positive

* Zero flag will be set if $rd is 0x0

**Programming Notes:**

None

Signed Add - Example add

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| R  **00\_0000** | **1\_0010** | **1\_0100** | **0\_1000** | **0\_0000** | **10\_0000** |

6 5 5 5 5 6

$t0 = 0x0000\_0001

$s2 = 0x7111\_0943

$s4 = 0x39BB\_3A23

add $t0, $s2, $s4

$s2 + $s4 = 0xAACC\_4366

$t0 🡨 0xAACC\_4366

Unsigned Add addu

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| R  **0x00** | **rs** | **rt** | **rd** | **0x00** | **0x21** |

6 5 5 5 5 6

**Format:** addu $rd, $rs, $rt

**Purpose:**

Unsigned add two registers with each other and store the result to a register

**Description:** rd 🡨 rs + rt

This operation will unsigned add $rs to $rt and store the result into register $rd.

**Restrictions:**

* rd cannot be a zero register

**Operation:**

rd 🡨 rs + rt

**Exceptions:**

* Carry flag will be set if cf is a 1’b1
* Zero flag will be set if $rd is 0x0

**Programming Notes:**

None

Unsigned Add – Example addu

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| R  **00\_0000** | **1\_0001** | **1\_0010** | **1\_0000** | **0\_0000** | **10\_0001** |

6 5 5 5 5 6

$s0 = 0x0001\_0001

$s1 = 0x9900\_891D

$s2 = 0x00FD\_AA51

addu $s0, $s1, $s2

$s1 + $s2 = 0x99FE\_336E

$s0 🡨 0x99FE\_336E

Signed Subtract sub

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| R  **0x00** | **rs** | **rt** | **rd** | **0x00** | **0x22** |

6 5 5 5 5 6

**Format:** sub $rd, $rs, $rt

**Purpose:**

Signed subtract two registers with each other and store the result to a register

**Description:** rd 🡨 rs - rt

This operation will signed subtract $rs by $rt and store the result into register $rd.

**Restrictions:**

* rd cannot be a zero register

**Operation:**

{ cf, rd } 🡨 rs - rt

**Exceptions:**

* Negative flag will be affected by $rd’s parity bit
* Carry flag will be set if cf is a 1’b1
* Overflow flag will be set for the following two cases:

1. positive - negative = negative
2. negative - positive = positive

* Zero flag will be set if $rd is 0x0

**Programming Notes:**

None

Signed Subtract – Example sub

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| R  **00\_0000** | **1\_0011** | **1\_0100** | **0\_1000** | **0\_0000** | **10\_0010** |

6 5 5 5 5 6

$t0 = 0x0000\_0000

$s3 = 0x0111\_341B

$s4 = 0x8149\_33EF

sub $t0, $s3, $s4

$s3 - $s4 = 0x7FC8\_002C

$t0 🡨 0x7FC8\_002C

Unsigned Subtract subu

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| R  **0x00** | **rs** | **rt** | **rd** | **0x00** | **0x23** |

6 5 5 5 5 6

**Format:** subu $rd, $rs, $rt

**Purpose:**

Unsigned subtract two registers with each other and store the result to a register

**Description:** rd 🡨 rs - rt

This operation will unsigned subtract $rs by $rt and store the result into register $rd.

**Restrictions:**

* rd cannot be a zero register

**Operation:**

rd 🡨 rs - rt

**Exceptions:**

* Carry flag will be set if cf is a 1’b1
* Zero flag will be set if $rd is 0x0

**Programming Notes:**

None

Unsigned Subtract - Example subu

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| R  **00\_0000** | **1\_0000** | **1\_0101** | **0\_1011** | **0\_0000** | **10\_0011** |

6 5 5 5 5 6

$t3 = 0x0000\_0000

$s0 = 0xFFFF\_FFFF

$s5 = 0x341F\_DD89

subu $t3, $s0, $s5

$s0 - $s5 = 0xCBE0\_2276

$t3 🡨 0xCBE0\_2276

And and

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| R  **0x00** | **rs** | **rt** | **rd** | **0x00** | **0x24** |

6 5 5 5 5 6

**Format:** and $rd, $rs, $rt

**Purpose:**

Bitwise and two registers with each other and store the result to a register

**Description:** rd 🡨 rs and rt

This operation will bitwise and $rs with $rt and store the result into register $rd.

**Restrictions:**

* rd cannot be a zero register

**Operation:**

rd 🡨 rs & rt

**Exceptions:**

* Negative flag will be affected by $rd’s parity bit
* Zero flag will be set if $rd is 0x0

**Programming Notes:**

None

And – Example and

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| R  **00\_0000** | **1\_0000** | **1\_0010** | **1\_0110** | **0\_0000** | **10\_0100** |

6 5 5 5 5 6

$s6 = 0x0000\_0000

$s0 = 0x0123\_FFFF

$s2 = 0x6745\_34AB

and $s6, $s0, $s2

$s0 & $s2 = 0x0101\_34AB

$s6 🡨 0x0101\_34AB

Or or

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| R  **0x00** | **rs** | **rt** | **rd** | **0x00** | **0x25** |

6 5 5 5 5 6

**Format:** or $rd, $rs, $rt

**Purpose:**

Bitwise or two registers with each other and store the result to a register

**Description:** rd 🡨 rs or rt

This operation will bitwise or $rs with $rt and store the result into register $rd.

**Restrictions:**

* rd cannot be a zero register

**Operation:**

rd 🡨 rs | rt

**Exceptions:**

* Negative flag will be affected by $rd’s parity bit
* Zero flag will be set if $rd is 0x0

**Programming Notes:**

None

Or – Example or

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| R  **00\_0000** | **1\_0001** | **1\_0010** | **1\_0100** | **0\_0000** | **10\_0101** |

6 5 5 5 5 6

$s4 = 0x0000\_0000

$s1 = 0xFFFF\_76BE

$s2 = 0x9234\_5377

or $s4, $s1, $s2

$s1 | $s2 = 0x9234\_5256

$s4 🡨 0x9234\_5256

Exclusive Or xor

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| R  **0x00** | **rs** | **rt** | **rd** | **0x00** | **0x26** |

6 5 5 5 5 6

**Format:** xor $rd, $rs, $rt

**Purpose:**

Exclusive or two registers with each other and store the result to a register

**Description:** rd 🡨 rs xor rt

This operation will exclusive or $rs with $rt and store the result into register $rd.

**Restrictions:**

* rd cannot be a zero register

**Operation:**

rd 🡨 rs ^ rt

**Exceptions:**

* Negative flag will be affected by $rd’s parity bit
* Zero flag will be set if $rd is 0x0

**Programming Notes:**

None

Exclusive Or - Example xor

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| R  **00\_0000** | **1\_0011** | **1\_0101** | **1\_0000** | **0\_0000** | **10\_0110** |

6 5 5 5 5 6

$s0 = 0x0000\_0000

$s3 = 0xFFFF\_76BE

$s5 = 0x9234\_5377

xor $s0, $s3, $s5

$s3 ^ $s5 = 0x6DCB\_25C9

$s0 🡨 0x6DCB\_25C9

Not Or nor

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| R  **0x00** | **rs** | **rt** | **rd** | **0x00** | **0x27** |

6 5 5 5 5 6

**Format:** nor $rd, $rs, $rt

**Purpose:**

Not or two registers with each other and store the result to a register

**Description:** rd 🡨 rs nor rt

This operation will not or $rs with $rt and store the result into register $rd.

**Restrictions:**

* rd cannot be a zero register

**Operation:**

rd 🡨 ! ( rs | rt )

**Exceptions:**

* Negative flag will be affected by $rd’s parity bit
* Zero flag will be set if $rd is 0x0

**Programming Notes:**

None

Not Or – Example nor

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| R  **00\_0000** | **1\_0010** | **1\_0011** | **1\_0010** | **0\_0000** | **10\_0111** |

6 5 5 5 5 6

$s2 = 0x0126\_FEF3

$s3 = 0x9234\_5377

nor $s2, $s2, $s3

~($s2 | $s3) = 0x6CC9\_0008

$s2 🡨 0x6CC9\_0008

Signed Set if Less Than slt

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| R  **0x00** | **rs** | **rt** | **rd** | **0x00** | **0x2A** |

6 5 5 5 5 6

**Format:** slt $rd, $rs, $rt

**Purpose:**

Check if a register is less than another register and store the result to a register (signed)

**Description:** rd 🡨 rs < rt

This operation will check if $rs is less than $rt and store the result into register $rd (signed).

**Restrictions:**

* rd cannot be a zero register

**Operation:**

if ( ( rs(31) == 1’b1 ) && ( rt(31) == 1’b0 ) )

rd 🡨 32’h1

if ( ( rs(31) == 1’b0 ) && ( rt(31) == 1’b1 ) )

rd 🡨 32’h0

else ( rs(31) == rt(31) )

rd 🡨 rs(30:0) < rt(30:0)

**Exceptions:**

* Zero flag will be set if $rd is 0x0

**Programming Notes:**

None

Signed Set if Less Than - Example slt

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| R  **00\_0000** | **1\_0000** | **1\_0100** | **0\_1001** | **0\_0000** | **10\_1010** |

6 5 5 5 5 6

$t1 = 0x0000\_0000

$s0 = 0x7563\_FABB

$s4 = 0xF444\_0000

slt $t1, $s0, $s4

$s0 < $s4 = 0x0

$t1 🡨 0x0000\_0000

Unsigned Set if Less Than sltu

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| R  **0x00** | **rs** | **rt** | **rd** | **0x00** | **0x2B** |

6 5 5 5 5 6

**Format:** sltu $rd, $rs, $rt

**Purpose:**

Check if a register is less than another register and store the result to a register (unsigned)

**Description:** rd 🡨 rs < rt

This operation will check if $rs is less than $rt and store the result into register $rd (unsigned).

**Restrictions:**

* rd cannot be a zero register

**Operation:**

rd 🡨 rs < rt

**Exceptions:**

* Zero flag will be set if $rd is 0x0

**Programming Notes:**

None

Unsigned Set if Less Than - Example sltu

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| R  **00\_0000** | **1\_0010** | **1\_0100** | **0\_1010** | **0\_0000** | **10\_1011** |

6 5 5 5 5 6

$t2 = 0x0000\_0000

$s2 = 0x4412\_0098

$s4 = 0x993B\_AABC

sltu $t1, $s0, $s4

$s2 < $s3 = 0x1

$s2 🡨 0x0000\_0001

Break break

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| R  **0x00** | **0x00** | **0x00** | **0x00** | **0x00** | **0x0D** |

6 5 5 5 5 6

**Format:** break

**Purpose:** The halt operation of the MIPS Processor

**Description:**

The instruction stops all the operations of the processor.

**Restrictions:**

None

**Operation:**

When the control unit detects 0x0D on the last 6 bits of the Instruction Register, all operation of the Processor stops.

**Exceptions:**

None

**Programming Notes:**

None

Set Interrupt Enable setie

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| R  **0x00** | **0x00** | **0x00** | **0x00** | **0x00** | **0x1F** |

6 5 5 5 5 6

**Format:** setie

**Purpose:** Enables the processor to check for Interrupts.

**Description:**

The instruction sets the processor to check for Interrupts

**Restrictions:**

None

**Operation:**

When the control unit detects 0x1F on the last 6 bits of the Instruction Register, the processor is activated to detect Interrupts.

**Exceptions:**

None

**Programming Notes:**

None

Branch on Equal beq

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |
| --- | --- | --- | --- |
| I  **00\_0100** | **rs** | **rt** | **16-bit immediate** |

6 5 5 16

**Format:** beq $rt, $rs, ‘branch address’

**Purpose:** Conditional Branch that jumps to the 16-bit immediate value address.

**Description:** if(rs == rt), branch

If the contents of register $rs and register $rt are equal, branch to the target address ’16-bit immediate value’.

**Restrictions:**

None

**Operation:**

if(rs == rt)  
 PC 🡨 PC + 4 + ’16-bit immediate value’

**Exceptions:**

None

**Programming Notes:**

None

Branch on Equal – Example beq

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |
| --- | --- | --- | --- |
| I  **00\_0100** | **0\_0111** | **0\_0110** | **1111\_1111\_1111\_1100** |

6 5 5 16

beq $r7, $r6, Label

R[r7] = 0x7070\_7070

R[r6] = 0x7070\_7070

Label = 0xFFFC

Since $rs is equal to $rt, PC 🡨 0xFFFC

Branch not Equal bne

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |
| --- | --- | --- | --- |
| I  **00\_0101** | **rs** | **rt** | **16-bit immediate** |

6 5 5 16

**Format:** bne $rt, $rs, ‘branch address’

**Purpose:** Conditional Branch that jumps to the 16-bit immediate value address.

**Description:** if(rs != rt), branch

If the contents of register $rs and register $rt are not equal, branch to the target address ’16-bit immediate value’.

**Restrictions:**

None

**Operation:**

if(rs != rt)  
 PC 🡨’16-bit immediate value’

**Exceptions:**

None

**Programming Notes:**

None

Branch not Equal – Example bne

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |
| --- | --- | --- | --- |
| I  **00\_0101** | **0\_0010** | **0\_0011** | **0000\_0000\_0111\_1100** |

6 5 5 16

bne $r2, $r3, Label

R[r7] = 0x7070\_7070

R[r6] = 0xFFFF\_ABCD

Label = 0x007C

Since $rs is not equal to $rt, PC 🡨 0x007C

Branch on Less Than or Equal to Zero blez

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |
| --- | --- | --- | --- |
| I  **00\_0110** | **rs** | **0\_0000** | **16-bit immediate** |

6 5 5 16

**Format:** blez $rs, ‘branch address’

**Purpose:** Conditional Branch that jumps to the 16-bit immediate value address.

**Description:** if(rs  0), branch

If the value of register $rs is less than or equal to zero, branch to the target address   
’16-bit immediate value’.

**Restrictions:**

None

**Operation:**

if(rs  0)  
 PC 🡨’16-bit immediate value’

**Exceptions:**

None

**Programming Notes:**

None

Branch on Less Than or Equal to Zero – Example blez

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |
| --- | --- | --- | --- |
| I  **00\_0110** | **0\_0111** | **0\_0000** | **0000\_0111\_0000\_0101** |

6 5 5 16

blez $r1, Label

R[r1] = 0xF010\_1010

Label = 0x0705

Since $r1 is less than zero, PC 🡨 0x0705

Branch on Greater Than Zero bgtz

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |
| --- | --- | --- | --- |
| I  **00\_0111** | **rs** | **0x00** | **16-bit immediate** |

6 5 5 16

**Format:** bgtz $rs, ‘branch address’

**Purpose:** Conditional Branch that jumps to the 16-bit immediate value address.

**Description:** if(rs  0), branch

If the value of register $rs is greater than zero, branch to the target address   
’16-bit immediate value’.

**Restrictions:**

None

**Operation:**

if(rs  0)  
 PC 🡨’16-bit immediate value’

**Exceptions:**

None

**Programming Notes:**

None

Branch on Greater than Zero – Example bgtz

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |
| --- | --- | --- | --- |
| I  **00\_0111** | **0\_0111** | **0\_0000** | **1111\_1111\_1111\_1100** |

6 5 5 16

bgtz $r7, Label

R[r7] = 0x0727\_2727

Label = 0x0027

Since $rs is greater than zero, PC 🡨 0x0027

Add Immediate addi

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |
| --- | --- | --- | --- |
| I  **00\_1000** | **rs** | **rt** | **16-bit immediate** |

6 5 5 16

**Format:** addi $rt, $rs, ’16-bit immediate value’

**Purpose:** Add instruction that takes a 16-bit immediate value, add it with register $rs and store to register $rt

**Description:** rt 🡨 rs + ‘16-bit immediate’

The operation adds the register $rt with the 16-bit immediate value and store the result   
into the register $rs.

**Restrictions:**

None

**Operation:**

rt = rs + ’16-bit immediate value’

**Exceptions:**

* Integer Overflow Flag is affected if result is wrong.
* Carry Flag is affected if result exceeds the max 32-bit value (0xFFFF\_FFFF)

**Programming Notes:**

None

Add Immediate – Example addi

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |
| --- | --- | --- | --- |
| I  **00\_0100** | **0\_0011** | **0\_0101** | **1111\_1111\_1110\_1110** |

6 5 5 16

addi $r5, $r3, 0x0002

R[r5] = …

R[r3] = 0xFEEB\_1234

Immediate Value = 0x0002

R3 -> FEEB\_1234  
Imm -> 0000\_0002 +  
R5 -> FEEB\_1236

Register $r5 gets the value 0xFEEB\_1236

R[r5] = 0xFEEB\_1236

Set Less Than Immediate slti

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |
| --- | --- | --- | --- |
| I  **00\_1010** | **rs** | **rt** | **16-bit immediate** |

6 5 5 16

**Format:** slti $rt, $rs, ’16-bit immediate value’

**Purpose:** Boolean operation that checks if the register $rs is less than the ’16-bit immediate value’ and sets register $rt to 1 if it is and 0 otherwise.

**Description:** rt 🡨 if(rs < ’16-bit immediate value’)

If register $rs is less than the ’16-bit immediate value’, register $rt is 1, otherwise 0.

**Restrictions:**

None

**Operation:**

if(rs < ’16-bit immediate value’)

rt = 1

else

rt = 0

**Exceptions:**

None

**Programming Notes:**

None

Set Less Than Immediate – Example slti

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |
| --- | --- | --- | --- |
| I  **00\_1010** | **0\_0010** | **0\_0001** | **0111\_1111\_1111\_1100** |

6 5 5 16

slti $r2, $r1, 0x7FFC

R[r2] = …

R[r1] = 0x0000\_2070

Immediate Value = 0x7FFC

Since value of register $r1 is than the immediate value, register $r2 gets 32-bit value of 1

R[r2] = 0x0000\_0001

Set Less Than Immediate Unsigned sltiu

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |
| --- | --- | --- | --- |
| I  **00\_1011** | **rs** | **rt** | **16-bit immediate** |

6 5 5 16

**Format:** sltiu $rt, $rs, ’16-bit immediate value’

**Purpose:** Boolean operation that checks if the register $rs is less than the ’16-bit immediate value’ and sets register $rt to 1 if it is and 0 otherwise. (UNSIGNED)

**Description:** rt 🡨 if(rs < ’16-bit immediate value’)

If register $rs is less than the ’16-bit immediate value’, register $rt is 1, otherwise 0.

**Restrictions:**

None

**Operation:**

if(rs < ’16-bit immediate value’)

rt = 1

else

rt = 0

**Exceptions:**

None

**Programming Notes:**

None

Set Less Than Immediate Unsigned – Example sltiu

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |
| --- | --- | --- | --- |
| I  **00\_1011** | **0\_0011** | **0\_0001** | **0111\_0111\_0111\_0111** |

6 5 5 16

sltiu $r3, $r1, 0x7777

R[r3] = …

R[r1] = 0x0000\_2070

Immediate Value = 0x7777

Since value of register $r1 is than the immediate value, register $r3 gets 32-bit value of 1

R[r3] = 0x0000\_0001

AND Immediate andi

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |
| --- | --- | --- | --- |
| I  **00\_1100** | **rs** | **rt** | **16-bit immediate** |

6 5 5 16

**Format:** andi $rt, $rs, ’16-bit immediate value’

**Purpose:** Logical AND between register $rs and the ’16-bit immediate value’

**Description:** rt 🡨 rs AND ’16-bit immediate value’

The operation does a bitwise logical AND between register $rs and the zero extended to the left ’16-bit immediate value’ and stores the result into register $rt

**Restrictions:**

None

**Operation:**

rt = rs & {16’h0, ’16-bit immediate value’}

**Exceptions:**

None

**Programming Notes:**

None

AND Immediate – Example andi

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |
| --- | --- | --- | --- |
| I  **00\_1100** | **0\_0001** | **0\_0111** | **0111\_0010\_0111\_0010** |

6 5 5 16

andi $r7, $r1, 0x7272

R[r7] = …

R[r1] = 0x5A5A\_5A5A

Immediate Value = 0x7272

R1 -> 5A5A\_5A5A  
Imm -> 0000\_7272 AND  
R7 -> 0000\_5252

Register $r7 gets the value 0x0000\_5252

R[r7] = 0x0000\_5252

OR Immediate ori

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |
| --- | --- | --- | --- |
| I  **00\_1101** | **rs** | **rt** | **16-bit immediate** |

6 5 5 16

**Format:** ori $rt, $rs, ’16-bit immediate value’

**Purpose:** Logical OR between register $rs and the ’16-bit immediate value’

**Description:** rt 🡨 rs OR ’16-bit immediate value’

The operation does a bitwise logical OR between register $rs and the ’16-bit immediate value’ and stores the result into register $rt

**Restrictions:**

None

**Operation:**

rt = rs | {16’h0, ’16-bit immediate value’}

**Exceptions:**

None

**Programming Notes:**

None

OR Immediate – Example ori

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |
| --- | --- | --- | --- |
| I  **00\_1101** | **0\_0010** | **0\_0101** | **0100\_1011\_0100\_1011** |

6 5 5 16

ori $r5, $r2, 0x2B2B

R[r5] = …

R[r2] = 0xB2B2\_B2B2

Immediate Value = 0x2B2B

R2 -> B2B2\_B4B4  
Imm -> 0000\_4B4B OR  
R5 -> 0000\_FFFF

Register $r5 gets the value 0x0000\_FFFF

R[r5] = 0x0000\_FFFF

Exclusive OR Immediate xori

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |
| --- | --- | --- | --- |
| I  **00\_1110** | **Rs** | **rt** | **16-bit immediate** |

6 5 5 16

**Format:** xori $rt, $rs, ’16-bit immediate value’

**Purpose:** Logical XOR between register $rs and the ’16-bit immediate value’

**Description:** rt 🡨 rs XOR ’16-bit immediate value’

The operation does a bitwise logical XOR between register $rs and the ’16-bit immediate value’ and stores the result into register $rt

**Restrictions:**

None

**Operation:**

rt = rs ^ {16’h0, ’16-bit immediate value’}

**Exceptions:**

None

**Programming Notes:**

None

Exclusive OR Immediate – Example xori

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |
| --- | --- | --- | --- |
| I  **00\_1110** | **0\_1000** | **0\_0010** | **0000\_0000\_1111\_1111** |

6 5 5 16

xori $r2, $r8, 0x00FF

R[r2] = …

R[r8] = 0x2F2F\_1111

Immediate Value = 0x00FF

R8 -> 2F2F\_3333  
Imm -> 0000\_00FF XOR  
R2 -> 2F2F\_33CC

Register $r2 gets the value 0x2F2F\_33CC

R[r2] = 0x2F2F\_33CC

Load Upper Immediate lui

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |
| --- | --- | --- | --- |
| I  **00\_1111** | **0\_0000** | **rt** | **16-bit immediate** |

6 5 5 16

**Format:** lui $rt, ’16-bit immediate value’

**Purpose:** Loads register $rt with the 16-bit immediate value on the upper 16 bits

**Description:** rt 🡨 {’16-bit immediate value, 16’h0}

The operation stores the 16-bit immediate value and 16 bits of zeros into the register $rt.

**Restrictions:**

None

**Operation:**

rt = rs & {’16-bit immediate value’, 16’h0}

**Exceptions:**

None

**Programming Notes:**

None

Load Upper Immediate – Example lui

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |
| --- | --- | --- | --- |
| I  **00\_1111** | **0\_0000** | **rt** | **0111\_1111\_1111\_0111** |

6 5 5 16

lui $r3, 0x7FF7

R[r3] = …

Immediate Value = 0x7FF7

Register $r3 loads the immediate value as the upper bits and zero fill.

R[r3] = 0x7FF7\_0000

Load Word lw

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |
| --- | --- | --- | --- |
| I  **10\_0011** | **rs** | **rt** | **16-bit immediate** |

6 5 5 16

**Format:** lw $rt, ’16-bit immediate value’($rs)

**Purpose:** Loads word from memory into a register

**Description:** rs is the base address

rt 🡨 M[rs + 16-bit immediate value]

The operation loads the value from a specific memory address specified by the base(register $rs) + the offset(’16-bit immediate value’) into the register $rt

**Restrictions:**

None

**Operation:**

rt = M[rs + 16-bit immediate value]

**Exceptions:**

None

**Programming Notes:**

None

Load Word – Example lw

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |
| --- | --- | --- | --- |
| I  **10\_0011** | **0\_0010** | **0\_0001** | **0000\_0000\_0000\_0100** |

6 5 5 16

lw $r1, 0x0004($r2)

R[r1] = …

R[r2] = 0x2111\_FBC0

Immediate Value = 0x0004

r2 -> 2111\_FBC0  
Imm -> 0000\_0004 +  
Adr -> 2111\_FBC4

|  |  |
| --- | --- |
| 2111\_FBC0 🡪 | ……… |
| 2111\_FBC4 🡪 | 1234\_5678 |
| 2111\_FBC8 🡪 | ……... |

Register $r1 loads the value from memory of address [0x2111\_FBC4]

R[r1] = 0x1234\_5678

Store Word sw

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |
| --- | --- | --- | --- |
| I  **10\_1011** | **rs** | **rt** | **16-bit immediate** |

6 5 5 16

**Format:** sw $rt, ’16-bit immediate value’($rs)

**Purpose:** Stores word from register into memory

**Description:** rs is the base address

M[rs + 16-bit immediate value] 🡨rt

The operation stores the value from a register $rt into a specific memory address specified by the base(register $rs) + the offset(’16-bit immediate value’)

**Restrictions:**

None

**Operation:**

M[rs + 16-bit immediate value] = rt

**Exceptions:**

None

**Programming Notes:**

None

Store Word – Example sw

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |
| --- | --- | --- | --- |
| I  **10\_1011** | **0\_0111** | **0\_0110** | **0000\_0000\_0000\_1000** |

6 5 5 16

sw $r6, 0x1234($r7)

R[r6] = 0x2222\_7777

R[r7] = 0xB0B0\_1234

Immediate Value = 0x0008

r7 -> B0B0\_1234  
Imm -> 0000\_0008 +  
Adr -> B0B0\_123C

Register $r6 is stored into the Memory Address (B0B0\_123C)

|  |  |
| --- | --- |
| B0B0\_1238 🡪 | ……… |
| B0B0\_123C 🡪 | 2222\_7777 |
| B0B0\_123F 🡪 | ……... |

M[Adr] = 0x2222\_7777

Jump j

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |
| --- | --- |
| J  **00\_0010** | **26-bit Jump Address** |

6 26

**Format:** j ’26-bit Jump Address’

**Purpose:** Unconditional Jump to the 26-bit Jump Address

**Description:** PC 🡨 ’26-bit Jump Address’

Jumps to the target address specified by the ’26-bit Jump Address’

**Restrictions:**

None

**Operation:**

PC = ’26-bit Jump Address’

**Exceptions:**

None

**Programming Notes:**

None

Jump - Example j

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |
| --- | --- |
| J  **00\_0010** | **10\_1110\_1101\_1100\_1011\_1010\_1000** |

6 26

j 0x02ED\_CBA8

Program Counter changes

Program Counter = 0x02ED\_CBA8

|  |  |  |
| --- | --- | --- |
|  | 02ED\_CBA4 | ……… |
| PC 🡪 | 02ED\_CBA8 | ……… |
|  | 02ED\_CBAC | ……… |

Jump and Link jal

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |
| --- | --- |
| J  **00\_0011** | **26-bit Jump Address** |

6 26

**Format:** jal ’26-bit Jump Address’

**Purpose:** Unconditional Jump to the 26-bit Jump Address

**Description:** PC 🡨 ’26-bit Jump Address’

Jumps to the target address specified by the ’26-bit Jump Address’

**Restrictions:**

None

**Operation:**

PC = ’26-bit Jump Address’

**Exceptions:**

None

**Programming Notes:**

None

Jump and Link – Example jal

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |
| --- | --- |
| J  **00\_0011** | **26-bit Jump Address** |

6 26

jal 0x02ED\_CBA8

Program Counter changes

Program Counter = 0x02ED\_CBA8

|  |  |  |
| --- | --- | --- |
|  | 02ED\_CBA4 | ……… |
| PC 🡪 | 02ED\_CBA8 | ……… |
|  | 02ED\_CBAC | ……… |

Input input

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |
| --- | --- | --- | --- |
| I  **01\_1100** | **rs** | **rt** | **16-bit immediate** |

6 5 5 16

**Format:** input $rt, ’16-bit immediate value’($rs)

**Purpose:** Loads word from I/O into a register

**Description:** rs is the base address

rt 🡨 IO[rs + 16-bit immediate value]

The operation loads the value from a specific IO memory address specified by the base(register $rs) + the offset(’16-bit immediate value’) into the register $rt

**Restrictions:**

None

**Operation:**

rt = IO[rs + 16-bit immediate value]

**Exceptions:**

None

**Programming Notes:**

None

Input – Example input

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |
| --- | --- | --- | --- |
| I  **01\_1100** | **1\_0001** | **1\_0000** | **0000\_0000\_0000\_1000** |

6 5 5 16

input $s0, 0x0008($s1)

R[s1] = 0x1111\_2224

Immediate Value = 0x0008

s1 -> 1111\_2224  
Imm -> 0000\_0008 +  
Adr -> 1111\_222C

|  |  |
| --- | --- |
| 1111\_2228 🡪 | ……… |
| 1111\_222C 🡪 | FFFE\_ABCD |
| 1111\_2230 🡪 | ……... |

Register $s0 loads the value from the IO memory of address [0x1111\_222C]

R[s0] 🡨IO[0x1111\_222C]

R[s0] 🡨0xFFFE\_ABCD

Output output

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |
| --- | --- | --- | --- |
| I  **01\_1101** | **rs** | **rt** | **16-bit immediate** |

6 5 5 16

**Format:** output $rt, ’16-bit immediate value’($rs)

**Purpose:** Stores word from register into the I/O

**Description:** rs is the base address

IO[rs + 16-bit immediate value] 🡨rt

The operation stores the value from a register $rt into a specific IO memory address specified by the base(register $rs) + the offset(’16-bit immediate value’)

**Restrictions:**

None

**Operation:**

IO[rs + 16-bit immediate value] = rt

**Exceptions:**

None

**Programming Notes:**

None

Output – Example output

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |
| --- | --- | --- | --- |
| I  **01\_1101** | **0\_1000** | **0\_1100** | **0000\_0000\_0001\_0000** |

6 5 5 16

output $t4, 0x0010($t0)

R[t0] = 0x001F\_EEDC

R[t4] = 0xB0B0\_B0B0

Immediate Value = 0x0010

t0 -> 001F\_EEDC  
Imm -> 0000\_0010 +  
Adr -> 001F\_EEEC

Register $r6 is stored into the Memory Address (B0B0\_123C)

|  |  |
| --- | --- |
| 001F\_EEE8 🡪 | ……… |
| 001F\_EEEC 🡪 | B0B0\_B0B0 |
| 001F\_EEF0 🡪 | ……... |

M[0x001F\_EEEC] 🡨 0xB0B0\_B0B0

Return Interrupt reti

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |
| --- | --- |
| J  **01\_1110** | **00\_0000\_0000\_0000\_0000\_0000\_0000** |

6 26

**Format:** reti

**Purpose:** Jumps back to the original PC before the jump to the Interrupt Service Routine

**Description:**

The operation takes the value from the top of stack (M[0x404]), and updates the status flags to the original flags before the Interrupt, decrements stack pointer by 4, then takes the value from the new top of stack (M[0x400]), and jumps to that address, and decrements one more time by 4, and update the stack pointer to 0x3FC.

It takes 6 clock cycles to complete the operation.

**Restrictions:**

None

**Operation:**

1. ALU\_Out 🡨 RS(0x404)
2. D\_in 🡨 dM[ALU\_Out(0x404)] ,
3. I, C, V, N, Z 🡨 D\_in(28’b0, intr, C, V, N, Z),   
   ALU\_Out 🡨 RS(0x404) – 4,   
   RS 🡨 RS(0x404) - 4
4. D\_in 🡨 dM[ALU\_Out(0x400)] ,  
   ALU\_Out 🡨RS(0x400) – 4,
5. PC 🡨 D\_in (dM[0x400])
6. $sp 🡨ALU\_Out(0x3FC)

**Exceptions:**

None

**Programming Notes:**

None

Rotate Left rotleft

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| R  **0x00** | **0x00** | **rt** | **rd** | **shtamt** | **0x2C** |

6 5 5 5 5 6

**Format:** rotleft $rd, $rt, shtamt

**Purpose:**

Execute logical rotate left by specified shift amount.

**Description:** rd 🡨 rt << shtamt

This operation will execute a rotate left to a register $rt by shift amount shtamt and store the result into a destination register $rd.

**Restrictions:**

* rd cannot be a zero register

**Operation:**

rd 🡨 rt << shtamt

**Exceptions:**

* Negative flag will be affected by $rd’s parity bit
* Zero flag will be set if $rd is 0x00
* Compared to Shift Left Logical (SLL), this operation does not affect the carry flag

**Programming Notes:**

None

Rotate Left – Example rotleft

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| R  **00\_0000** | **0000\_0** | **0\_1001** | **0\_1001** | **0\_0010** | **0x2C** |

6 5 5 5 5 6

$t1 = 0xFEDC\_BA98

rotleft $t1, $t1, 2

1111\_1110\_1101\_1100\_1011\_1010\_1001\_1000 11

$t1 🡨 0xFB72\_EA63

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| R  **00\_0000** | **0000\_0** | **0\_1100** | **0\_1000** | **0\_0110** | **0x2C** |

6 5 5 5 5 6

$t4 = 0xABCD\_EF01

rotleft $t0, $t4, 6

1010\_1011\_1100\_1101\_1110\_1111\_0000\_0001 1010\_10

$t0 🡨 0xF37B\_C06A

Rotate Right rotright

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| R  **0x00** | **0x00** | **rt** | **rd** | **shtamt** | **0x2D** |

6 5 5 5 5 6

**Format:** rotright $rd, $rt, shtamt

**Purpose:**

Execute logical rotate right by specified shift amount.

**Description:** rd 🡨 rt >> shtamt

This operation will execute a rotate right to a register $rt by shift amount shtamt and store the result into a destination register $rd.

**Restrictions:**

* rd cannot be a zero register

**Operation:**

rd 🡨 rt >> shtamt

**Exceptions:**

* Negative flag will be affected by $rd’s parity bit
* Zero flag will be set if $rd is 0x00
* Compared to Shift Right Logical (SRL), this operation does not affect the carry flag

**Programming Notes:**

None

Rotate Right – Example rotright

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| R  **00\_0000** | **0000\_0** | **1\_0000** | **1\_0001** | **0\_0100** | **0x2D** |

6 5 5 5 5 6

$s0 = 0x0000\_00CD

rotright $s1, $s0, 4

1101\_0000\_0000\_0000\_0000\_0000\_0000\_1100\_1101

$s1 🡨 0xD000\_000C

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| R  **00\_0000** | **0000\_0** | **1\_0100** | **1\_0000** | **0\_0110** | **0x2D** |

6 5 5 5 5 6

$s4 = 0x1234\_5678

rotright $s0, $s4, 8

0111\_1000\_0001\_0010\_0011\_0100\_0101\_0110\_0111\_1000

$s0 🡨 0x7812\_3456

III. Verilog Implementation

1. Top Level Module

`timescale 1ns / 1ps

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* C E C S 4 4 0 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*

\* File Name: CPU\_test.v

\* Project: CECS 440 Senior Project - Enhanced MIPS Processor

\* Designer: Benjamin Santos and Naoaki Takatsu

\* Email: benjaminsantos@gmx.com

\* naoaki.takatsu@student.csulb.edu

\* Rev. No.: Version 1.3

\* Rev. Date: 11/8/2018

\*

\* Purpose: This is the testbench that will instantiate and verify the

\* correctness of the Central Processing Unit(CPU) with the Data

\* Memory module and the InputOutput.v module.

\*

\* Notes: This module will set an interrupt at 300ns in the middle of

\* the Finite State Machine to test the validity of the

\* interrupt, and goes back to the original Program Counter

\* before the interrupt occurs.

\*

\* Revision Notes:

\* 1.2: (Revised from CPU\_test1.v testbench module)

\* Added IO module for testing the Input and Output

\* instructions

\* 1.3: Changed to work for iMem 14module CPU\_test1;

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/   
module CPU\_test3;

// Inputs

reg clk, reset; //Clock and Reset

reg intr; //Interrupt Request

// Outputs

wire int\_ack; //Interrupt Acknowledge Flag

wire [31:0] D\_MemToInt; //Data from DM to Datapath

wire [31:0] D\_IOToInt; //Data from IO to Datapath

wire [31:0] D\_MemOrIO, Addr; //Data Output from Datapath

wire dm\_cs, dm\_wr, dm\_rd; //Data Memory flags

wire io\_cs, io\_wr, io\_rd; //IO module flags

// Instantiate the Unit Under Tests

CPU cpu(

.clk(clk), .reset(reset), //Clock and Reset

.intr(intr), //Interrupt Request

.int\_ack(int\_ack), //Interrupt Acknowledge

.D\_MemToInt(D\_MemToInt), //Data Input from Memory

.D\_IOToInt(D\_IOToInt), //Data Input from IO

.Addr(Addr), //Data Address output of CPU

.D\_OUT(D\_MemOrIO), //Data Output for Memory or IO

.dm\_cs(dm\_cs), // \

.dm\_rd(dm\_rd), // --> Data Memory Flags

.dm\_wr(dm\_wr), // /

.io\_cs(io\_cs), // \

.io\_rd(io\_rd), // --> I/O Module Flags

.io\_wr(io\_wr) // /

);

DataMemory data\_mem (

.clk(clk), //Clock

.Address( { 20'b0, //Only takes first 12 bits

Addr[11:0]}), // of Address

.D\_In(D\_MemOrIO), //Data Input from CPU

.D\_Out(D\_MemToInt), //Data Output to CPU

.dm\_cs(dm\_cs), // \

.dm\_rd(dm\_rd), // --> Data Memory Flags

.dm\_wr(dm\_wr) // /

);

InputOutput in\_out (

.clk(clk), //Clock

.Address( { 20'b0, //Only takes first 12 bits

Addr[11:0]}), //of Address

.D\_In(D\_MemOrIO), //Data Input from CPU

.D\_Out(D\_IOToInt), //Data Output to CPU

.io\_cs(io\_cs), // \

.io\_rd(io\_rd), // --> Data Memory Flags

.io\_wr(io\_wr), // /

.int\_ack(int\_ack) //Interrupt Acknowledge

);

// Create a 10 ns clock

always #5 clk = ~clk;

initial begin

// Initialize Inputs

clk = 0;

reset = 1;

intr = 0;

$timeformat( -9, 1, " ps", 9 ); //Display time in nanoseconds

// store .dat files into their respective memories (Data and Instruction)

// $readmemh( "dMem\_1.dat", data\_mem.M ); //Test 1

// $readmemh( "iMem\_1.dat", cpu.IU.IM.IMem );

// $readmemh( "dMem\_2.dat", data\_mem.M ); //Test 2

// $readmemh( "iMem\_2.dat", cpu.IU.IM.IMem );

// $readmemh( "dMem\_3.dat", data\_mem.M ); //Test 3

// $readmemh( "iMem\_3.dat", cpu.IU.IM.IMem );

// $readmemh( "dMem\_4.dat", data\_mem.M ); //Test 4

// $readmemh( "iMem\_4.dat", cpu.IU.IM.IMem );

// $readmemh( "dMem\_5.dat", data\_mem.M ); //Test 5

// $readmemh( "iMem\_5.dat", cpu.IU.IM.IMem );

// $readmemh( "dMem\_6.dat", data\_mem.M ); //Test 6

// $readmemh( "iMem\_6.dat", cpu.IU.IM.IMem );

// $readmemh( "dMem\_7.dat", data\_mem.M ); //Test 7

// $readmemh( "iMem\_7.dat", cpu.IU.IM.IMem );

// $readmemh( "dMem\_8.dat", data\_mem.M ); //Test 8

// $readmemh( "iMem\_8.dat", cpu.IU.IM.IMem );

// $readmemh( "dMem\_9.dat", data\_mem.M ); //Test 9

// $readmemh( "iMem\_9.dat", cpu.IU.IM.IMem );

// $readmemh( "dMem\_10.dat", data\_mem.M ); //Test 10

// $readmemh( "iMem\_10.dat", cpu.IU.IM.IMem );

// $readmemh( "dMem\_11.dat", data\_mem.M ); //Test 11

// $readmemh( "iMem\_11.dat", cpu.IU.IM.IMem );

// $readmemh( "dMem\_12.dat", data\_mem.M ); //Test 12

// $readmemh( "iMem\_12.dat", cpu.IU.IM.IMem );

// $readmemh( "dMem\_13.dat", data\_mem.M ); //Test 13

// $readmemh( "iMem\_13.dat", cpu.IU.IM.IMem );

$readmemh( "dMem\_14.dat", data\_mem.M ); //Test 14

$readmemh( "iMem\_14.dat", cpu.IU.IM.IMem );

// $readmemh( "dMem\_rotleft.dat", data\_mem.M ); //ROTLEFT Instruction

// $readmemh( "iMem\_rotleft.dat", cpu.IU.IM.IMem );

// $readmemh( "dMem\_rotright.dat", data\_mem.M );//ROTRIGHT Instruction

// $readmemh( "iMem\_rotright.dat", cpu.IU.IM.IMem );

// Wait 100 ns for global reset to finish

#100;

reset = 0;

//Set interrupt to 1 at 150ns

#300;

intr = 1;

@(posedge int\_ack)

intr = 0;

$display(" "); $display(" ");  
 $display("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

$display(" CECS 440 - MIPS ISA Control Unit (Test for iM 1-14) ");

$display("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

$display(" ");

end //initial

endmodule

CPU

`timescale 1ns / 1ps

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* C E C S 4 4 0 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*

\* File Name: CPU.v

\* Project: CECS 440 Senior Project - Enhanced MIPS Processor

\* Designer: Benjamin Santos and Naoaki Takatsu

\* Email: benjaminsantos@gmx.com

\* naoaki.takatsu@student.csulb.edu

\* Rev. No.: Version 1.2

\* Rev. Date: 11/8/2018

\*

\* Purpose: The Central Processing Unit(CPU) module has the Control

\* Unit, Instruction Unit and the Datpath modules.

\*

\* Notes: The CPU can be connected to the DataMemory module and the

\* InputOutput module that can be written to and read from

\* according to the Instruction from the Instruction Memory

\* The Instruction Memory is instantiated using $readmemh on the

\* testbench module (CPU\_test1.v)

\*

\* Revision Notes:

\* 1.2: Added IO flag outputs (io\_cs, io\_rd, io\_wr)

\* Added wires M\_sel - Determines data going into Datapath

\* DY - Input for DY reg of the Datapath

\* 1.3: Added wires D\_val - Value from D\_OUT of Datapath

\* stat\_sel

\* - Determines the Data type being

\* output by the CPU

\* S\_sel - Determines where register RS in the

\* Datapath is getting Data from

\* D\_in - Register D\_in from the Datapath now

\* connects to the Control Unit to

\* acquire the status flags from the

\* stack \*

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

module CPU(clk, reset, //Clock and Reset

intr, int\_ack, //Interrupt Flags

D\_MemToInt, D\_IOToInt, //Data Into IDP

Addr, D\_OUT, //Data going to Memory

dm\_cs, dm\_rd, dm\_wr, //Data Memory Flags

io\_cs, io\_rd, io\_wr //IO Module Flags

);

input clk, reset, intr;

input [31:0] D\_MemToInt, D\_IOToInt;

output int\_ack;

output [31:0] Addr, D\_OUT;

output dm\_cs, dm\_rd, dm\_wr;

output io\_cs, io\_rd, io\_wr;

//Wires

wire N, Z, C, V;

wire [31:0] IR\_out;

wire [31:0] PC\_IUtoID, SE\_16;

wire [31:0] DY; //Added for Rev 1.2

//MCU control words(control words of state machine)

wire [4:0] FS; //5-bit control word

wire [2:0] Y\_Sel; //3-bit control word

wire [1:0] pc\_sel, DA\_sel; //2-bit control words

wire pc\_ld, pc\_inc, ir\_ld, im\_cs, im\_rd, im\_wr,

D\_En, T\_sel, HILO\_LD;

//Added for Rev 1.2

wire M\_sel;

//Added for Rev 1.3

wire [31:0] D\_val; //D\_OUT of Datapath

wire [1:0] stat\_sel; //0 - D\_val from IDP, 1 - PC, 2 - Status Flags

wire S\_sel; //0 - S from RegFile, 1 - Y\_lo from ALU

wire [31:0] D\_in; //D\_in register from IDP to the CU

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Instantiate Control Unit, Instruction Unit and Datapath

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//Control Unit

MCU mips\_ctrl(

.sys\_clk(clk), .reset(reset),//Clock and Reset

.intr(intr), //Interrupt Request

.c(C), .n(N), .z(Z), .v(V), //ALU status Flags

.IR(IR\_out), //Instruction Register input

.int\_ack(int\_ack), //output to I/O subsystem

.FS(FS), //Function Select

.ir\_ld(ir\_ld), //IR Laod Enable

.pc\_sel(pc\_sel), // \

.pc\_ld(pc\_ld), // --> PC Flags

.pc\_inc(pc\_inc), // /

.im\_cs(im\_cs), // \

.im\_rd(im\_rd), // --> Instruction Memory Flags

.im\_wr(im\_wr), // /

.D\_En(D\_En), // \

.DA\_sel(DA\_sel), // \

.T\_sel(T\_Sel), // ---> Datapath Flags

.HILO\_ld(HILO\_LD), // /

.Y\_sel(Y\_Sel), // /

.dm\_cs(dm\_cs), // \

.dm\_rd(dm\_rd), // --> Data Memory Flags

.dm\_wr(dm\_wr), // /

.io\_cs(io\_cs), // \

.io\_rd(io\_rd), // --> I/O Module Flags

.io\_wr(io\_wr), // /

.M\_sel(M\_sel), //DM or IO Select

.stat\_sel(stat\_sel), //CPU Output type select

.S\_sel(S\_sel) //Select for register RS

.D\_in(D\_in) //Register D\_in from IDP

);

//Instruction Unit

Instruction\_Unit IU(

.clk(clk), .reset(reset), //Clock and Reset

.PC\_in(Addr), //Value to change PC

.ir\_ld(ir\_ld), //IR Load Enable

.pc\_sel(pc\_sel), // \

.pc\_ld(pc\_ld), // --> PC Flags

.pc\_inc(pc\_inc), // /

.im\_cs(im\_cs), // \

.im\_rd(im\_rd), // --> Instruction Memory Flags

.im\_wr(), // /

.PC\_out(PC\_IUtoID), //Current PC sending to Datapath

.IR\_out(IR\_out), //Current IR sending to Datapath

.SE\_16(SE\_16) //Sign-Extended IR[15:0]

);

//Datapath

Integer\_Datapath

IDP (

.clk(clk), .reset(reset), //Clock and Reset

.D\_En(D\_En), //Write Enable

.D\_Addr(IR\_out[15:11]), //D - Address

.S\_Addr(IR\_out[25:21]), //S - Address

.T\_Addr(IR\_out[20:16]), //T - Address

.DT(SE\_16), //Sign Extended IR[15:0]

.DA\_sel(DA\_sel), //Register File Write Selector

.T\_Sel(T\_Sel), //RT value Selector

.FS(FS), //Function Select

.N(N), .Z(Z), .C(C), .V(V), //ALU Status Flags

.HILO\_LD(HILO\_LD), //Enable for HI and LO

.DY(DY), //Value of Memory to D\_in register

.PC\_in(PC\_IUtoID), //PC value into the Datapath

.Y\_Sel(Y\_Sel), //ALU\_Out Selector

.ALU\_OUT(Addr), //Data Address output of CPU

.D\_OUT(D\_val), //Data Value output of Datapath

.shift\_val(IR\_out[10:6]), //Shift value for Barrel Shift

.S\_sel(S\_sel) //Select for register RS

.D\_in(D\_in) //Register D\_in to CU

);

// Select between Data Memory and IO Module

// (0 - Data Memory, 1 - I/O)

assign DY = (M\_sel == 0) ? D\_MemToInt : D\_IOToInt;

// Type of Data to send to memory

assign D\_OUT = (stat\_sel == 2'b00) ? D\_val : //RT

(stat\_sel == 2'b01) ? PC\_IUtoID : //PC

(stat\_sel == 2'b10) ? {28'b0, intr, C, V, N, Z} : //Status Flags

D\_OUT;

endmodule

MIPS Control Unit (MCU)

`timescale 1ns / 1ps

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* C E C S 4 4 0 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*

\* File Name: MCU.v

\* Project: CECS 440 Senior Project - Enhanced MIPS Processor

\* Designer: Benjamin Santos and Naoaki Takatsu

\* Email: benjaminsantos@gmx.com

\* naoaki.takatsu@student.csulb.edu

\* Rev. No.: Version 1.4

\* Rev. Date: 11/20/2018

\*

\* Purpose: A state machine implementing the MIPS Control Unit (MCU) for the

\* major cycles of fetch, execute and some MIPS instructions from

\* memory, including checking for interrupts.

\*

\* Notes: MCU Control Word:

\*

\* {pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

\* {im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

\* {D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_0\_0\_000; FS = 5'h0;

\* {dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0; int\_ack = 0;

\*

\* Revision Notes:

\* 1.2: Added Instructions: BEQ, BNE, ADDI, SRL, JMP, SRA, SLL,

\* SLT, SLTI, JAL, MULT, MFLO, MFHI,

\* XOR, SLTU, DIV, XORI, SUB, ANDI,

\* SLTIU, BLEZ, BGTZ

\* Added present and next state registers for the Interrupt

\* and ALU status flags

\* 1.3: Added Instructions: SETIE, INPUT, OUTPUT

\* 1.4: Added Enhanced Instructions: ROTLEFT, ROTRIGHT

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

module MCU (sys\_clk, reset, intr, //system inputs

c, n, z, v, //ALU status inputs

IR, //Instruction Register input

int\_ack, //Output to I/O subsystem

FS, //Function Select

pc\_sel, pc\_ld, pc\_inc, ir\_ld, //PC and IR control words

im\_cs, im\_rd, im\_wr, //Inst. Memory control words

D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel, //Datapath Control Words

dm\_cs, dm\_rd, dm\_wr, //Data memory control words

io\_cs, io\_rd, io\_wr, M\_sel, //IO module control words

stat\_sel, //Select for CPU output

S\_sel //RS read select

D\_in //D\_in reg (return flags)

);

input sys\_clk, reset, intr;

input c, n, z, v;

input [31:0] IR;

output reg int\_ack;

//control words

output reg [4:0] FS; //5-bit control word

output reg [2:0] Y\_sel; //3-bit control word

output reg [1:0] pc\_sel, DA\_sel; //2-bit control words

output reg pc\_ld, pc\_inc, ir\_ld, im\_cs, im\_rd, im\_wr,

D\_En, T\_sel, HILO\_ld, dm\_cs, dm\_rd, dm\_wr;

//Added for Rev 1.3

output reg io\_cs, io\_rd, io\_wr, M\_sel;

//Added for Rev 1.4

output reg [1:0] stat\_sel;

output reg S\_sel;

input [31:0] D\_in;

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Internal Data Structures

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//state assignments

parameter

RESET = 00, FETCH = 01, DECODE = 02,

ADD = 10, ADDU = 11, AND = 12, OR = 13, NOR = 14,

ORI = 20, LUI = 21, LW = 22, LW2 = 23, SW = 24,

WB\_alu = 30, WB\_imm = 31, WB\_Din = 32,

WB\_hi = 33, WB\_lo = 34, WB\_mem = 35,

JR1 = 40, JR2 = 41,

// INTR\_1 = 501, INTR\_2 = 502, INTR\_3 = 503,

BREAK = 510,

ILLEGAL\_OP = 511,

//added

BEQ = 50, BEQ2 = 51, //Test 1

BNE = 60, BNE2 = 61,

ADDI = 70, SRL = 71, JMP = 72, //Test 2

SRA = 80, //Test 3

SLL = 90, SLT = 91, //Test 4

SLTI = 92, //Test 5

JAL = 100, JAL2 = 101, //Test 7

MULT = 110, MFLO = 111, MFHI = 112,//Test 8

XOR = 120, SLTU = 121, //Test 9

DIV = 130, //Test 10

XORI = 140, SUB = 141, ANDI = 142,//Test 11

SLTIU = 143,

BLEZ = 150, BLEZ2 = 151, //Test 12

BGTZ = 152, BGTZ2 = 153,

SETIE = 160, //Test 13

INPUT = 161, INPUT2 = 162,

OUTPUT = 163, OUTPUT2= 164,

INTR\_1 = 201, INTR\_2 = 202, INTR\_3 = 203,//Test 14

INTR\_4 = 204, INTR\_5 = 205, INTR\_6 = 206,

INTR\_7 = 207, INTR\_8 = 208,

RETI1 = 171, RETI2 = 172, RETI3 = 173,

RETI4 = 174, RETI5 = 175, RETI6 = 176,

//Enhanced Instructions

ROTLEFT = 180, ROTRIGHT = 181;

//Registers

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

reg psi, psc, psv, psn, psz, //Present state Status Flags

nsi, nsc, nsv, nsn, nsz; //Next state Status Flags

always @(posedge sys\_clk, posedge reset)

if(reset)

{psi, psc, psv, psn, psz} = 5'b0;

else

{psi, psc, psv, psn, psz} = {nsi, nsc, nsv, nsn, nsz};

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// 440 MIPS Control Unit (Finite State Machine)

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//Next State Logic

always @(posedge sys\_clk, posedge reset)

if(reset)

begin

// PC <- 32'b0, R31($ra) <- 32'b0

// NS <- RESET

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b10\_1\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b1\_10\_0\_0\_100; FS = 5'h15;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0; int\_ack = 0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_0;

state = RESET;

end

else

case(state)

FETCH: @(negedge sys\_clk) begin

if(int\_ack == 0 & intr == 1) begin

// control word assignments for 'deasserting' everything

// NS <- INTR\_1

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_0\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

FS = 5'h00; int\_ack = 0;

{stat\_sel, S\_sel} = 3'b00\_0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = INTR\_1;

end

else

begin

if(int\_ack == 1 & intr == 0) int\_ack = 0;

// IR <- iM[PC], PC <- PC + 4

// NS <- DECODE

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_1\_1;

{im\_cs, im\_rd, im\_wr} = 3'b1\_1\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_0\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_0;

FS = 5'h00; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = DECODE;

end

end //fetch

RESET: @(negedge sys\_clk)

begin

// R29($sp) <- ALU\_Out(32'h3FC)

// NS <- FETCH

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b1\_11\_0\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_0;

FS = 5'h00; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = FETCH;

end //reset

DECODE: @(negedge sys\_clk)

begin

if( IR[31:26] == 6'h0 ) //check for MIPS format

begin //R-type format

// RS <- $rs RT <- $rt (default)

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_0\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_0;

FS = 5'h00; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} ={psi, c, v, n, z};

//Decides which state to go to according to IR[5:0]

case ( IR[5:0] )

6'h0D : state = BREAK;

6'h20 : state = ADD;

6'h08 : state = JR1;

6'h02 : state = SRL; //iMem 2 test

6'h03 : state = SRA; //iMem 3 test

6'h00 : state = SLL; //iMem 4 test

6'h2A : state = SLT;

6'h18 : state = MULT; //iMem 8 test

6'h10 : state = MFHI;

6'h12 : state = MFLO;

6'h26 : state = XOR; //iMem 9 test

6'h24 : state = AND;

6'h25 : state = OR;

6'h27 : state = NOR;

6'h2B : state = SLTU;

6'h1A : state = DIV; //iMem 10 test

6'h22 : state = SUB; //iMem 11 test

6'h1F : state = SETIE; //iMem 13 test

6'h2C : state = ROTLEFT; //Enhanced Instructions

6'h2D : state = ROTRIGHT;

default: state = ILLEGAL\_OP;

endcase

end //end of R - type

else

begin //I-type or J-type format

// RS <- $rs, RT <- DT

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_1\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_0;

FS = 5'h00; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} ={psi, c, v, n, z};

//Decides which state to go to according to IR[31:26]

case ( IR[31:26] )

6'h0D : state = ORI;

6'h0F : state = LUI;

6'h2B : state = SW;

6'h04 : {T\_sel, state} = {1'b0, BEQ}; //iMem 1 test

6'h05 : {T\_sel, state} = {1'b0, BNE}; //iMem 1 test

6'h08 : state = ADDI; //iMem 2 test

6'h02 : state = JMP; //iMem 2 test

6'h0A : state = SLTI; //iMem 5 test

6'h23 : state = LW; //iMem 6 test

6'h03 : state = JAL; //iMem 7 test

6'h0E : state = XORI; //iMem 11 test

6'h0C : state = ANDI;

6'h0B : state = SLTIU;

6'h06 : state = BLEZ; //iMem 12 test

6'h07 : state = BGTZ;

6'h1C : state = INPUT; //iMem 13 test

6'h1D : state = OUTPUT;

6'h1E : state = RETI1; //iMem 14 test

default: state = ILLEGAL\_OP;

endcase

end //end of I - type

end //end of DECODE

ADD: @(negedge sys\_clk) //Add (R-type)

begin

// ALU\_Out <- RS($rs) + RT($rt)

// NS <- WB\_alu

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_0\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_0;

FS = 5'h02; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = WB\_alu;

end //ADD

ORI: @(negedge sys\_clk) //Or Immediate (I-type)

begin

// ALU\_Out <- RS($rs) or {16'h0, RT[15:0]}

// NS <- WB\_imm

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_1\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_0;

FS = 5'h17; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = WB\_imm;

end //ORI

LUI: @(negedge sys\_clk) //Load Upper Immediate (I-type)

begin

// ALU\_Out <- { RT[15:0], 16'h0 }

// NS <- WB\_imm

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_1\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_0;

FS = 5'h18; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = WB\_imm;

end //LUI

SW: @(negedge sys\_clk) //Store Word (I-type)

begin

// ALU\_Out <- RS($rs) + RT(se\_16), RT <- $rt

// NS <- WB\_mem

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_1\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_0;

FS = 5'h02; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = WB\_mem;

end //SW

WB\_alu: @(negedge sys\_clk) //Write Back to D\_Addr

begin

// R[rd] <- ALU\_Out

// NS <- FETCH

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b1\_00\_0\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_0;

FS = 5'h00; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = FETCH;

end //WB\_alu

WB\_imm: @(negedge sys\_clk) //Write Back to T\_Addr

begin

// R[rt] <- ALU\_Out

// NS <- FETCH

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b1\_01\_1\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_0;

FS = 5'h00; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = FETCH;

end //WB\_imm

WB\_mem: @(negedge sys\_clk) //Write Back to Memory

begin

// M[ ALU\_Out($rs + se\_16) ] <- RT($rt)

// NS <- FETCH

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_0\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b1\_0\_1;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_0;

FS = 5'h00; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = FETCH;

end //WB\_mem

BREAK: @(negedge sys\_clk)

begin

$display("BREAK INSTRUCTION FETCHED %t", $time);

//control word assignments for 'deasserting' everything

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_0\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

FS = 5'h00; int\_ack = 0;

$display(" R E G I S T E R ' S A F T E R B R E A K");

$display(" ");

Reg\_Dump(); //Dumps Register Values

//Dump\_DMEM();

Dump\_DMEM\_INTR(); //Dumps Values in Memory (0x3F0 - 0x404)

Dump\_IO(); //Dumps Values in IO Memory

Dump\_PCIR();

$display(" ");

$finish; //ends simulation

end //BREAK

ILLEGAL\_OP:

@(negedge sys\_clk)

begin //ILLEGAL\_OP

$display("ILLEGAL OPCODE FETCHED %t", $time);

// control word assignments for 'deasserting everything'

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_0\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

FS = 5'h00; int\_ack = 0;

Reg\_Dump();

Dump\_PCIR(); //Dumps Value of PC and IR

$finish; //ends simulation

end //ILLEGAL\_OP

INTR\_1: @(negedge sys\_clk)

begin

//PC gets address of interrupt vector, Save PC in stack

// RS <- 0x3FC, ALU\_Out <- 0x3FC, R[$ra] <- PC

// NS <- INTR\_2

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_0\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_1;

FS = 5'h15; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = INTR\_2;

end //INTR\_1

INTR\_2: @(negedge sys\_clk)

begin

// ALU\_Out <- RS(0x3FC) + 4, RS <- RS(0x3FC) + 4

// NS <- INTR\_3

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_0\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_1;

FS = 5'h11; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = INTR\_3;

end //INTR\_2

INTR\_3: @(negedge sys\_clk)

begin

// M[ ALU\_Out(0x400) ] <- PC, RS <- ALU\_Out(0x400)

// NS <- INTR\_4

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_0\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b1\_0\_1;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b01\_1;

FS = 5'h00; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = INTR\_4;

end //INTR\_3

INTR\_4: @(negedge sys\_clk)

begin

// ALU\_Out <- RS(0x400) + 4

// NS <- INTR\_5

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_0\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_0;

FS = 5'h11; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = INTR\_5;

end //INTR\_4

INTR\_5: @(negedge sys\_clk)

begin

// M[ ALU\_Out(0x404) ] <- status flags(psi, c, v, n, z)

// $sp <- ALU\_Out(0x404)

// NS <- INTR\_6

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_0\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b1\_0\_1;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b10\_0;

FS = 5'h00; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = INTR\_6;

end //INTR\_5

INTR\_6: @(negedge sys\_clk)

begin

// ALU\_Out <- 0x3FC, RS <- 0x3FC

// NS <- INTR\_7

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_0\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_1;

FS = 5'h15; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = INTR\_7;

end //INTR\_6

INTR\_7: @(negedge sys\_clk)

begin

//Read address of ISR into D\_in

// D\_in <- dM[ALU\_Out(0x3FC)]

// NS <- INTR\_8

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_0\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b1\_1\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_0;

FS = 5'h00; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = INTR\_8;

end //INTR\_7

INTR\_8: @(negedge sys\_clk)

begin

//Reload PC with address of ISR; ack the intr; goto FETCH

// PC <- D\_in( dM[0x3FC] ), int\_ack <- 1

// NS <- FETCH

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b10\_1\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_0\_0\_011;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_0;

FS = 5'h00; int\_ack = 1;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = FETCH;

end //INTR\_8

JR1: @(negedge sys\_clk) //Jump Register (R-type)

begin

// ALU\_Out <- RS($rs)

// NS <- JR2

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_0\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_0;

FS = 5'h00; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = JR2;

end //JR1

JR2: @(negedge sys\_clk)

begin

// PC <- ALU\_Out($rs)

// NS <- FETCH

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b10\_1\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_0\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_0;

FS = 5'h00; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = FETCH;

end //JR2

//Added States for iMem 1-12

//test 1

BEQ: @(negedge sys\_clk) //Branch on Equal (I-type)

begin

// ALU\_Out <- RS($rs) - RT($rt)

// NS <- BEQ2

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_0\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_0;

FS = 5'h03; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = BEQ2;

end //BEQ

BEQ2: @(negedge sys\_clk)

begin

// if(z==1) PC <- ALU\_Out(se\_16), NS <- FETCH

// else NS <- FETCH

if(z == 1) begin //BEQ pass

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_1\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_0\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_0;

FS = 5'h01; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = FETCH;

end

else begin //BEQ fail

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_0\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_0;

FS = 5'h00; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = FETCH;

end

end //BEQ2

BNE: @(negedge sys\_clk) //Branch Not Equal (I-type)

begin

// ALU\_Out <- RS($rs) - RT($rt)

// NS <- BRANCH(pass) OR NS <- FETCH(fail)

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_0\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_0;

FS = 5'h03; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = BNE2;

end //BNE

BNE2: @(negedge sys\_clk)

begin

// if(z==1) PC <- PC + ALU\_Out(se\_16), NS <- FETCH

// else NS <- FETCH

if(z != 1) begin //BNE pass

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_1\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_0\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_0;

FS = 5'h01; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = FETCH;

end

else begin //BNE fail

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_0\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_0;

FS = 5'h00; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = FETCH;

end

end //BNE2

//test 2

ADDI: @(negedge sys\_clk) //Add Immediate (I-type)

begin

// ALU\_Out <- RS($rs) + RT(se\_16), RT <- RT($rt)

// NS <- WB\_imm

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_0\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_0;

FS = 5'h02; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = WB\_imm;

end //ADDI

SRL: @(negedge sys\_clk) //Shift Right Logical (R-type)

begin

// ALU\_Out <- RT($rt) >> IR[10:6]

// NS <- WB\_alu

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_0\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_0;

FS = 5'h0D; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = WB\_alu;

end //SRL

JMP: @(negedge sys\_clk) //Jump (J-type)

begin

// PC <- {PC[31:28], IR[25:0], 2'b00}

// NS <- FETCH

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b01\_1\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_0\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_0;

FS = 5'h00; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = FETCH;

end //JMP

//test 3

SRA: @(negedge sys\_clk) //Shift Right Arithmetic (R-type)

begin

// ALU\_Out <- {$rt[31], %rt[31:1]}

// NS <- WB\_alu

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_0\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_0;

FS = 5'h0E; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = WB\_alu;

end //SRA

//test 4

SLL: @(negedge sys\_clk) //Shift Left Logical (I-type)

begin

// ALU\_Out <- RT($rt) << IR[10:6]

// NS <- WB\_alu

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_0\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_0;

FS = 5'h0C; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = WB\_alu;

end //SLL

SLT: @(negedge sys\_clk) //Set Less Than (R-type)

begin

// if( RS($rs) < RT($rt) ) ALU\_Out <- 1

// else ALU\_Out <- 0

// NS <- WB\_alu

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_0\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_0;

FS = 5'h06; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = WB\_alu;

end //SLT

//test 5

SLTI: @(negedge sys\_clk) //Set Less Than Immediate (I-type)

begin

// if( RS($rs) < RT(se\_16) ) ALU\_Out <- 1

// else ALU\_Out <- 0

// NS <- WB\_imm

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_1\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_0;

FS = 5'h06; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = WB\_imm;

end //SLTI

//test 6

LW: @(negedge sys\_clk) //Load Word (I-type)

begin

// ALU\_Out <- RS($rs) + RT(se\_16)

// NS <- LW2

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_0\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_0;

FS = 5'h02; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = LW2;

end //LW

LW2: @(negedge sys\_clk)

begin

// D\_in <- M[rs + se\_16]

// NS <- WB\_Din

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_0\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b1\_1\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_0;

FS = 5'h00; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = WB\_Din;

end //LW2

WB\_Din: @(negedge sys\_clk) //Write Back to Address of D\_in

begin

// R[rt] <- D\_in( M[rs + se\_16] )

// NS <- FETCH

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b1\_01\_1\_0\_011;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_0;

FS = 5'h00; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = FETCH;

end //WB\_Din

//test 7

JAL: @(negedge sys\_clk) //Jump and Link (J-type)

begin

// R31 <- PC

// NS <- JAL2

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b1\_10\_0\_0\_100;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_0;

FS = 5'h00; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = JAL2;

end //JAL

JAL2: @(negedge sys\_clk)

begin

// PC <- {PC[31:28], IR[25:0], 2'b00}

// NS <- FETCH

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b01\_1\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_0\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_0;

FS = 5'h00; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = FETCH;

end //JAL2

//test 8

MULT: @(negedge sys\_clk) //Multiply (R-type)

begin

// {HI, LO} <- RS($rs) \* RT($rt)

// NS <- FETCH

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_0\_1\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_0;

FS = 5'h1E; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = FETCH;

end //MULT

MFLO: @(negedge sys\_clk) //Move from LO (R-type)

begin

// R[rd] <- LO

// NS <- FETCH

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b1\_00\_0\_0\_010;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_0;

FS = 5'h00; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = FETCH;

end //MFLO

MFHI: @(negedge sys\_clk) //Move from HI (R-type)

begin

// R[rd] <- HI

// NS <- FETCH

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b1\_00\_0\_0\_001;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_0;

FS = 5'h00; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = FETCH;

end //MFHI

//test 9

XOR: @(negedge sys\_clk) //Exclusive OR (R-type)

begin

// ALU\_Out <- RS($rs) ^ RT($rt)

// NS <- WB\_alu

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_0\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_0;

FS = 5'h0A; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = WB\_alu;

end //XOR

AND: @(negedge sys\_clk) //AND (R-type)

begin

// ALU\_Out <- RS($rs) & RT($rt)

// NS <- WB\_alu

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_0\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_0;

FS = 5'h08; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = WB\_alu;

end //AND

OR: @(negedge sys\_clk) //OR (R-type)

begin

// ALU\_Out <- RS($rs) | RT($rt)

// NS <- WB\_alu

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_0\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_0;

FS = 5'h09; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = WB\_alu;

end //OR

NOR: @(negedge sys\_clk) //Not OR (R-type)

begin

// ALU\_Out <- ~( RS($rs) | RT($rt) )

// NS <- WB\_alu

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_0\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_0;

FS = 5'h0B; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = WB\_alu;

end //NOR

SLTU: @(negedge sys\_clk) //Set Less Than Unsigned (R-type)

begin

// if( RS($rs) < RT(%rt) ) ALU\_Out <- 1

// else ALU\_Out <- 0

// NS <- WB\_alu

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_0\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_0;

FS = 5'h07; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = WB\_alu;

end //SLTU

//test 10

DIV: @(negedge sys\_clk) //Divide (R-type)

begin

// {HI, LO} <- RS($rs) / RT($rt)

// NS <- FETCH

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_0\_1\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_0;

FS = 5'h1F; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = FETCH;

end //DIV

//test 11

XORI: @(negedge sys\_clk) //Exclusive OR Immediate (I-type)

begin

// ALU\_Out <- RS($rs) ^ RT(se\_16)

// NS <- WB\_alu

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_1\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_0;

FS = 5'h19; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = WB\_imm;

end //XORI

SUB: @(negedge sys\_clk) //Subtract (R-type)

begin

// ALU\_Out <- RS($rs) - RT($rt)

// NS <- WB\_alu

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_0\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_0;

FS = 5'h03; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = WB\_alu;

end //SUB

ANDI: @(negedge sys\_clk) //AND Immediate (I-type)

begin

// ALU\_Out <- RS($rs) & RT(se\_16)

// NS <- WB\_alu

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_1\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_0;

FS = 5'h16; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = WB\_imm;

end //ANDI

SLTIU: @(negedge sys\_clk) //Set Less Than Immediate Unsigned (I-type)

begin

// if( RS($rs) < RT(%se\_16) ) ALU\_Out <- 1

// else ALU\_Out <- 0

// NS <- WB\_alu

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_1\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_0;

FS = 5'h07; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = WB\_imm;

end //SLTIU

//test 12 BLEZ, BGTZ

BLEZ: @(negedge sys\_clk)//Branch on Less than or Equal to Zero (I type)

begin

// ALU\_Out <- RS($rs) - RT($rt), RT <- DT(se\_16)

// NS <- BLEZ2

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_0\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_0;

FS = 5'h03; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = BLEZ2;

end //BLEZ

BLEZ2: @(negedge sys\_clk)

begin

// if(z==1) or (n==1) PC <- ALU\_Out(se\_16), NS <- FETCH

// else NS <- FETCH

if( (z == 1) | (n == 1) ) begin //BLEZ pass

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_1\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_0\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_0;

FS = 5'h01; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = FETCH;

end

else begin //BLEZ fail

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_0\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_0;

FS = 5'h00; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = FETCH;

end

end //BLEZ2

BGTZ: @(negedge sys\_clk) //Branch on Greater Than Zero (I-type)

begin

// ALU\_Out <- RS($rs) - RT($rt), RT <- DT(se\_16)

// NS <- BGTZ2

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_0\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_0;

FS = 5'h03; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = BGTZ2;

end //BGTZ

BGTZ2: @(negedge sys\_clk)

begin

// if(z==1) PC <- ALU\_Out(se\_16), NS <- FETCH

// else NS <- FETCH

if( (z == 1) | (n == 1) ) begin //BGTZ fail

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_0\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_0;

FS = 5'h00; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = FETCH;

end

else begin //BGTZ pass

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_1\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_0\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_0;

FS = 5'h00; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = FETCH;

end

end //BGTZ2

//test 13

SETIE: @(negedge sys\_clk) //Set Interrupt Enable

begin

// NSI <- 1'b1

// NS <- FETCH

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_0\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_0;

FS = 5'h00; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {1'b1, c, v, n, z};

state = FETCH;

end //SETIE

INPUT: @(negedge sys\_clk) //INPUT (similar to LW)

begin

// ALU\_Out <- RS($rs) + RT(se\_16)

// NS <- INPUT2

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_0\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_0;

FS = 5'h02; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = INPUT2;

end //INPUT

INPUT2: @(negedge sys\_clk) // (similar to LW2)

begin

// D\_in <- I\_O[rs + se\_16]

// NS <- WB\_Din

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_0\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b1\_1\_0\_1;

{stat\_sel, S\_sel} = 3'b00\_0;

FS = 5'h00; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = WB\_Din;

end //INPUT2

OUTPUT: @(negedge sys\_clk) //OUTPUT (similar to SW)

begin

// ALU\_Out <- RS($rs) + RT(se\_16), RT <- $rt

// NS <- OUTPUT2

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_1\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_0;

FS = 5'h02; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = OUTPUT2;

end //OUTPUT

OUTPUT2: @(negedge sys\_clk) // (similar to WB\_mem)

begin

// I\_O[ ALU\_Out($rs + se\_16) ] <- RT($rt)

// NS <- FETCH

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_0\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b1\_0\_1\_0;

{stat\_sel, S\_sel} = 3'b00\_0;

FS = 5'h00; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = FETCH;

end //OUTPUT2

//test 14

RETI1: @(negedge sys\_clk) //Return Interrupt

begin

// ALU\_Out <- RS(0x404)

// NS <- RETI2

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_0\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_1;

FS = 5'h00; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = RETI2;

end //RETI1

RETI2: @(negedge sys\_clk)

begin

// D\_in <- dM[ALU\_Out(0x404)],

// NS <- RETI3

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_0\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b1\_1\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_1;

FS = 5'h00; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = RETI3;

end //RETI2

RETI3: @(negedge sys\_clk)

begin

// I, C, V, N, Z <- D\_in(28'b0, intr, C, V, N, Z),

// ALU\_Out <- RS(0x404) - 4, RS <- 0x404 - 4

// NS <- RETI4

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_0\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_1;

FS = 5'h12; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = D\_in[4:0];

state = RETI4;

end //RETI3

RETI4: @(negedge sys\_clk)

begin

// D\_in <- dM[ALU\_Out(0x400)],

// ALU\_Out <- RS(0x400) - 4, RS <- 0x400 - 4

// NS <- RETI5

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_0\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b1\_1\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_1;

FS = 5'h12; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = RETI5;

end //RETI4

RETI5: @(negedge sys\_clk)

begin

// PC <- D\_in( dM[0x400] ),

// NS <- RETI6

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b10\_1\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_0\_0\_011;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_1;

FS = 5'h00; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = RETI6;

end //RETI5

RETI6: @(negedge sys\_clk)

begin

// $sp <- ALU\_Out(0x3FC), int\_ack = 1,

// NS <- FETCH

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b1\_11\_0\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_1;

FS = 5'h00; int\_ack = 1;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = FETCH;

end //RETI6

//Enhanced Instructions

ROTLEFT: @(negedge sys\_clk)

begin

// ALU\_Out <- RT($rt) (rotate left) IR[10:6]

// NS <- WB\_alu

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_0\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_0;

FS = 5'h1C; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = WB\_alu;

end //ROTLEFT

ROTRIGHT: @(negedge sys\_clk)

begin

// ALU\_Out <- RT($rt) (rotate right) IR[10:6]

// NS <- WB\_alu

{pc\_sel, pc\_ld, pc\_inc, ir\_ld} = 5'b00\_0\_0\_0;

{im\_cs, im\_rd, im\_wr} = 3'b0\_0\_0;

{D\_En, DA\_sel, T\_sel, HILO\_ld, Y\_sel} = 8'b0\_00\_0\_0\_000;

{dm\_cs, dm\_rd, dm\_wr} = 3'b0\_0\_0;

{io\_cs, io\_rd, io\_wr, M\_sel} = 4'b0\_0\_0\_0;

{stat\_sel, S\_sel} = 3'b00\_0;

FS = 5'h1D; int\_ack = 0;

#1 {nsi, nsc, nsv, nsn, nsz} = {psi, c, v, n, z};

state = WB\_alu;

end //ROTRIGHT

endcase //end of FSM logic

// Simulation Tasks

integer i;

//Task to dump values of RegFile

task Reg\_Dump;

//Output register values

for( i = 0; i < 16; i = i + 1 )

@(negedge sys\_clk)

begin

$display( "t=%t, $r%0d = 0x%h || $r%0d = 0x%h",

$time, i, CPU\_test3.cpu.IDP.regfile.regs[i],

i+16, CPU\_test3.cpu.IDP.regfile.regs[i+16]);

end

endtask

//Task to dump values of Memory Locations 0xC0 to 0xFF

task Dump\_DMEM; begin

$display(" "); $display("Memory Locations 0xC0 to 0xFF");

for(i = 12'h0C0; i < 12'h100; i = i + 4)

@(negedge sys\_clk)

$display( "t=%t, M[%h] = 0x%h%h%h%h",

$time, i, CPU\_test3.data\_mem.M[i],

CPU\_test3.data\_mem.M[i+1],

CPU\_test3.data\_mem.M[i+2],

CPU\_test3.data\_mem.M[i+3]);

end

endtask

//Task to dump value of PC and IR

task Dump\_PCIR;

@(negedge sys\_clk)

$display( "t=%t, PC = 0x%h, IR = 0x%h",

$time, CPU\_test3.cpu.IU.PC, CPU\_test3.cpu.IU.IR);

endtask

//Task to dump values from IO Location 0xC0 to 0xFF

task Dump\_IO; begin

$display(" "); $display("IO Memory Locations 0xC0 to 0xFF");

for(i = 12'h0C0; i < 12'h100; i = i + 4)

@(negedge sys\_clk)

$display( "t=%t, IO[%h] = 0x%h%h%h%h",

$time, i, CPU\_test3.in\_out.IO[i],

CPU\_test3.in\_out.IO[i+1],

CPU\_test3.in\_out.IO[i+2],

CPU\_test3.in\_out.IO[i+3]);

end

endtask

//Task to dump value from Memory Location 0x3F0

task Dump\_DMEM\_3F0; begin

$display(" "); $display("Memory Location 0x3F0");

i = 12'h3F0;

@(negedge sys\_clk)

$display( "t=%t, M[%h] = 0x%h%h%h%h",

$time, i, CPU\_test3.data\_mem.M[i],

CPU\_test3.data\_mem.M[i+1],

CPU\_test3.data\_mem.M[i+2],

CPU\_test3.data\_mem.M[i+3]);

end

endtask

//Task to dump values from Memory Locations 0x3F0 to 0x404

task Dump\_DMEM\_INTR; begin

$display(" "); $display("Memory Location 0x3F0 to 0x404");

for(i = 12'h3F0; i < 12'h408; i = i + 4)

@(negedge sys\_clk)

$display( "t=%t, M[%h] = 0x%h%h%h%h",

$time, i, CPU\_test3.data\_mem.M[i],

CPU\_test3.data\_mem.M[i+1],

CPU\_test3.data\_mem.M[i+2],

CPU\_test3.data\_mem.M[i+3]);

end

endtask

endmodule

Instruction Unit (IU)

`timescale 1ns / 1ps

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* C E C S 4 4 0 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*

\* File Name: InstructionUnit.v

\* Project: CECS 440 Senior Project - Enhanced MIPS Processor

\* Designer: Benjamin Santos and Naoaki Takatsu

\* Email: benjaminsantos@gmx.com

\* naoaki.takatsu@student.csulb.edu

\* Rev. No.: Version 1.2

\* Rev. Date: 11/8/2018

\*

\* Purpose: This is the Instruction Unit that holds and controls the

\* Program Counter, Instruction Memory and the Instruction Register.

\* Notes: This module holds the 32-bit Program Counter, the Instruction

\* Memory following the MIPS Instruction Formats, and the

\* 32-bit Instruction Register for the Datapath.

\* IR\_out[25:21] -> S\_Addr for the Datapath

\* IR\_out[20:16] -> T\_Addr for the Datapath

\* IR\_out[15:11] -> D\_Addr for the Datapath

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

module Instruction\_Unit(clk, reset, PC\_in, pc\_sel, pc\_ld, pc\_inc, PC\_out,

im\_cs, im\_wr, im\_rd,

ir\_ld, IR\_out, SE\_16);

input clk, reset; //Clock and Reset

input [31:0] PC\_in; //Value to change PC

input [1:0] pc\_sel; //Select for new PC value

input pc\_ld, pc\_inc; //PC write enable and increment flag

input ir\_ld; //Instruction Register Load Enable

input im\_cs, im\_wr, im\_rd; //Instruction Memory Flags

output [31:0] PC\_out; //Output current PC

output [31:0] IR\_out, //Output current IR

SE\_16; //Output sign extended IR

reg [31:0] PC, IR; //PC and IR register

wire [31:0] IMem\_to\_IR; //Data from I-Memory to IR if ir\_ld is 1

InstructionMemory IM (

.clk(clk), //Clock

.Address( { 20'b0, //Only takes first 12 bits

PC\_out[11:0]}), //of Address

.D\_In(32'h0), //Handled by $readmemh in TB

.im\_cs(im\_cs), // \

.im\_rd(im\_rd), // --> Instruction Memory Flags

.im\_wr(), // /

.D\_Out(IMem\_to\_IR)

);

always @(posedge clk, posedge reset)

if(reset)

{PC, IR} <= 64'b0; //Reset PC and IR values

else

begin

//Program Counter Input Logic

if(pc\_ld)

case(pc\_sel)

2'b00: PC <= PC + { SE\_16[29:0], 2'b00 }; //for Branches

2'b01: PC <= { PC[31:28], IR[25:0], 2'b00 }; //for Jumps

2'b10: PC <= PC\_in; //for Returns

default: PC <= PC;

endcase

else

if(pc\_inc) //Increment PC

PC <= PC + 32'd4;

else

PC <= PC;

//IR

if(ir\_ld) IR <= IMem\_to\_IR;

else IR <= IR;

end

// Asynchronous Outputs of PC, IR and SE

assign PC\_out = PC;

assign IR\_out = IR;

assign SE\_16 = { {16{IR[15]}} , IR[15:0] }; //sign extended Instruction Register

endmodule

Instruction Memory (iM)

`timescale 1ns / 1ps

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* C E C S 4 4 0 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*

\* File Name: InstructionMemory.v

\* Project: CECS 440 Senior Project - Enhanced MIPS Processor

\* Designer: Benjamin Santos and Naoaki Takatsu

\* Email: benjaminsantos@gmx.com

\* naoaki.takatsu@student.csulb.edu

\* Rev. No.: Version 1.0

\* Rev. Date: 10/9/2018

\*

\* Purpose: This is the 4098x8 bit readable/writable instruction memory module.

\*

\* Notes: This module will be used to read and store memory data.

\* To read or write data, chip select "im\_cs" must be set to 1.

\* To read, "im\_rd" must be set to 1. To write, "im\_wr" must be

\* set to 1. Data will be accessed through big endian format.

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

module InstructionMemory(clk, Address, D\_In,

im\_cs, im\_wr, im\_rd, D\_Out);

input clk;

input [31:0] Address, D\_In;

input im\_cs, im\_wr, im\_rd;

output [31:0] D\_Out;

//use 1024x32 'big endian' MSB goes to smaller address

reg [7:0] IMem [0:4095]; //4098x8 memory

always @(posedge clk) begin

//write

if(im\_cs && im\_wr) begin

IMem[Address] <= D\_In[31:24];

IMem[Address + 1] <= D\_In[23:16];

IMem[Address + 2] <= D\_In[15:8];

IMem[Address + 3] <= D\_In[7:0];

end

end

//asynchronous continuous output READ

assign D\_Out = (im\_cs && im\_rd) ?

{IMem[Address],IMem[Address + 1],IMem[Address + 2],IMem[Address+3]}:

32'hZZZZ\_ZZZZ; //high impedance

endmodule

Integer Datapath (IDP)

`timescale 1ns / 1ps

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* C E C S 4 4 0 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*

\* File Name: Integer\_Datapath.v

\* Project: CECS 440 Senior Project - Enhanced MIPS Processor

\* Designer: Benjamin Santos and Naoaki Takatsu

\* Email: benjaminsantos@gmx.com

\* naoaki.takatsu@student.csulb.edu

\* Rev. No.: Version 1.2

\* Rev. Date: 11/8/2018

\*

\* Purpose: This module instantiates two modules:

\* regfile\_32 and alu\_32. This module contains two MUX and two

\* 32 bit register blocks for HI and LO.

\*

\* Notes: This is the integer datapath module. It will connect the 32 bit

\* regfile module with the 32 bit alu module. Within the T-Mux,

\* T\_Sel will be used to select T data from two choices: regfile T

\* and immediate 32 bit value DT. Multiplication and Division

\* outputs will be stored in HI LO register blocks. Within the

\* Y-Mux, Y\_Sel will select ALU\_OUT from five choices: HI, LO,

\* Y\_lo, 32 bit immediate DY, and 32 bit immediate PC\_in. ALU\_OUT

\* will be stored in register specified by address D\_Addr when

\* D\_En is enabled.

\*

\* Revision Notes:

\* 1.2: Added input shift\_val - Shift Value for Barrel Shift

\* 1.3: Added input S\_sel - Select for register RS

\* output D\_in - For the MCU to acquire flags

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

module Integer\_Datapath( clk, reset, S\_Addr, T\_Addr, D\_Addr, D\_En, DT, DA\_sel,

T\_Sel, HILO\_LD, FS, DY, PC\_in, Y\_Sel,

C, V, N, Z, ALU\_OUT, D\_OUT,

shift\_val,

S\_sel

D\_in);

input clk, reset, D\_En, T\_Sel, HILO\_LD;

input [1:0] DA\_sel;

input [2:0] Y\_Sel;

input [4:0] S\_Addr, T\_Addr, D\_Addr, FS;

input [31:0] DT, DY, PC\_in;

input [4:0] shift\_val; //Added for Rev 1.2

input S\_sel; //Added for Rev 1.3

output C, V, N, Z;

output reg [31:0] ALU\_OUT, D\_OUT;

//Wires and Registers

wire [31:0] S, T, Y\_hi, Y\_lo;

wire [4:0] DA\_Addr;

reg [31:0] HI, LO, RS, RT, ALU\_reg, D\_in;

output reg [31:0] D\_in; //D\_in is output to write back the status flags

//Register File module Instantiation

regfile\_32 regfile (

.clk(clk), .reset(reset), .D(ALU\_OUT), .D\_En(D\_En),

.D\_Addr(DA\_Addr),.S\_Addr(S\_Addr), .T\_Addr(T\_Addr),

.S(S), .T(T)

);

//ALU module Instantiation

alu\_32 alu (

.FS(FS), .S(RS), .T(RT),

.N(N), .Z(Z), .V(V), .C(C),   
 .Y\_hi(Y\_hi), .Y\_lo(Y\_lo),  
 .shift\_val(shift\_val)

);

//2-to-1 MUX that determines the D\_OUT output of the CPU

always @(\*)  
 if(T\_Sel) D\_OUT <= DT;

else D\_OUT <= T;

//4-to-1 MUX that determines the address of RegFile to write to

assign DA\_Addr = (DA\_sel == 2'b00)? D\_Addr : //IR[15:11]

(DA\_sel == 2'b01)? T\_Addr : //IR[20:16]

(DA\_sel == 2'b10)? 5'h1F : //Scratch PC register R31

(DA\_sel == 2'b11)? 5'h1D : //Stack Pointer register R29

DA\_Addr;

//Synchronous Register Assignments

always @(posedge clk, posedge reset)

if(reset)

{RS, RT, HI, LO, ALU\_reg, D\_in, ALU\_OUT, D\_OUT} <= 256'h0;

else begin

RT <= D\_OUT;

{ALU\_reg, D\_in} <= {Y\_lo, DY};

if(S\_sel) RS <= Y\_lo;

else RS <= S;

if(HILO\_LD) {HI, LO} <= {Y\_hi, Y\_lo};

else {HI, LO} <= {HI, LO};

end

//5-to-1 MUX that determines the address

always @ (\*)

begin

case( Y\_Sel )

3'b000 : ALU\_OUT = ALU\_reg; // pass SA

3'b001 : ALU\_OUT = HI; // pass SB

3'b010 : ALU\_OUT = LO; // pass SA

3'b011 : ALU\_OUT = D\_in; // pass SB

3'b100 : ALU\_OUT = PC\_in; // pass SB

default : ALU\_OUT = 32'b0; // pass 32 bits of 0s

endcase

end

endmodule

RegFile\_32

`timescale 1ns / 1ps

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* C E C S 4 4 0 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*

\* File Name: regfile\_32.v

\* Project: CECS 440 Senior Project - Enhanced MIPS Processor

\* Designer: Benjamin Santos and Naoaki Takatsu

\* Email: benjaminsantos@gmx.com

\* naoaki.takatsu@student.csulb.edu

\* Rev. No.: Version 1.0

\* Rev. Date: 10/7/2018

\*

\* Purpose: This is the 32 bit register file module.

\*

\* Notes: This 32 bit register file module will store 32 registers of

\* size 32 bit. D is the incoming 32 bit data. S and T are 32 bit

\* data outputs of the module. D\_Addr, S\_Addr, and T\_Addr

\* will address which register to select respectively for D, S,

\* and T. D\_En will act as a write enable. S and T are

\* asynchronous outputs while D will be a synchronous input.

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

module regfile\_32(clk, reset, D, D\_En, D\_Addr, S\_Addr, T\_Addr, S, T);

input clk, reset, D\_En;

input [4:0] S\_Addr, D\_Addr, T\_Addr;

input [31:0] D;

output reg [31:0] S, T;

//32-bit 32 registers

reg [31:0] regs [31:0];

//Asynchronous Reading

always @ (\*)

begin   
 S <= regs[S\_Addr];

T <= regs[T\_Addr];

end

//Synchronous Writing

always @ ( posedge clk, posedge reset )

begin

if ( reset )

regs[0] <= 32'h0;

else if ( D\_En && ( D\_Addr != 5'h0 ) )

regs[D\_Addr] <= D;

else

regs[D\_Addr] <= regs[D\_Addr];

end

endmodule

alu\_32

`timescale 1ns / 1ps

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* C E C S 4 4 0 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*

\* File Name: alu\_32.v

\* Project: CECS 440 Senior Project - Enhanced MIPS Processor

\* Designer: Benjamin Santos and Naoaki Takatsu

\* Email: benjaminsantos@gmx.com

\* naoaki.takatsu@student.csulb.edu

\* Rev. No.: Version 1.3

\* Rev. Date: 11/20/2018

\*

\* Purpose: This module will instantiate three modules:

\* MIPS\_32, DIV\_32, and MPY\_32.

\*

\* Notes: This is the Arithmetic Logic Unit module. It will act as a

\* wrapper instantiating MIPS\_32, DIV\_32, and MPY\_32 modules.

\* 5 bit function select "FS" will select which wires to use for

\* this module's outputs.

\*

\* Revision Notes:

\* 1.2: Added Barrel Shift module

\* Added input shift\_val - Shift Value for Barrel Shift

\* 1.3: Added enhancement Rotate module

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

module alu\_32( FS, S, T, N, Z, V, C, Y\_hi, Y\_lo, shift\_val );

input wire [4:0] FS; // function select

input wire [31:0] S, T; // two 32 bit inputs

input [4:0] shift\_val; //Added for Rev 1.2

output reg N, Z, V, C; // negative, zero, overflow, carry flags

output reg [31:0] Y\_hi, Y\_lo; // two 32 bit outputs

//Temporary Wires

wire [63:0] Y\_mips, Y\_mul, Y\_div; //Y values of MIPS, MPY and DIV

wire N\_mps, Z\_mps, V\_mps, C\_mps; //All flags of the MIPS module

wire N\_mul, Z\_mul, V\_mul, C\_mul; //All flags of the MPY module

wire N\_div, Z\_div, V\_div, C\_div; //All flags of the DIV module

//Added Wires for Rev 1.2

wire [31:0] Y\_BS;

wire C\_BS;

//Added Wires for Rev 1.3

wire [31:0] Y\_ROT;

// instantiate MIPS, DIV, and MPY modules

MIPS\_32 MIPS(

.S(S), //32-bit Input S

.T(T), //32-bit Input T

.FS(FS), //Function Select

.Y\_hi(Y\_mips[63:32]), //32-bit upper output from MIPS

.Y\_lo(Y\_mips[31:0]), //32-bit lower output from MIPS

.N(N\_mps), //Negative flag output of MIPS

.Z(Z\_mps), //Zero flag output of MIPS

.V(V\_mps), //Overflow flag output of MIPS

.C(C\_mps) //Carry flag output of MIPS

);

MPY\_32 MPY(

.S(S), //32-bit Input S

.T(T), //32-bit Input T

.Y\_hi(Y\_mul[63:32]), //32-bit upper output from MPY

.Y\_lo(Y\_mul[31:0]), //32-bit lower output from MPY

.N(N\_mul), //Negative flag output of MPY

.Z(Z\_mul), //Zero flag output of MPY

.V(V\_mul), //Overflow flag output of MPY

.C(C\_mul) //Carry flag output of MPY

);

DIV\_32 DIV(

.S(S), //32-bit Input S

.T(T), //32-bit Input T

.Y\_hi(Y\_div[63:32]), //32-bit upper output from MPY

.Y\_lo(Y\_div[31:0]), //32-bit upper output from MPY

.N(N\_div), //Negative flag output of DIV

.Z(Z\_div), //Zero flag output of DIV

.V(V\_div), //Overflow flag output of DIV

.C(C\_div) //Carry flag output of DIV

);

barrelSHIFT\_32 BS(

.D(T), //32-bit Input T

.shift\_val(shift\_val), //Shift value

.LRRA(FS[1:0]), //Select for shift type

.Y(Y\_BS), //32-bit Output of Barrel Shift

.C\_BS(C\_BS) //Carry of Barrel Shift

);

rotate\_32 ROT(

.D(T), //32-bit Input T

.shift\_val(shift\_val), //Rotate Value

.LorR(FS[0]), //Rotate Direction

.Y(Y\_ROT) //32-bit Output of Rotate

);

//sets the ALU Y value to the Y value from its needed module.

always @(\*) begin

//Barrel Shift

if( (FS == 5'h0C) | (FS == 5'h0D) | (FS == 5'h0E) ) begin

{Y\_hi, Y\_lo} = {32'b0, Y\_BS};

{N, Z, V, C} = {

(Y\_BS[31]) ? 1'b1:1'b0, //Negative

(Y\_BS == 32'h0) ? 1'b1:1'b0, //Zero

1'b0, //Overflow

C\_BS //Carry

};

end

//Rotate

else if( (FS == 5'h1C) | (FS == 5'h1D) ) begin

{Y\_hi, Y\_lo} = {32'b0, Y\_ROT};

{N, Z, V, C} = {

(Y\_ROT[31]) ? 1'b1:1'b0, //Negative

(Y\_ROT == 32'h0) ? 1'b1:1'b0, //Zero

1'b0, //Overflow

1'b0 //Carry

};

end

//Multiply

else if(FS == 5'h1E) begin

{Y\_hi, Y\_lo} = Y\_mul;

{N, Z, V, C} = {N\_mul, Z\_mul, 1'bx, 1'bx};

end

//Divide

else if(FS == 5'h1F) begin

{Y\_hi, Y\_lo} = Y\_div;

{N, Z, V, C} = {N\_div, Z\_div, V\_div, 1'bx};

end

//MIPS

else begin

{Y\_hi, Y\_lo} = Y\_mips;

{N, Z, V, C} = {N\_mps, Z\_mps, V\_mps, C\_mps};

end

end //always

endmodule

MIPS\_32

`timescale 1ns / 1ps

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* C E C S 4 4 0 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*

\* File Name: MIPS\_32.v

\* Project: CECS 440 Senior Project - Enhanced MIPS Processor

\* Designer: Benjamin Santos and Naoaki Takatsu

\* Email: benjaminsantos@gmx.com

\* naoaki.takatsu@student.csulb.edu

\* Rev. No.: Version 1.2

\* Rev. Date: 11/8/2018

\*

\* Purpose: This module will perform the following operations for each of

\* its corresponding 5 bit function select hex value:

\*

\* 0 - Pass\_S 7 - SLTU E - SRA 15 - SP\_INIT

\* 1 - Pass\_T 8 - AND F - INC 16 - ANDI

\* 2 - ADD 9 - OR 10 - DEC 17 - ORI

\* 3 - SUB A - XOR 11 - INC4 18 - LUI

\* 4 - ADDU B - NOR 12 - DEC4 19 - XORI

\* 5 - SUBU C - SLL 13 - ZEROS default - all x's

\* 6 - SLT D - SRL 14 - ONES

\*

\* This module will detect if the result: is a negative value,

\* is a zero, has overflow, or has carry. The results of the

\* operation will be sent out as outputs.

\*

\* Notes: This is the 32 bit MIPS operation module. It will take a 5 bit

\* function selet input "FS" to determine which operation it

\* should execute. I will also take two 32 bit values for its

\* input. Flags 'N', 'Z', 'V', and 'C' (negative, zero, overflow,

\* and carry) will be outputted along with the resulting two

\* 32 bit values "Y\_hi" and "Y\_lo".

\*

\* Revision Notes:

\* 1.2: SLL, SRL and SRA are commented out and are handled by

\* the Barrel Shift module

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

module MIPS\_32( FS, S, T, N, Z, V, C, Y\_hi, Y\_lo );

input [31:0] S, T;

input [4:0] FS;

output reg [31:0] Y\_hi, Y\_lo;

output reg N, Z, V, C;

reg neg, zero, ovf, carry;

//cast S and T as integers for the SLT op

integer inta, intb;

//always @(S, T, FS) //any change of the inputs(S, T or FS)

always @(\*)

begin

case(FS)

5'h00: begin //PASS\_S (Y\_lo = S)

Y\_lo = S;

if (S[31] == 1'b1) neg = 1; else neg = 0;

if (S == 32'b0) zero = 1; else zero = 0;

{N, Z, V, C} = {neg, zero, 2'bxx};

end

5'h01: begin //PASS\_T (Y\_lo = T)

Y\_lo = T;

if (T[31] == 1'b1) neg = 1; else neg = 0;

if (T == 32'b0) zero = 1; else zero = 0;

{N, Z, V, C} = {neg, zero, 2'bxx};

end

5'h02: begin //ADD (Y\_lo = S + T)

{carry, Y\_lo} = S + T;

if (Y\_lo[31] == 1'b1) neg = 1; else neg = 0;

if (Y\_lo == 32'b0) zero = 1; else zero = 0;

if ( (S[31] == 1'b0) & (T[31] == 1'b0) & (Y\_lo[31] == 1'b1) )

ovf = 1'b1;

else if ( (S[31] == 1'b1) & (T[31] == 1'b1) &

(Y\_lo[31] == 1'b0) )

ovf = 1'b1;

else

ovf = 1'b0;

{N, Z, V, C} = {neg, zero, ovf, carry};

end

5'h03: begin //SUB (Y\_lo = S - T)

{carry, Y\_lo} = S - T;

if (Y\_lo[31] == 1'b1) neg = 1; else neg = 0;

if (Y\_lo == 32'b0) zero = 1; else zero = 0;

if ( (S[31] == 1'b0) & (T[31] == 1'b0) & (Y\_lo[31] == 1'b1) )

ovf = 1'b1;

else if ( (S[31] == 1'b1) & (T[31] == 1'b1) &

(Y\_lo[31] == 1'b0) )

ovf = 1'b1;

else

ovf = 1'b0;

{N, Z, V, C} = {neg, zero, ovf, carry};

end

5'h04: begin //ADDU (Y\_lo = S + T) unsigned

{carry, Y\_lo} = S + T;

if (Y\_lo == 32'b0) zero = 1; else zero = 0;

ovf = carry;

{N, Z, V, C} = {1'b0, zero, ovf, carry};

end

5'h05: begin //SUBU (Y\_lo = S - T) unsigned

Y\_lo = S - T;

if (S < T) neg = 1; else neg = 0;

if (Y\_lo == 32'b0) zero = 1; else zero = 0;

if (T > S) carry = 1; else carry = 0;

ovf = carry;

{N, Z, V, C} = {1'b0, zero, ovf, carry};

end

5'h06: begin //SLT (if S < T -> Y\_lo = 1, else Y\_lo = 0)

//S and T are casted as integers to compare

//signed integers

inta = S;

intb = T;

if(inta < intb) Y\_lo = 32'd1;

else Y\_lo = 32'b0;

if (Y\_lo[31] == 1'b1) neg = 1; else neg = 0;

if (Y\_lo == 32'b0) zero = 1; else zero = 0;

{N, Z, V, C} = {neg, zero, 2'bxx};

end

5'h07: begin //SLTU (if S < T -> Y\_lo = 1, else Y\_lo = 0) unsigned

if(S < T) Y\_lo = 32'd1;

else Y\_lo = 32'b0;

if (Y\_lo == 32'b0) zero = 1; else zero = 0;

{N, Z, V, C} = {1'b0, zero, 2'bxx};

end

5'h08: begin //AND (Y\_lo = S & T)

Y\_lo = S & T;

if (Y\_lo[31] == 1'b1) neg = 1; else neg = 0;

if (Y\_lo == 32'b0) zero = 1; else zero = 0;

{N, Z, V, C} = {neg, zero, 2'bxx};

end

5'h09: begin //OR (Y\_lo = S | T)

Y\_lo = S | T;

if (Y\_lo[31] == 1'b1) neg = 1; else neg = 0;

if (Y\_lo == 32'b0) zero = 1; else zero = 0;

{N, Z, V, C} = {neg, zero, 2'bxx};

end

5'h0A: begin //XOR (Y\_lo = S ^ T)

Y\_lo = S ^ T;

if (Y\_lo[31] == 1'b1) neg = 1; else neg = 0;

if (Y\_lo == 32'b0) zero = 1; else zero = 0;

{N, Z, V, C} = {neg, zero, 2'bxx};

end

5'h0B: begin //NOR (Y\_lo = ~(S | T))

Y\_lo = ~(S | T);

if (Y\_lo[31] == 1'b1) neg = 1; else neg = 0;

if (Y\_lo == 32'b0) zero = 1; else zero = 0;

{N, Z, V, C} = {neg, zero, 2'bxx};

end

/\*

\* Shift Instructions are handled by Barrel Shift module \*

5'h0C: begin //SLL (Y\_lo = T < 1) with 0 fill

Y\_lo = {T[30:0], 1'b0};

if (Y\_lo[31] == 1'b1) neg = 1; else neg = 0;

if (Y\_lo == 32'b0) zero = 1; else zero = 0;

{N, Z, V, C} = {neg, zero, 1'bx, T[31]};

end

5'h0D: begin //SRL (Y\_lo = T > 1) with 0 fill

Y\_lo = {1'b0, T[31:1]};

if (Y\_lo[31] == 1'b1) neg = 1; else neg = 0;

if (Y\_lo == 32'b0) zero = 1; else zero = 0;

{N, Z, V, C} = {neg, zero, 1'bx, T[0]};

end

5'h0E: begin //SRA(keeps MSB and shifts to the right)

Y\_lo = {T[31], T[31:1]};

if (Y\_lo[31] == 1'b1) neg = 1; else neg = 0;

if (Y\_lo == 32'b0) zero = 1; else zero = 0;

{N, Z, V, C} = {neg, zero, 1'bx, T[0]};

end

\*/

5'h0F: begin //INC (Y\_lo = S + 1)

{carry, Y\_lo} = S + 1'b1;

if (Y\_lo[31] == 1'b1) neg = 1; else neg = 0;

if (Y\_lo == 32'b0) zero = 1; else zero = 0;

if ( (S[31] == 1'b0) & (T[31] == 1'b0) & (Y\_lo[31] == 1'b1) )

ovf = 1'b1;

else if ( (S[31] == 1'b1) & (T[31] == 1'b1) &

(Y\_lo[31] == 1'b0) )

ovf = 1'b1;

else

ovf = 1'b0;

{N, Z, V, C} = {neg, zero, ovf, carry};

end

5'h10: begin //DEC (Y\_lo = S - 1)

Y\_lo = S - 1'b1;

if (Y\_lo[31] == 1'b1) neg = 1; else neg = 0;

if (Y\_lo == 32'b0) zero = 1; else zero = 0;

if (S == 32'b0) carry = 1; else carry = 0;

if ( (S[31] == 1'b0) & (T[31] == 1'b0) & (Y\_lo[31] == 1'b1) )

ovf = 1'b1;

else if ( (S[31] == 1'b1) & (T[31] == 1'b1) &

(Y\_lo[31] == 1'b0) )

ovf = 1'b1;

else

ovf = 1'b0;

{N, Z, V, C} = {neg, zero, ovf, carry};

end

5'h11: begin //INC4 (Y\_lo = S + 4)

{carry, Y\_lo} = S + 3'd4;

if (Y\_lo[31] == 1'b1) neg = 1; else neg = 0;

if (Y\_lo == 32'b0) zero = 1; else zero = 0;

if ( (S[31] == 1'b0) & (T[31] == 1'b0) & (Y\_lo[31] == 1'b1) )

ovf = 1'b1;

else if ( (S[31] == 1'b1) & (T[31] == 1'b1) &

(Y\_lo[31] == 1'b0) )

ovf = 1'b1;

else

ovf = 1'b0;

{N, Z, V, C} = {neg, zero, ovf, carry};

end

5'h12: begin //DEC4 (Y\_lo = S - 4)

Y\_lo = S - 3'd4;

if (Y\_lo[31] == 1'b1) neg = 1; else neg = 0;

if (Y\_lo == 32'b0) zero = 1; else zero = 0;

if (S > 32'hFFFF\_FFFB) carry = 1; else carry = 0;

if ( (S[31] == 1'b0) & (T[31] == 1'b0) &

(Y\_lo[31] == 1'b1) )

ovf = 1'b1;

else if ( (S[31] == 1'b1) & (T[31] == 1'b1) &

(Y\_lo[31] == 1'b0) )

ovf = 1'b1;

else

ovf = 1'b0;

{N, Z, V, C} = {neg, zero, ovf, carry};

end

5'h13: begin //ZEROS (Y\_lo = 0)

Y\_lo = 32'h0;

{N, Z, V, C} = 4'b01xx;

end

5'h14: begin //ONES (Y\_lo = FFFF\_FFFF)

Y\_lo = 32'hFFFF\_FFFF;

{N, Z, V, C} = 4'b10xx;

end

5'h15: begin //SP\_INIT (Y\_lo = 32'h3FC)

Y\_lo = 32'h3FC;

if (Y\_lo[31] == 1'b1) neg = 1; else neg = 0;

if (Y\_lo == 32'b0) zero = 1; else zero = 0;

{N, Z, V, C} = {neg, zero, 2'bxx};

end

5'h16: begin //ANDI (Y\_lo = ( S & {16'h0, T[15:0]} ))

Y\_lo = ( S & {16'h0, T[15:0]} );

if (Y\_lo[31] == 1'b1) neg = 1; else neg = 0;

if (Y\_lo == 32'b0) zero = 1; else zero = 0;

{N, Z, V, C} = {neg, zero, 2'bxx};

end

5'h17: begin //ORI (Y\_lo = ( S | {16'h0, T[15:0]} ))

Y\_lo = ( S | {16'h0, T[15:0]} );

if (Y\_lo[31] == 1'b1) neg = 1; else neg = 0;

if (Y\_lo == 32'b0) zero = 1; else zero = 0;

{N, Z, V, C} = {neg, zero, 2'bxx};

end

5'h18: begin //LUI (Y\_lo = {T[15:0], 16'h0})

Y\_lo = {T[15:0], 16'h0};

if (Y\_lo[31] == 1'b1) neg = 1; else neg = 0;

if (Y\_lo == 32'b0) zero = 1; else zero = 0;

{N, Z, V, C} = {neg, zero, 2'bxx};

end

5'h19: begin //XORI (Y\_lo = ( S ^ {16'h0, T[15:0]} ))

Y\_lo = ( S ^ {16'h0, T[15:0]} );

if (Y\_lo[31] == 1'b1) neg = 1; else neg = 0;

if (Y\_lo == 32'b0) zero = 1; else zero = 0;

{N, Z, V, C} = {neg, zero, 2'bxx};

end

default: begin //default set to the PASS\_S function

Y\_lo = S;

if (Y\_lo[31] == 1'b1) neg = 1; else neg = 0;

if (Y\_lo == 32'b0) zero = 1; else zero = 0;

{N, Z, V, C} = {neg, zero, 2'bxx};

end

endcase

//since Y\_hi is never used in MIPS, set it to 0

Y\_hi = 32'b0;

end

endmodule

MPY\_32

`timescale 1ns / 1ps

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* C E C S 4 4 0 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*

\* File Name: MPY\_32.v

\* Project: CECS 440 Senior Project - Enhanced MIPS Processor

\* Designer: Benjamin Santos and Naoaki Takatsu

\* Email: benjaminsantos@gmx.com

\* naoaki.takatsu@student.csulb.edu

\* Rev. No.: Version 1.0

\* Rev. Date: 10/7/2018

\*

\* Purpose: This module will take two 32 bit inputs and perform a mutlipy.

\* The result will be sent out as a 64 bit output.

\*

\* Notes: This is the 32 bit multiplication module. It will multiply

\* 32 bit A and B inputs, then send the resulting 64 bit Y as its

\* output. Prior to its multiplication step, this module will

\* change all negative inputs to positive values. After

\* multiplying, it will check the two inputs' sign bit and change

\* the result to a negative value if both sign bits are not equal

\* to each other.

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

module MPY\_32(S, T, Y\_hi, Y\_lo, N, Z, V, C);

input [31:0] S, T;

output reg [31:0] Y\_hi, Y\_lo;

output reg N, Z, V, C;

//cast S and T into the following integers

integer inta, intb;

reg neg, zero;

reg [63:0] mul;

always @(\*) begin

inta = S;

intb = T;

mul = inta \* intb;

{Y\_hi, Y\_lo} = mul; //Product

if (mul[31] == 1'b1) neg = 1; //Negative flag

else neg = 0;

if ( (S == 32'b0) & (T == 32'b0) ) //Zero flag

zero = 1;

else

zero = 0;

{N, Z, V, C} = {neg, zero, 2'bxx}; //Set the output flags

end

endmodule

DIV\_32

`timescale 1ns / 1ps

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* C E C S 4 4 0 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*

\* File Name: DIV\_32.v

\* Project: CECS 440 Senior Project - Enhanced MIPS Processor

\* Designer: Benjamin Santos and Naoaki Takatsu

\* Email: benjaminsantos@gmx.com

\* naoaki.takatsu@student.csulb.edu

\* Rev. No.: Version 1.0

\* Rev. Date: 10/7/2018

\*

\* Purpose: This module will take two 32 bit inputs and perform a divide.

\* The result will be sent out as a 64 bit output.

\*

\* Notes: This is the 32 bit division module. It will divide

\* 32 bit A and B inputs, then send the resulting 64 bit Y as its

\* output. Y[63:32] will hold the remainder value while Y[31:0]

\* will hold the quotient value. Prior to its division step, this

\* module will change all negative inputs to positive values.

\* After dividing, it will check the two inputs' sign bit and

\* change the results to a negative quotient and remainder value

\* depending on the inputs' (divisor and dividend's) sign bit.

\* Overflow bit "Vt" will be set to 1 if the operation is a

\* division by zero.

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

module DIV\_32(S, T, Y\_hi, Y\_lo, N, Z, V, C);

input [31:0] S, T;

output reg [31:0] Y\_hi, Y\_lo;

output reg N, Z, V, C;

reg neg, zero;

reg [31:0] quot, rem;

//cast S and T into the following integers

integer inta;

integer intb;

always @(\*) begin

inta = S;

intb = T;

quot = inta / intb;

rem = inta % intb;

{Y\_hi, Y\_lo} = {rem, quot}; //Quotient

if (quot[31] == 1'b1) neg = 1; else neg = 0; //Negative flag

if (Y\_lo == 32'b0) zero = 1; else zero = 0; //Zero flag

{N, Z, V, C} = {neg, zero, 2'bxx}; //Set the output flags

end

endmodule

Barrel Shift

`timescale 1ns / 1ps

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* C E C S 4 4 0 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*

\* File Name: barrelSHIFT\_32.v

\* Project: CECS 440 Senior Project - Enhanced MIPS Processor

\* Designer: Benjamin Santos and Naoaki Takatsu

\* Email: benjaminsantos@gmx.com

\* naoaki.takatsu@student.csulb.edu

\* Rev. No.: Version 1.0

\* Rev. Date: 11/20/2018

\*

\* Purpose: Provides the ALU the ability to shift up to 32 times.

\* The Barrel Shift can Shift Left Logical(SLL), Shift Right

\* Logical(SRL) and Shift Right Arithmetic(SRA).

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

module barrelSHIFT\_32(D, shift\_val, LRRA, Y, C\_BS);

input [31:0] D; //32-bit input to be shifted

input [4:0] shift\_val; //Amount of Shifts to be done

input [1:0] LRRA; //Type of Shift

output reg [31:0] Y; //32-bit output of the Barrel Shift

output reg C\_BS; //Carry Flag from the Shifts

always @(\*)

if(LRRA == 2'b00) begin //Left Shift Logical

case(shift\_val)

5'd1: Y = { D[30:0], 1'b0 }; 5'd17: Y = { D[14:0], 17'b0 };

5'd2: Y = { D[29:0], 2'b0 }; 5'd18: Y = { D[13:0], 18'b0 };

5'd3: Y = { D[28:0], 3'b0 }; 5'd19: Y = { D[12:0], 19'b0 };

5'd4: Y = { D[27:0], 4'b0 }; 5'd20: Y = { D[11:0], 20'b0 };

5'd5: Y = { D[26:0], 5'b0 }; 5'd21: Y = { D[10:0], 21'b0 };

5'd6: Y = { D[25:0], 6'b0 }; 5'd22: Y = { D[9:0], 22'b0 };

5'd7: Y = { D[24:0], 7'b0 }; 5'd23: Y = { D[8:0], 23'b0 };

5'd8: Y = { D[23:0], 8'b0 }; 5'd24: Y = { D[7:0], 24'b0 };

5'd9: Y = { D[22:0], 9'b0 }; 5'd25: Y = { D[6:0], 25'b0 };

5'd10: Y = { D[21:0], 10'b0 }; 5'd26: Y = { D[5:0], 26'b0 };

5'd11: Y = { D[20:0], 11'b0 }; 5'd27: Y = { D[4:0], 27'b0 };

5'd12: Y = { D[19:0], 12'b0 }; 5'd28: Y = { D[3:0], 28'b0 };

5'd13: Y = { D[18:0], 13'b0 }; 5'd29: Y = { D[2:0], 29'b0 };

5'd14: Y = { D[17:0], 14'b0 }; 5'd30: Y = { D[1:0], 30'b0 };

5'd15: Y = { D[16:0], 15'b0 }; 5'd31: Y = { D[0], 31'b0 };

5'd16: Y = { D[15:0], 16'b0 };

default: Y = Y;

endcase

C\_BS = D[31];

end

else if(LRRA == 2'b01) begin //Right Shift Logical

case(shift\_val)

5'd1: Y = { 1'b0, D[31:1] }; 5'd17: Y = { 17'b0, D[31:17] };

5'd2: Y = { 2'b0, D[31:2] }; 5'd18: Y = { 18'b0, D[31:18] };

5'd3: Y = { 3'b0, D[31:3] }; 5'd19: Y = { 19'b0, D[31:19] };

5'd4: Y = { 4'b0, D[31:4] }; 5'd20: Y = { 20'b0, D[31:20] };

5'd5: Y = { 5'b0, D[31:5] }; 5'd21: Y = { 21'b0, D[31:21] };

5'd6: Y = { 6'b0, D[31:6] }; 5'd22: Y = { 22'b0, D[31:22] };

5'd7: Y = { 7'b0, D[31:7] }; 5'd23: Y = { 23'b0, D[31:23] };

5'd8: Y = { 8'b0, D[31:8] }; 5'd24: Y = { 24'b0, D[31:24] };

5'd9: Y = { 9'b0, D[31:9] }; 5'd25: Y = { 25'b0, D[31:25] };

5'd10: Y = { 10'b0, D[31:10] }; 5'd26: Y = { 26'b0, D[31:26] };

5'd11: Y = { 11'b0, D[31:11] }; 5'd27: Y = { 27'b0, D[31:27] };

5'd12: Y = { 12'b0, D[31:12] }; 5'd28: Y = { 28'b0, D[31:28] };

5'd13: Y = { 13'b0, D[31:13] }; 5'd29: Y = { 29'b0, D[31:29] };

5'd14: Y = { 14'b0, D[31:14] }; 5'd30: Y = { 30'b0, D[31:30] };

5'd15: Y = { 15'b0, D[31:15] }; 5'd31: Y = { 31'b0, D[31] };

5'd16: Y = { 16'b0, D[31:16] };

default: Y = Y;

endcase

C\_BS = D[0];

end

else if (LRRA == 2'b10) begin //Right Shift Arithmetic

case(shift\_val)

5'd1: Y = { D[31], D[31:1] };

5'd2: Y = { { 2{D[31]}}, D[31:2] };

5'd3: Y = { { 3{D[31]}}, D[31:3] };

5'd4: Y = { { 4{D[31]}}, D[31:4] };

5'd5: Y = { { 5{D[31]}}, D[31:5] };

5'd6: Y = { { 6{D[31]}}, D[31:6] };

5'd7: Y = { { 7{D[31]}}, D[31:7] };

5'd8: Y = { { 8{D[31]}}, D[31:8] };

5'd9: Y = { { 9{D[31]}}, D[31:9] };

5'd10: Y = { {10{D[31]}}, D[31:10] };

5'd11: Y = { {11{D[31]}}, D[31:11] };

5'd12: Y = { {12{D[31]}}, D[31:12] };

5'd13: Y = { {13{D[31]}}, D[31:13] };

5'd14: Y = { {14{D[31]}}, D[31:14] };

5'd15: Y = { {15{D[31]}}, D[31:15] };

5'd16: Y = { {16{D[31]}}, D[31:16] };

5'd17: Y = { {17{D[31]}}, D[31:17] };

5'd18: Y = { {18{D[31]}}, D[31:18] };

5'd19: Y = { {19{D[31]}}, D[31:19] };

5'd20: Y = { {20{D[31]}}, D[31:20] };

5'd21: Y = { {21{D[31]}}, D[31:21] };

5'd22: Y = { {22{D[31]}}, D[31:22] };

5'd23: Y = { {23{D[31]}}, D[31:23] };

5'd24: Y = { {24{D[31]}}, D[31:24] };

5'd25: Y = { {25{D[31]}}, D[31:25] };

5'd26: Y = { {26{D[31]}}, D[31:26] };

5'd27: Y = { {27{D[31]}}, D[31:27] };

5'd28: Y = { {28{D[31]}}, D[31:28] };

5'd29: Y = { {29{D[31]}}, D[31:29] };

5'd30: Y = { {30{D[31]}}, D[31:30] };

5'd31: Y = { {31{D[31]}}, D[31] };

default: Y = Y;

endcase

C\_BS = D[0];

end

endmodule

Rotate\_32

`timescale 1ns / 1ps

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* C E C S 4 4 0 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*

\* File Name: rotate\_32.v

\* Project: CECS 440 Senior Project - Enhanced MIPS Processor

\* Designer: Benjamin Santos and Naoaki Takatsu

\* Email: benjaminsantos@gmx.com

\* naoaki.takatsu@student.csulb.edu

\* Rev. No.: Version 1.0

\* Rev. Date: 11/20/2018

\*

\* Purpose: Provides the ALU the ability to rotate up to 31 times.

\* The Rotate module can rotate left and right.

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

module rotate\_32(D, shift\_val, LorR, Y);

input [31:0] D; //32-bit input to be rotated

input [4:0] shift\_val; //Amount of Shifts to be done

input LorR; //Rotate rule

output reg [31:0] Y; //32-bit output of the Barrel Shift

always @(\*) begin

if(LorR == 1'b0) //Rotate Left

case(shift\_val)

5'd1: Y = { D[30:0], D[31] }; 5'd17: Y = { D[14:0], D[31:15] };

5'd2: Y = { D[29:0], D[31:30] }; 5'd18: Y = { D[13:0], D[31:14] };

5'd3: Y = { D[28:0], D[31:29] }; 5'd19: Y = { D[12:0], D[31:13] };

5'd4: Y = { D[27:0], D[31:28] }; 5'd20: Y = { D[11:0], D[31:12] };

5'd5: Y = { D[26:0], D[31:27] }; 5'd21: Y = { D[10:0], D[31:11] };

5'd6: Y = { D[25:0], D[31:26] }; 5'd22: Y = { D[9:0], D[31:10] };

5'd7: Y = { D[24:0], D[31:25] }; 5'd23: Y = { D[8:0], D[31:9] };

5'd8: Y = { D[23:0], D[31:24] }; 5'd24: Y = { D[7:0], D[31:8] };

5'd9: Y = { D[22:0], D[31:23] }; 5'd25: Y = { D[6:0], D[31:7] };

5'd10: Y = { D[21:0], D[31:22] }; 5'd26: Y = { D[5:0], D[31:6] };

5'd11: Y = { D[20:0], D[31:21] }; 5'd27: Y = { D[4:0], D[31:5] };

5'd12: Y = { D[19:0], D[31:20] }; 5'd28: Y = { D[3:0], D[31:4] };

5'd13: Y = { D[18:0], D[31:19] }; 5'd29: Y = { D[2:0], D[31:3] };

5'd14: Y = { D[17:0], D[31:18] }; 5'd30: Y = { D[1:0], D[31:2] };

5'd15: Y = { D[16:0], D[31:17] }; 5'd31: Y = { D[0], D[31:1] };

5'd16: Y = { D[15:0], D[31:16] };

default: Y = Y;

endcase

else //Rotate Right

case(shift\_val)

5'd1: Y = { D[0], D[31:1] }; 5'd17: Y = { D[16:0], D[31:17] };

5'd2: Y = { D[1:0], D[31:2] }; 5'd18: Y = { D[17:0], D[31:18] };

5'd3: Y = { D[2:0], D[31:3] }; 5'd19: Y = { D[18:0], D[31:19] };

5'd4: Y = { D[3:0], D[31:4] }; 5'd20: Y = { D[19:0], D[31:20] };

5'd5: Y = { D[4:0], D[31:5] }; 5'd21: Y = { D[20:0], D[31:21] };

5'd6: Y = { D[5:0], D[31:6] }; 5'd22: Y = { D[21:0], D[31:22] };

5'd7: Y = { D[6:0], D[31:7] }; 5'd23: Y = { D[22:0], D[31:23] };

5'd8: Y = { D[7:0], D[31:8] }; 5'd24: Y = { D[23:0], D[31:24] };

5'd9: Y = { D[8:0], D[31:9] }; 5'd25: Y = { D[24:0], D[31:25] };

5'd10: Y = { D[9:0], D[31:10] }; 5'd26: Y = { D[25:0], D[31:26] };

5'd11: Y = { D[10:0], D[31:11] }; 5'd27: Y = { D[26:0], D[31:27] };

5'd12: Y = { D[11:0], D[31:12] }; 5'd28: Y = { D[27:0], D[31:28] };

5'd13: Y = { D[12:0], D[31:13] }; 5'd29: Y = { D[28:0], D[31:29] };

5'd14: Y = { D[13:0], D[31:14] }; 5'd30: Y = { D[29:0], D[31:30] };

5'd15: Y = { D[14:0], D[31:15] }; 5'd31: Y = { D[30:0], D[31] };

5'd16: Y = { D[15:0], D[31:16] };

default: Y = Y;

endcase

end

endmodule

DataMemory

`timescale 1ns / 1ps

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* C E C S 4 4 0 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*

\* File Name: DataMemory.v

\* Project: CECS 440 Senior Project - Enhanced MIPS Processor

\* Designer: Benjamin Santos and Naoaki Takatsu

\* Email: benjaminsantos@gmx.com

\* naoaki.takatsu@student.csulb.edu

\* Rev. No.: Version 1.0

\* Rev. Date: 10/7/2018

\*

\* Purpose: This is the 4098x8 bit readable/writable data memory module.

\*

\* Notes: This module will be used to read and store memory data.

\* To read or write data, chip select "dm\_cs" must be set to 1.

\* To read, "dm\_rd" must be set to 1. To write, "dm\_wr" must be

\* set to 1. Data will be accessed through big endian format.

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

module DataMemory(clk, Address, D\_In, dm\_cs, dm\_wr, dm\_rd, D\_Out);

input clk, dm\_cs, dm\_wr, dm\_rd;

input [31:0] Address, D\_In;

output [31:0] D\_Out;

//use 1024x32 'big endian' MSB goes to smaller address

reg [7:0] M [0:4095]; //4098x8 memory

always @(posedge clk) begin

//write

if(dm\_cs && dm\_wr) begin

M[Address] <= D\_In[31:24];

M[Address + 1] <= D\_In[23:16];

M[Address + 2] <= D\_In[15:8];

M[Address + 3] <= D\_In[7:0];

end

end

//asynchronous continuous output READ

assign D\_Out = (dm\_cs && dm\_rd) ?

{M[Address], M[Address + 1], M[Address + 2], M[Address + 3]} :

32'hZZZZ\_ZZZZ; //high impedance

endmodule

InputOutput

`timescale 1ns / 1ps

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* C E C S 4 4 0 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*

\* File Name: InputOutput.v

\* Project: CECS 440 Senior Project - Enhanced MIPS Processor

\* Designer: Benjamin Santos and Naoaki Takatsu

\* Email: benjaminsantos@gmx.com

\* naoaki.takatsu@student.csulb.edu

\* Rev. No.: Version 1.0

\* Rev. Date: 10/7/2018

\*

\* Purpose: This is the 4098x8 bit readable/writable I/O memory module.

\*

\* Notes: This module will be used to read and store I/O data.

\* To read or write data, chip select "io\_cs" must be set to 1.

\* To read, "io\_rd" must be set to 1. To write, "io\_wr" must be

\* set to 1. Data will be accessed through big endian format.

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

module InputOutput(clk, Address, D\_In, io\_cs, io\_wr, io\_rd, D\_Out, int\_ack, intr);

input clk, io\_cs, io\_wr, io\_rd;

input int\_ack;

input [31:0] Address, D\_In;

output [31:0] D\_Out;

output reg intr;

//use 1024x32 'big endian' MSB goes to smaller address

reg [7:0] IO [0:4095]; //4098x8 memory

always @(posedge clk) begin

//write

if(io\_cs && io\_wr) begin

IO[Address] <= D\_In[31:24];

IO[Address + 1] <= D\_In[23:16];

IO[Address + 2] <= D\_In[15:8];

IO[Address + 3] <= D\_In[7:0];

end

end

//asynchronous continuous output READ

assign D\_Out = (io\_cs && io\_rd) ?

{IO[Address], IO[Address + 1], IO[Address + 2], IO[Address + 3]} :

32'hZZZZ\_ZZZZ; //high impedance

//Deassert Interrupt when Interrupt is acknowledged by the CPU

always @(posedge int\_ack)

intr <= 1'b0;

endmodule

**Instruction Memory Module 1**

**Testing Instructions:** LUI, ORI, ADD, BEQ, BNE, SW, BREAK

@0

3c 01 12 34 // main: lui $01, 0x1234

34 21 56 78 // ori $01, 0x5678 # LI R01, 0x12345678

3c 02 87 65 // lui $02, 0x8765

34 42 43 21 // ori $02, 0x4321 # LI R02, 0x87654321

00 01 18 20 // add $03, $00, $01 # COPY R03, R01

10 22 00 01 // beq $01, $02, no\_eq # should not branch

10 23 00 03 // beq $01, $03, yes\_eq # should branch

3c 0e ff ff // no\_eq: lui $14, 0xFFFF

35 ce ff ff // ori $14, 0xFFFF # LI R14,0xFFFFFFFF"fail flag"

00 00 00 0d // break

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

14 23 00 01 // bne $01, $03, no\_ne # should not branch

14 22 00 03 // bne $01, $02, yes\_ne # should branch

3c 0f ff ff // no\_ne: lui $15, 0xFFFF

35 ef ff ff // ori $15, 0xFFFF # LI R15 0xFFFFFFFF"fail flag"

00 00 00 0d // break

00 00 78 20 // yes\_ne: add $15, $0, $0 # CLR R15 "pass flag"

3c 0d 10 01 // lui $13, 0x1001

35 ad 00 c0 // ori $13, 0x00C0 # LI R13, 0x100100C0

ad a1 00 00 // sw $01, 0($13) # ST [R13], R01

00 00 00 0d // break

- Registers $r1 and $r2 gets their respective values from the LUI and ORI instructions where   
 LUI fills the upper 16 bits and ORI fills the lower 16 bits of the data.

- Register $r3 gets its value from the ADD instruction adding $r1 and $zero

- Since register $r14 has 0x00000000, this proves that the BEQ instruction is working as   
 intended

- Since register $r15 has 0x00000000, this proves that the BNE instruction is working   
 correctly

- Since data in Memory Location 0x0C0 has 0x12345678, this proves that the SW instruction   
 is working correctly

- Since the final value of IR has 0x0000000d, this proves the functionality of the BREAK  
 instruction

BREAK INSTRUCTION FETCHED 700.0 ps

R E G I S T E R ' S A F T E R B R E A K

t= 710.0 ps, $r0 = 0x00000000 || $r16 = 0xxxxxxxxx

t= 720.0 ps, $r1 = 0x12345678 || $r17 = 0xxxxxxxxx

t= 730.0 ps, $r2 = 0x87654321 || $r18 = 0xxxxxxxxx

t= 740.0 ps, $r3 = 0x12345678 || $r19 = 0xxxxxxxxx

t= 750.0 ps, $r4 = 0xxxxxxxxx || $r20 = 0xxxxxxxxx

t= 760.0 ps, $r5 = 0xxxxxxxxx || $r21 = 0xxxxxxxxx

t= 770.0 ps, $r6 = 0xxxxxxxxx || $r22 = 0xxxxxxxxx

t= 780.0 ps, $r7 = 0xxxxxxxxx || $r23 = 0xxxxxxxxx

t= 790.0 ps, $r8 = 0xxxxxxxxx || $r24 = 0xxxxxxxxx

t= 800.0 ps, $r9 = 0xxxxxxxxx || $r25 = 0xxxxxxxxx

t= 810.0 ps, $r10 = 0xxxxxxxxx || $r26 = 0xxxxxxxxx

t= 820.0 ps, $r11 = 0xxxxxxxxx || $r27 = 0xxxxxxxxx

t= 830.0 ps, $r12 = 0xxxxxxxxx || $r28 = 0xxxxxxxxx

t= 840.0 ps, $r13 = 0x100100c0 || $r29 = 0x000003fc

t= 850.0 ps, $r14 = 0x00000000 || $r30 = 0xxxxxxxxx

t= 860.0 ps, $r15 = 0x00000000 || $r31 = 0x00000000

Memory Locations 0xC0 to 0xFF

t= 870.0 ps, M[000000c0] = 0x12345678

t= 880.0 ps, M[000000c4] = 0xxxxxxxxx

t= 890.0 ps, M[000000c8] = 0xxxxxxxxx

t= 900.0 ps, M[000000cc] = 0xxxxxxxxx

t= 910.0 ps, M[000000d0] = 0xxxxxxxxx

t= 920.0 ps, M[000000d4] = 0xxxxxxxxx

t= 930.0 ps, M[000000d8] = 0xxxxxxxxx

t= 940.0 ps, M[000000dc] = 0xxxxxxxxx

t= 950.0 ps, M[000000e0] = 0xxxxxxxxx

t= 960.0 ps, M[000000e4] = 0xxxxxxxxx

t= 970.0 ps, M[000000e8] = 0xxxxxxxxx

t= 980.0 ps, M[000000ec] = 0xxxxxxxxx

t= 990.0 ps, M[000000f0] = 0xxxxxxxxx

t=1000.0 ps, M[000000f4] = 0xxxxxxxxx

t=1010.0 ps, M[000000f8] = 0xxxxxxxxx

t=1020.0 ps, M[000000fc] = 0xxxxxxxxx

t=1030.0 ps, PC = 0x00000054, IR = 0x0000000d

**Instruction Memory Module 2**

**Testing Instructions:**  ADDI, SRL, J

@0

3c 01 ff ff // main: lui $01, 0xFFFF

34 21 ff ff // ori $01, 0xFFFF # LI R01, 0xFFFFFFF

20 02 00 10 // addi $02, $00, 0x10 # LI R02, 0x10

3c 0f 10 01 // lui $15, 0x1001

35 ef 00 c0 // ori $15, 0x00C0 # LI R15, 0x100100C0

00 01 08 42 // top: srl $01, $01, 1 # logical shift right 1 bit

ad e1 00 00 // sw $01, 0($15) # ST [R15], R01

21 ef 00 04 // addi $15, $15, 4 # inc memory pointer by 4

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 done

08 10 00 0c // j exit # jump around a halt instr

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

$r2 is decremented by 1, sixteen times until it reaches 0 and JUMPS to the exit.

$r15 from 0x100100C0 incremented by 4, sixteen times and resulted into 0x10010100  
- Register $r2 having the value 0x00000000 proves that the ADDI function works correctly.

- Since register $r14 has 0x5A5A3C3C, this proves that the J instruction is working as   
 intended

- Values inside the Memory Locations 0xC0 to 0xFF proves that both SRL and SW functions   
 work correctly.

BREAK INSTRUCTION FETCHED 3650.0 ps

R E G I S T E R ' S A F T E R B R E A K

t=3660.0 ps, $r0 = 0x00000000 || $r16 = 0xxxxxxxxx

t=3670.0 ps, $r1 = 0x0000ffff || $r17 = 0xxxxxxxxx

t=3680.0 ps, $r2 = 0x00000000 || $r18 = 0xxxxxxxxx

t=3690.0 ps, $r3 = 0xxxxxxxxx || $r19 = 0xxxxxxxxx

t=3700.0 ps, $r4 = 0xxxxxxxxx || $r20 = 0xxxxxxxxx

t=3710.0 ps, $r5 = 0xxxxxxxxx || $r21 = 0xxxxxxxxx

t=3720.0 ps, $r6 = 0xxxxxxxxx || $r22 = 0xxxxxxxxx

t=3730.0 ps, $r7 = 0xxxxxxxxx || $r23 = 0xxxxxxxxx

t=3740.0 ps, $r8 = 0xxxxxxxxx || $r24 = 0xxxxxxxxx

t=3750.0 ps, $r9 = 0xxxxxxxxx || $r25 = 0xxxxxxxxx

t=3760.0 ps, $r10 = 0xxxxxxxxx || $r26 = 0xxxxxxxxx

t=3770.0 ps, $r11 = 0xxxxxxxxx || $r27 = 0xxxxxxxxx

t=3780.0 ps, $r12 = 0xxxxxxxxx || $r28 = 0xxxxxxxxx

t=3790.0 ps, $r13 = 0xxxxxxxxx || $r29 = 0x000003fc

t=3800.0 ps, $r14 = 0x5a5a3c3c || $r30 = 0xxxxxxxxx

t=3810.0 ps, $r15 = 0x10010100 || $r31 = 0x00000000

Memory Locations 0xC0 to 0xFF

t=3820.0 ps, M[000000c0] = 0x7fffffff

t=3830.0 ps, M[000000c4] = 0x3fffffff

t=3840.0 ps, M[000000c8] = 0x1fffffff

t=3850.0 ps, M[000000cc] = 0x0fffffff

t=3860.0 ps, M[000000d0] = 0x07ffffff

t=3870.0 ps, M[000000d4] = 0x03ffffff

t=3880.0 ps, M[000000d8] = 0x01ffffff

t=3890.0 ps, M[000000dc] = 0x00ffffff

t=3900.0 ps, M[000000e0] = 0x007fffff

t=3910.0 ps, M[000000e4] = 0x003fffff

t=3920.0 ps, M[000000e8] = 0x001fffff

t=3930.0 ps, M[000000ec] = 0x000fffff

t=3940.0 ps, M[000000f0] = 0x0007ffff

t=3950.0 ps, M[000000f4] = 0x0003ffff

t=3960.0 ps, M[000000f8] = 0x0001ffff

t=3970.0 ps, M[000000fc] = 0x0000ffff

t=3980.0 ps, PC = 0x0040003c, IR = 0x0000000d

**Instruction Memory Module 3**

**Testing Instructions:** SRA

@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 # inc memory pointer by 4

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 done

08 10 00 0c // j exit # jump around a halt instr

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

- SRA instruction is done 16 times on register $r1 and changes original value from  
 0x8000FFFF to 0xFFFF8000

- Values inside the Memory Locations 0xC0 to 0xFF proves that both SRA and SW functions   
 work correctly.

- Since register $r14 has 0x5A5A3C3C, this proves that the all the instructions are working as

intended

BREAK INSTRUCTION FETCHED 3650.0 ps

R E G I S T E R ' S A F T E R B R E A K

t=3660.0 ps, $r0 = 0x00000000 || $r16 = 0xxxxxxxxx

t=3670.0 ps, $r1 = 0xffff8000 || $r17 = 0xxxxxxxxx

t=3680.0 ps, $r2 = 0x00000000 || $r18 = 0xxxxxxxxx

t=3690.0 ps, $r3 = 0xxxxxxxxx || $r19 = 0xxxxxxxxx

t=3700.0 ps, $r4 = 0xxxxxxxxx || $r20 = 0xxxxxxxxx

t=3710.0 ps, $r5 = 0xxxxxxxxx || $r21 = 0xxxxxxxxx

t=3720.0 ps, $r6 = 0xxxxxxxxx || $r22 = 0xxxxxxxxx

t=3730.0 ps, $r7 = 0xxxxxxxxx || $r23 = 0xxxxxxxxx

t=3740.0 ps, $r8 = 0xxxxxxxxx || $r24 = 0xxxxxxxxx

t=3750.0 ps, $r9 = 0xxxxxxxxx || $r25 = 0xxxxxxxxx

t=3760.0 ps, $r10 = 0xxxxxxxxx || $r26 = 0xxxxxxxxx

t=3770.0 ps, $r11 = 0xxxxxxxxx || $r27 = 0xxxxxxxxx

t=3780.0 ps, $r12 = 0xxxxxxxxx || $r28 = 0xxxxxxxxx

t=3790.0 ps, $r13 = 0xxxxxxxxx || $r29 = 0x000003fc

t=3800.0 ps, $r14 = 0x5a5a3c3c || $r30 = 0xxxxxxxxx

t=3810.0 ps, $r15 = 0x10010100 || $r31 = 0x00000000

Memory Locations 0xC0 to 0xFF

t=3820.0 ps, M[000000c0] = 0xc0007fff

t=3830.0 ps, M[000000c4] = 0xe0003fff

t=3840.0 ps, M[000000c8] = 0xf0001fff

t=3850.0 ps, M[000000cc] = 0xf8000fff

t=3860.0 ps, M[000000d0] = 0xfc0007ff

t=3870.0 ps, M[000000d4] = 0xfe0003ff

t=3880.0 ps, M[000000d8] = 0xff0001ff

t=3890.0 ps, M[000000dc] = 0xff8000ff

t=3900.0 ps, M[000000e0] = 0xffc0007f

t=3910.0 ps, M[000000e4] = 0xffe0003f

t=3920.0 ps, M[000000e8] = 0xfff0001f

t=3930.0 ps, M[000000ec] = 0xfff8000f

t=3940.0 ps, M[000000f0] = 0xfffc0007

t=3950.0 ps, M[000000f4] = 0xfffe0003

t=3960.0 ps, M[000000f8] = 0xffff0001

t=3970.0 ps, M[000000fc] = 0xffff8000

t=3980.0 ps, PC = 0x0040003c, IR = 0x0000000d

**Instruction Memory Module 4**

**Testing Instructions:** SLL, SLT

@0

3c 01 ff ff // main: lui $01, 0xFFFF

34 21 ff ff // ori $01, 0xFFFF # LI R01, 0xFFFFFFF

20 02 00 10 // addi $02, $00, 0x10 # LI R02, 0x10

3c 0f 10 01 // lui $15, 0x1001

35 ef 00 c0 // ori $15, 0x00C0 # LI R15, 0x100100C0

00 01 08 40 // top: sll $01, $01, 1 # logical shift left 1 bit

ad e1 00 00 // sw $01, 0($15) # ST [R15], R01

21 ef 00 04 // addi $15, $15, 4 # inc memory pointer by 4

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 instr

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

- SLL instruction is done 16 times on register $r1 and changes original value from  
 0xFFFFFFFF to 0xFFFF0000

- SLT instruction sets register $r3 to 1 if register $r0 is less than $r2 and is used with   
 conjunction with the ADDI and BNE instructions to loop 16 times.

- Values inside the Memory Locations 0xC0 to 0xFF proves that both SLL and SW functions   
 work correctly.

- Since register $r14 has 0x5A5A3C3C, this proves that the all the instructions are working as

intended

BREAK INSTRUCTION FETCHED 4290.0 ps

R E G I S T E R ' S A F T E R B R E A K

t=4300.0 ps, $r0 = 0x00000000 || $r16 = 0xxxxxxxxx

t=4310.0 ps, $r1 = 0xffff0000 || $r17 = 0xxxxxxxxx

t=4320.0 ps, $r2 = 0x00000000 || $r18 = 0xxxxxxxxx

t=4330.0 ps, $r3 = 0x00000000 || $r19 = 0xxxxxxxxx

t=4340.0 ps, $r4 = 0xxxxxxxxx || $r20 = 0xxxxxxxxx

t=4350.0 ps, $r5 = 0xxxxxxxxx || $r21 = 0xxxxxxxxx

t=4360.0 ps, $r6 = 0xxxxxxxxx || $r22 = 0xxxxxxxxx

t=4370.0 ps, $r7 = 0xxxxxxxxx || $r23 = 0xxxxxxxxx

t=4380.0 ps, $r8 = 0xxxxxxxxx || $r24 = 0xxxxxxxxx

t=4390.0 ps, $r9 = 0xxxxxxxxx || $r25 = 0xxxxxxxxx

t=4400.0 ps, $r10 = 0xxxxxxxxx || $r26 = 0xxxxxxxxx

t=4410.0 ps, $r11 = 0xxxxxxxxx || $r27 = 0xxxxxxxxx

t=4420.0 ps, $r12 = 0xxxxxxxxx || $r28 = 0xxxxxxxxx

t=4430.0 ps, $r13 = 0xxxxxxxxx || $r29 = 0x000003fc

t=4440.0 ps, $r14 = 0x5a5a3c3c || $r30 = 0xxxxxxxxx

t=4450.0 ps, $r15 = 0x10010100 || $r31 = 0x00000000

Memory Locations 0xC0 to 0xFF

t=4460.0 ps, M[000000c0] = 0xfffffffe

t=4470.0 ps, M[000000c4] = 0xfffffffc

t=4480.0 ps, M[000000c8] = 0xfffffff8

t=4490.0 ps, M[000000cc] = 0xfffffff0

t=4500.0 ps, M[000000d0] = 0xffffffe0

t=4510.0 ps, M[000000d4] = 0xffffffc0

t=4520.0 ps, M[000000d8] = 0xffffff80

t=4530.0 ps, M[000000dc] = 0xffffff00

t=4540.0 ps, M[000000e0] = 0xfffffe00

t=4550.0 ps, M[000000e4] = 0xfffffc00

t=4560.0 ps, M[000000e8] = 0xfffff800

t=4570.0 ps, M[000000ec] = 0xfffff000

t=4580.0 ps, M[000000f0] = 0xffffe000

t=4590.0 ps, M[000000f4] = 0xffffc000

t=4600.0 ps, M[000000f8] = 0xffff8000

t=4610.0 ps, M[000000fc] = 0xffff0000

t=4620.0 ps, PC = 0x00400040, IR = 0x0000000d

**Instruction Memory Module 5**

**Testing Instructions:** SLTI

@0

3c 01 ff ff // main: lui $01, 0xFFFF

34 21 ff ff // ori $01, 0xFFFF # LI R01, 0xFFFFFFF

20 02 ff f0 // addi $02, $00, -16 # LI R02, -16

3c 0f 10 01 // lui $15, 0x1001

35 ef 00 c0 // ori $15, 0x00C0 # LI R15, 0x100100C0

00 01 08 40 // top: sll $01, $01, 1 # logical shift left 1 bit

ad e1 00 00 // sw $01, 0($15) # ST [R15], R01

21 ef 00 04 // addi $15, $15, 4 # inc memory pointer by 4

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 instr

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

- SLL instruction is done 16 times on register $r1 and changes original value from  
 0xFFFFFFFF to 0xFFFF0000

- SLTI instruction sets register $r3 to 1 if register $r0 is less than $r2 and is used with   
 conjunction with the ADDI and BNE instructions to loop 16 times.

- Values inside the Memory Locations 0xC0 to 0xFF proves that both SLTI and SW functions   
 work correctly.

- Since register $r14 has 0x5A5A3C3C, this proves that the all the instructions are working as

intended

BREAK INSTRUCTION FETCHED 4290.0 ps

R E G I S T E R ' S A F T E R B R E A K

t=4300.0 ps, $r0 = 0x00000000 || $r16 = 0xxxxxxxxx

t=4310.0 ps, $r1 = 0xffff0000 || $r17 = 0xxxxxxxxx

t=4320.0 ps, $r2 = 0x00000000 || $r18 = 0xxxxxxxxx

t=4330.0 ps, $r3 = 0x00000000 || $r19 = 0xxxxxxxxx

t=4340.0 ps, $r4 = 0xxxxxxxxx || $r20 = 0xxxxxxxxx

t=4350.0 ps, $r5 = 0xxxxxxxxx || $r21 = 0xxxxxxxxx

t=4360.0 ps, $r6 = 0xxxxxxxxx || $r22 = 0xxxxxxxxx

t=4370.0 ps, $r7 = 0xxxxxxxxx || $r23 = 0xxxxxxxxx

t=4380.0 ps, $r8 = 0xxxxxxxxx || $r24 = 0xxxxxxxxx

t=4390.0 ps, $r9 = 0xxxxxxxxx || $r25 = 0xxxxxxxxx

t=4400.0 ps, $r10 = 0xxxxxxxxx || $r26 = 0xxxxxxxxx

t=4410.0 ps, $r11 = 0xxxxxxxxx || $r27 = 0xxxxxxxxx

t=4420.0 ps, $r12 = 0xxxxxxxxx || $r28 = 0xxxxxxxxx

t=4430.0 ps, $r13 = 0xxxxxxxxx || $r29 = 0x000003fc

t=4440.0 ps, $r14 = 0x5a5a3c3c || $r30 = 0xxxxxxxxx

t=4450.0 ps, $r15 = 0x10010100 || $r31 = 0x00000000

Memory Locations 0xC0 to 0xFF

t=4460.0 ps, M[000000c0] = 0xfffffffe

t=4470.0 ps, M[000000c4] = 0xfffffffc

t=4480.0 ps, M[000000c8] = 0xfffffff8

t=4490.0 ps, M[000000cc] = 0xfffffff0

t=4500.0 ps, M[000000d0] = 0xffffffe0

t=4510.0 ps, M[000000d4] = 0xffffffc0

t=4520.0 ps, M[000000d8] = 0xffffff80

t=4530.0 ps, M[000000dc] = 0xffffff00

t=4540.0 ps, M[000000e0] = 0xfffffe00

t=4550.0 ps, M[000000e4] = 0xfffffc00

t=4560.0 ps, M[000000e8] = 0xfffff800

t=4570.0 ps, M[000000ec] = 0xfffff000

t=4580.0 ps, M[000000f0] = 0xffffe000

t=4590.0 ps, M[000000f4] = 0xffffc000

t=4600.0 ps, M[000000f8] = 0xffff8000

t=4610.0 ps, M[000000fc] = 0xffff0000

t=4620.0 ps, PC = 0x00400040, IR = 0x0000000d

**Instruction Memory Module 6**

**Testing Instructions:** LW

@0

3c 0f 10 01 // lui $15, 0x1001

35 ef 00 00 // ori $15, 0x0000 # LI R15,0x10010000 dest ptr

3c 0e 10 01 // lui $14, 0x1001

35 ce 00 c0 // ori $14, 0x00C0 # LI R14,0x100100C0 dest ptr

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

- Registers $r1 to $r12 are loaded in data from dM[0x04 to 0x33] using LW

- Registers from dM[0x00 to 0x3F] are transferred to dM[0xC0 to 0xFF] using LW and SW in   
 a loop

- Loops 16 times using ADDI and BNE

- Values inside the Memory Locations 0xC0 to 0xFF proves that all the instructions are   
 working as intended

BREAK INSTRUCTION FETCHED 4940.0 ps

R E G I S T E R ' S A F T E R B R E A K

t=4950.0 ps, $r0 = 0x00000000 || $r16 = 0xxxxxxxxx

t=4960.0 ps, $r1 = 0x12345678 || $r17 = 0x000075cc

t=4970.0 ps, $r2 = 0x89abcdef || $r18 = 0xxxxxxxxx

t=4980.0 ps, $r3 = 0xa5a5a5a5 || $r19 = 0xxxxxxxxx

t=4990.0 ps, $r4 = 0x5a5a5a5a || $r20 = 0xxxxxxxxx

t=5000.0 ps, $r5 = 0x2468ace0 || $r21 = 0xxxxxxxxx

t=5010.0 ps, $r6 = 0x13579bdf || $r22 = 0xxxxxxxxx

t=5020.0 ps, $r7 = 0x0f0f0f0f || $r23 = 0xxxxxxxxx

t=5030.0 ps, $r8 = 0xf0f0f0f0 || $r24 = 0xxxxxxxxx

t=5040.0 ps, $r9 = 0x00000009 || $r25 = 0xxxxxxxxx

t=5050.0 ps, $r10 = 0x0000000a || $r26 = 0xxxxxxxxx

t=5060.0 ps, $r11 = 0x0000000b || $r27 = 0xxxxxxxxx

t=5070.0 ps, $r12 = 0x0000000c || $r28 = 0xxxxxxxxx

t=5080.0 ps, $r13 = 0x00000000 || $r29 = 0x000003fc

t=5090.0 ps, $r14 = 0x10010100 || $r30 = 0xxxxxxxxx

t=5100.0 ps, $r15 = 0x10010040 || $r31 = 0x00000000

Memory Locations 0xC0 to 0xFF

t=5110.0 ps, M[000000c0] = 0xc3c3c3c3

t=5120.0 ps, M[000000c4] = 0x12345678

t=5130.0 ps, M[000000c8] = 0x89abcdef

t=5140.0 ps, M[000000cc] = 0xa5a5a5a5

t=5150.0 ps, M[000000d0] = 0x5a5a5a5a

t=5160.0 ps, M[000000d4] = 0x2468ace0

t=5170.0 ps, M[000000d8] = 0x13579bdf

t=5180.0 ps, M[000000dc] = 0x0f0f0f0f

t=5190.0 ps, M[000000e0] = 0xf0f0f0f0

t=5200.0 ps, M[000000e4] = 0x00000009

t=5210.0 ps, M[000000e8] = 0x0000000a

t=5220.0 ps, M[000000ec] = 0x0000000b

t=5230.0 ps, M[000000f0] = 0x0000000c

t=5240.0 ps, M[000000f4] = 0x0000000d

t=5250.0 ps, M[000000f8] = 0xfffffff8

t=5260.0 ps, M[000000fc] = 0x000075cc

t=5270.0 ps, PC = 0x00000060, IR = 0x0000000d

**Instruction Memory Module 7**

**Testing Instructions:** JAL, JR

@0

3c 0f 10 01 // main: lui $15, 0x1001

35 ef 00 00 // ori $15, 0x0000 # LI R15, 0x10010000 dest

3c 0e 10 01 // lui $14, 0x1001

35 ce 00 c0 // ori $14, 0x00C0 # LI R14, 0x100100C0 dest

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 til

03 e0 00 08 // jr $31 # return to old PC

00 00 00 0d // break # safety net

- The JAL instruction writes current PC to $r31 and jumps to mem2mem subroutine

- The JR instruction jumps back to the old saved PC in register $r31

- Registers $r1 to $r12 are loaded in data from dM[0x04 to 0x33] using LW

- Registers from dM[0x00 to 0x3F] are transferred to dM[0xC0 to 0xFF] using LW and SW in   
 a loop

- Loops 16 times using ADDI and BNE

BREAK INSTRUCTION FETCHED 5100.0 ps

R E G I S T E R ' S A F T E R B R E A K

t=5110.0 ps, $r0 = 0x00000000 || $r16 = 0xxxxxxxxx

t=5120.0 ps, $r1 = 0x12345678 || $r17 = 0x000075cc

t=5130.0 ps, $r2 = 0x89abcdef || $r18 = 0xxxxxxxxx

t=5140.0 ps, $r3 = 0xa5a5a5a5 || $r19 = 0xxxxxxxxx

t=5150.0 ps, $r4 = 0x5a5a5a5a || $r20 = 0xxxxxxxxx

t=5160.0 ps, $r5 = 0x2468ace0 || $r21 = 0xxxxxxxxx

t=5170.0 ps, $r6 = 0x13579bdf || $r22 = 0xxxxxxxxx

t=5180.0 ps, $r7 = 0x0f0f0f0f || $r23 = 0xxxxxxxxx

t=5190.0 ps, $r8 = 0xf0f0f0f0 || $r24 = 0xxxxxxxxx

t=5200.0 ps, $r9 = 0x00000009 || $r25 = 0xxxxxxxxx

t=5210.0 ps, $r10 = 0x0000000a || $r26 = 0xxxxxxxxx

t=5220.0 ps, $r11 = 0x0000000b || $r27 = 0xxxxxxxxx

t=5230.0 ps, $r12 = 0x0000000c || $r28 = 0xxxxxxxxx

t=5240.0 ps, $r13 = 0x00000000 || $r29 = 0x000003fc

t=5250.0 ps, $r14 = 0x10010100 || $r30 = 0xxxxxxxxx

t=5260.0 ps, $r15 = 0xffffffff || $r31 = 0x00000048

Memory Locations 0xC0 to 0xFF

t=5270.0 ps, M[000000c0] = 0xc3c3c3c3

t=5280.0 ps, M[000000c4] = 0x12345678

t=5290.0 ps, M[000000c8] = 0x89abcdef

t=5300.0 ps, M[000000cc] = 0xa5a5a5a5

t=5310.0 ps, M[000000d0] = 0x5a5a5a5a

t=5320.0 ps, M[000000d4] = 0x2468ace0

t=5330.0 ps, M[000000d8] = 0x13579bdf

t=5340.0 ps, M[000000dc] = 0x0f0f0f0f

t=5350.0 ps, M[000000e0] = 0xf0f0f0f0

t=5360.0 ps, M[000000e4] = 0x00000009

t=5370.0 ps, M[000000e8] = 0x0000000a

t=5380.0 ps, M[000000ec] = 0x0000000b

t=5390.0 ps, M[000000f0] = 0x0000000c

t=5400.0 ps, M[000000f4] = 0x0000000d

t=5410.0 ps, M[000000f8] = 0xfffffff8

t=5420.0 ps, M[000000fc] = 0x000075cc

t=5430.0 ps, PC = 0x00000054, IR = 0x0000000d

**Instruction Memory Module 8**

**Testing Instructions:** MULT, MFLO, MFHI

@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

00 00 00 0d // break # (Pass flag)

/\* Rest of ‘fail flags’ on the next page \*/

BREAK INSTRUCTION FETCHED 1190.0 ps

R E G I S T E R ' S A F T E R B R E A K

t=1200.0 ps, $r0 = 0x00000000 || $r16 = 0xxxxxxxxx

t=1210.0 ps, $r1 = 0x00000019 || $r17 = 0xxxxxxxxx

t=1220.0 ps, $r2 = 0x000003e8 || $r18 = 0xxxxxxxxx

t=1230.0 ps, $r3 = 0xffffffe7 || $r19 = 0xxxxxxxxx

t=1240.0 ps, $r4 = 0xfffffc18 || $r20 = 0xxxxxxxxx

t=1250.0 ps, $r5 = 0x000061a8 || $r21 = 0xxxxxxxxx

t=1260.0 ps, $r6 = 0xffff9e58 || $r22 = 0xxxxxxxxx

t=1270.0 ps, $r7 = 0xffffffff || $r23 = 0xxxxxxxxx

t=1280.0 ps, $r8 = 0x000061a8 || $r24 = 0xxxxxxxxx

t=1290.0 ps, $r9 = 0xffff9e58 || $r25 = 0xxxxxxxxx

t=1300.0 ps, $r10 = 0xffffffff || $r26 = 0xxxxxxxxx

t=1310.0 ps, $r11 = 0xffff9e58 || $r27 = 0xxxxxxxxx

t=1320.0 ps, $r12 = 0xffffffff || $r28 = 0xxxxxxxxx

t=1330.0 ps, $r13 = 0x000061a8 || $r29 = 0x000003fc

t=1340.0 ps, $r14 = 0x00000000 || $r30 = 0xxxxxxxxx

t=1350.0 ps, $r15 = 0x10010000 || $r31 = 0x00000000

Memory Locations 0xC0 to 0xFF

t=1360.0 ps, M[000000c0] = 0xxxxxxxxx

t=1370.0 ps, M[000000c4] = 0xxxxxxxxx

t=1380.0 ps, M[000000c8] = 0xxxxxxxxx

t=1390.0 ps, M[000000cc] = 0xxxxxxxxx

t=1400.0 ps, M[000000d0] = 0xxxxxxxxx

t=1410.0 ps, M[000000d4] = 0xxxxxxxxx

t=1420.0 ps, M[000000d8] = 0xxxxxxxxx

t=1430.0 ps, M[000000dc] = 0xxxxxxxxx

t=1440.0 ps, M[000000e0] = 0xxxxxxxxx

t=1450.0 ps, M[000000e4] = 0xxxxxxxxx

t=1460.0 ps, M[000000e8] = 0xxxxxxxxx

t=1470.0 ps, M[000000ec] = 0xxxxxxxxx

t=1480.0 ps, M[000000f0] = 0xxxxxxxxx

t=1490.0 ps, M[000000f4] = 0xxxxxxxxx

t=1500.0 ps, M[000000f8] = 0xxxxxxxxx

t=1510.0 ps, M[000000fc] = 0xxxxxxxxx

t=1520.0 ps, PC = 0x00000070, IR = 0x0000000d

**Instruction Memory Module 8 (page 2)**

**Testing Instructions:** MULT, MFLO, MFHI

3c 0e ff ff // fail1: lui $14, 0xFFFF

35 ce ff ff // ori $14, 0xFFFF # $r14 <-- 0xFFFFFFFF

00 00 00 0d // break # (Fail flag 1)

3c 0e ff ff // fail2L: lui $14, 0xFFFF

35 ce ff fe // ori $14, 0xFFFE # $r14 <-- 0xFFFFFFFE

00 00 00 0d // break # (Fail flag 2L)

3c 0e ff ff // fail2H: lui $14, 0xFFFF

35 ce ff fd // ori $14, 0xFFFD # $r14 <--0xFFFFFFFD   
00 00 00 0d // break # (Fail flag 2H)

3c 0e ff ff // fail3L: lui $14, 0xFFFF

35 ce ff fc // ori $14, 0xFFFC # $r14 <-- 0xFFFFFFFC

00 00 00 0d // break # (Fail flag 3L)

3c 0e ff ff // fail3H: lui $14, 0xFFFF

35 ce ff fb // ori $14, 0xFFFB # $r14 <-- 0xFFFFFFFB

00 00 00 0d // break # (Fail flag 3H)

3c 0e ff ff // fail4: lui $14, 0xFFFF

35 ce ff fa // ori $14, 0xFFFA # $r14 <-- 0xFFFFFFFA

00 00 00 0d // break # (Fail flag 4)

- Using LW registers $r1 to $r7 get their respective values from the Data Memory  
- The first use of the MULT instruction multiplies the value from $r1 and $r2 together and   
 stores the lower 32-bit value of the product into register $r8 using the MFLO instruction.  
 The BNE instruction checks the product if it has the same value in register $r5 and branches   
 to the fail flag if they are not equal.  
- The second use of the MULT instruction multiplies the value from $r3 and $r2 together and  
 stores the lower 32-bit value of the product into register $r9 using the MFLO instruction,   
 and stores the upper 32-bit value of the product into register $r10 using the MFHI   
 instruction. The BNE instruction checks if $r9 is equal to $r6 and if $r10 is equal to $r7 and  
 branches to their respective fail flags if not.  
- Since register $r14 has 0x00000000, this proves that the all the instructions are working as

intended

**Instruction Memory Module 9**

**Testing Instructions:** XOR, AND, OR, NOR, SLTU

@0

3c 0f 10 01 // main: lui $15, 0x1001

35 ef 00 c0 // ori $15, 0x00C0 # $r15 <-- 0x100100C0   
 # (destptr)

20 01 ff 8a // addi $01, $00, -118 # $r01 <-- 0xFFFFFF8A

20 02 00 8a // addi $02 $00, 138 # $r02 <-- 0x0000008A

0c 10 00 22 // jal slt\_tests

3c 0d 77 88 // lui $13, 0x7788

35 ad 77 88 // ori $13, 0x7788 # $r13 <-- 0x77887788   
 # (pattern1)

3c 0c 88 77 // lui $12, 0x8877

35 8c 88 77 // ori $12, 0x8877 # $r12 <-- 0x88778877   
 # (pattern2)

3c 0b ff ff // lui $11, 0xFFFF

35 6b ff ff // ori $11, 0xFFFF # $r11 <-- 0xFFFFFFFF   
 # (pattern3)

01 ac 50 26 // xor $10, $13, $12 # $r10 <-- 0xFFFFFFFF

11 4b 00 02 // beq $10, $11, xor\_pass

20 0e ff fb // addi $14, $00, -5 # r14 <-- 0xFFFF\_FFFB

00 00 00 0d // break # ^(fail flag 5)

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 # r14 <-- 0xFFFF\_FFFA

00 00 00 0d // break # ^(fail flag 6)

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 # r14 <-- 0xFFFF\_FFF9

00 00 00 0d // break # ^(fail flag 7)

01 e2 48 27 // or\_pass: nor $09, $15, $02 # $r09 <-- 0xEFFEFF35

3c 08 ef fe // lui $08, 0xEFFE

35 08 ff 35 // ori $08, 0xFF35 # $r08 <-- 0xEFFEFF35

11 09 00 02 // beq $08, $09, nor\_pass

20 0e ff f8 // addi $14, $00, -8 # r14 <-- 0xFFFF\_FFF8

00 00 00 0d // break # ^(fail flag 8)

ad e8 00 10 // nor\_pass: sw $08, 0x10($15) # M[D0] <-- 0xEFFEFF35

00 00 70 20 // add $14, $00, $00 # clear r14 "passed all"

00 00 00 0d // break # should stop here,

// # having completed all   
 // # the tests

BREAK INSTRUCTION FETCHED 1820.0 ps

R E G I S T E R ' S A F T E R B R E A K

t=1830.0 ps, $r0 = 0x00000000 || $r16 = 0xxxxxxxxx

t=1840.0 ps, $r1 = 0xffffff8a || $r17 = 0xxxxxxxxx

t=1850.0 ps, $r2 = 0x0000008a || $r18 = 0xxxxxxxxx

t=1860.0 ps, $r3 = 0x00000000 || $r19 = 0xxxxxxxxx

t=1870.0 ps, $r4 = 0x000000c0 || $r20 = 0xxxxxxxxx

t=1880.0 ps, $r5 = 0x000000c4 || $r21 = 0xxxxxxxxx

t=1890.0 ps, $r6 = 0x000000c8 || $r22 = 0xxxxxxxxx

t=1900.0 ps, $r7 = 0x000000cc || $r23 = 0xxxxxxxxx

t=1910.0 ps, $r8 = 0xeffeff35 || $r24 = 0xxxxxxxxx

t=1920.0 ps, $r9 = 0xeffeff35 || $r25 = 0xxxxxxxxx

t=1930.0 ps, $r10 = 0xffffffff || $r26 = 0xxxxxxxxx

t=1940.0 ps, $r11 = 0xffffffff || $r27 = 0xxxxxxxxx

t=1950.0 ps, $r12 = 0x88778877 || $r28 = 0xxxxxxxxx

t=1960.0 ps, $r13 = 0x77887788 || $r29 = 0x000003fc

t=1970.0 ps, $r14 = 0x00000000 || $r30 = 0xxxxxxxxx

t=1980.0 ps, $r15 = 0x100100c0 || $r31 = 0x00000014

Memory Locations 0xC0 to 0xFF

t=1990.0 ps, M[000000c0] = 0x000000c0

t=2000.0 ps, M[000000c4] = 0x000000c4

t=2010.0 ps, M[000000c8] = 0x000000c8

t=2020.0 ps, M[000000cc] = 0x000000cc

t=2030.0 ps, M[000000d0] = 0xeffeff35

t=2040.0 ps, M[000000d4] = 0xxxxxxxxx

t=2050.0 ps, M[000000d8] = 0xxxxxxxxx

t=2060.0 ps, M[000000dc] = 0xxxxxxxxx

t=2070.0 ps, M[000000e0] = 0xxxxxxxxx

t=2080.0 ps, M[000000e4] = 0xxxxxxxxx

t=2090.0 ps, M[000000e8] = 0xxxxxxxxx

t=2100.0 ps, M[000000ec] = 0xxxxxxxxx

t=2110.0 ps, M[000000f0] = 0xxxxxxxxx

t=2120.0 ps, M[000000f4] = 0xxxxxxxxx

t=2130.0 ps, M[000000f8] = 0xxxxxxxxx

t=2140.0 ps, M[000000fc] = 0xxxxxxxxx

t=2150.0 ps, PC = 0x00000088, IR = 0x0000000d

**Instruction Memory Module 9 (page 2)**

**Testing Instructions:** XOR, AND, OR, NOR, SLTU

00 22 18 2a // slt\_tests: slt $03, $01, $02 # for signed# r01 < r02

14 60 00 02 // bne $03, $00, slt1 # thus, should branch

20 0e ff ff // addi $14, $00, -1 # r14 <-- FFFF\_FFFF

00 00 00 0d // break # ^(fail flag 1)

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, should branch

20 0e ff fe // addi $14, $00, -2 # r14 <-- FFFF\_FFFE

00 00 00 0d // break # ^(fail flag 2)

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, should branch

20 0e ff fd // addi $14, $00, -3 # r14 <-- FFFF\_FFFD

00 00 00 0d // break # ^(fail flag 3)

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, should branch

20 0e ff fc // addi $14, $00, -4 # r14 <-- FFFF\_FFFC

00 00 00 0d // break # ^(fail flag 4)

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

**-** The SLT and SLTU instruction are checked in their own subroutine reached by the JAL   
 instruction and confirmed to be working correctly according to the pass flags in memory   
 locations 0xC0 to 0xCC

- The XOR, AND, OR and NOR instructions are verified with memory location 0xD0 having   
 the value 0xEFFEFF35 and the register $r14 having the value 0x00000000

**Instruction Memory Module 10**

**Testing Instructions:** DIV

@0

3c 0f 10 01 // main: lui $15, 0x1001

35 ef 00 00 // ori $15, 0x0000 # $r15 <-- 0x10010000(src)

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

8d e6 00 14 // lw $06, 20($15) # $r06 <-- 465

8d e7 00 18 // lw $07, 24($15) # $r07 <-- -264

8d e8 00 1c // lw $08, 28($15) # $r08 <-- -465

00 22 00 1a // div $01, $02

00 00 48 12 // mflo $09 # $r09 <-- 264   
00 00 50 10 // mfhi $10 # $r10 <-- 465

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 # $r09 <-- -264

00 00 50 10 // mfhi $10 # $r10 <-- -465

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 # $r09 <-- -264

00 00 50 10 // mfhi $10 # $r10 <-- 465

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 # $r09 <-- 264

00 00 50 10 // mfhi $10 # $r10 <-- -465

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)

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

/\* Rest of ‘fail flags’ on the next page \*/

BREAK INSTRUCTION FETCHED 1500.0 ps

R E G I S T E R ' S A F T E R B R E A K

t=1510.0 ps, $r0 = 0x00000000 || $r16 = 0xxxxxxxxx

t=1520.0 ps, $r1 = 0x00040911 || $r17 = 0xxxxxxxxx

t=1530.0 ps, $r2 = 0x000003e8 || $r18 = 0xxxxxxxxx

t=1540.0 ps, $r3 = 0xfffbf6ef || $r19 = 0xxxxxxxxx

t=1550.0 ps, $r4 = 0xfffffc18 || $r20 = 0xxxxxxxxx

t=1560.0 ps, $r5 = 0x00000108 || $r21 = 0xxxxxxxxx

t=1570.0 ps, $r6 = 0x000001d1 || $r22 = 0xxxxxxxxx

t=1580.0 ps, $r7 = 0xfffffef8 || $r23 = 0xxxxxxxxx

t=1590.0 ps, $r8 = 0xfffffe2f || $r24 = 0xxxxxxxxx

t=1600.0 ps, $r9 = 0x00000108 || $r25 = 0xxxxxxxxx

t=1610.0 ps, $r10 = 0xfffffe2f || $r26 = 0xxxxxxxxx

t=1620.0 ps, $r11 = 0x00000000 || $r27 = 0xxxxxxxxx

t=1630.0 ps, $r12 = 0x00000000 || $r28 = 0xxxxxxxxx

t=1640.0 ps, $r13 = 0x00000000 || $r29 = 0x000003fc

t=1650.0 ps, $r14 = 0x00000000 || $r30 = 0xxxxxxxxx

t=1660.0 ps, $r15 = 0x10010000 || $r31 = 0x00000000

Memory Locations 0xC0 to 0xFF

t=1670.0 ps, M[000000c0] = 0xxxxxxxxx

t=1680.0 ps, M[000000c4] = 0xxxxxxxxx

t=1690.0 ps, M[000000c8] = 0xxxxxxxxx

t=1700.0 ps, M[000000cc] = 0xxxxxxxxx

t=1710.0 ps, M[000000d0] = 0xxxxxxxxx

t=1720.0 ps, M[000000d4] = 0xxxxxxxxx

t=1730.0 ps, M[000000d8] = 0xxxxxxxxx

t=1740.0 ps, M[000000dc] = 0xxxxxxxxx

t=1750.0 ps, M[000000e0] = 0xxxxxxxxx

t=1760.0 ps, M[000000e4] = 0xxxxxxxxx

t=1770.0 ps, M[000000e8] = 0xxxxxxxxx

t=1780.0 ps, M[000000ec] = 0xxxxxxxxx

t=1790.0 ps, M[000000f0] = 0xxxxxxxxx

t=1800.0 ps, M[000000f4] = 0xxxxxxxxx

t=1810.0 ps, M[000000f8] = 0xxxxxxxxx

t=1820.0 ps, M[000000fc] = 0xxxxxxxxx

t=1830.0 ps, PC = 0x00000090, IR = 0x0000000d

**Instruction Memory Module 10 (page 2)**

**Testing Instructions:** DIV

3c 0e ff ff // fail1Q: lui $14, 0xFFFF

35 ce ff ff // ori $14, 0xFFFF # $r14 <-- 0xFFFFFFFF

00 00 00 0d // break # ^(Fail flag 1 Quot)

3c 0e ff ff // fail1R: lui $14, 0xFFFF

35 ce ff fe // ori $14, 0xFFFE # $r14 <-- 0xFFFFFFFE

00 00 00 0d // break # ^(Fail flag 1 Rem)

3c 0e ff ff // fail2Q: lui $14, 0xFFFF

35 ce ff fd // ori $14, 0xFFFD # $r14 <-- 0xFFFFFFFD

00 00 00 0d // break # ^(Fail flag 2 Quot)

3c 0e ff ff // fail2R: lui $14, 0xFFFF

35 ce ff fc // ori $14, 0xFFFC # $r14 <-- 0xFFFFFFFC

00 00 00 0d // break # ^(Fail flag 2 Rem)

3c 0e ff ff // fail3Q: lui $14, 0xFFFF

35 ce ff fb // ori $14, 0xFFFB # $r14 <-- 0xFFFFFFFB

00 00 00 0d // break # ^(Fail flag 3 Quot)

3c 0e ff ff // fail3R: lui $14, 0xFFFF

35 ce ff fa // ori $14, 0xFFFA # $r14 <-- 0xFFFFFFFA

00 00 00 0d // break # ^(Fail flag 3 Rem)

3c 0e ff ff // fail4Q: lui $14, 0xFFFF

35 ce ff f9 // ori $14, 0xFFF9 # $r14 <-- 0xFFFFFFF9

00 00 00 0d // break # ^(Fail flag 4 Quot)

3c 0e ff ff // fail4R: lui $14, 0xFFFF

35 ce ff f8 // ori $14, 0xFFF8 # $r14 <-- 0xFFFFFFF8

00 00 00 0d // break # (Fail flag 4 Rem)

- Using LW registers $r1 to $r8 get their respective values from the Data Memory  
- The first use of the DIV instruction divides the value from $r1 to $r2 and   
 stores the lower 32-bit value of the quotient into register $r9 using the MFLO instruction and   
 stores the upper 32-bit value of the quotient into register $r10 using the MFHI instruction.   
 The BNE instruction checks if $r9 is equal to $r5 and if $r10 is equal to $r6 and  
 branches to their respective fail flags if not.

- The DIV instruction divides $r3 and $r4 and stores the lower 32-bit on register $r9 and stores   
 the upper 32-bit value into register $r10. The BNE instruction then checks if $r9 is equal   
 to $r5 and if $r10 is equal to $r8 and branches to their respective fail flags if not.  
- Since registers $r11 to $r14 all have the value 0x00000000, this proves that the all the   
 instructions are working as intended and avoided all the previous fail flags.

**Instruction Memory Module 11**

**Testing Instructions:** XORI, SUB, ANDI, SLTIU

@0

3c 0f 10 01 // main: lui $15, 0x1001

35 ef 00 c0 // ori $15, 0x00C0 # $r15 <--0x100100C0(src)

20 01 ff 8a // addi $01, $00, -118 # $r01 <-- 0xFFFFFF8A

20 02 00 8a // addi $02 $00, 138 # $r02 <-- 0x0000008A

0c 10 00 1a // jal sltiu\_tests

3c 0d ff ff // lui $13, 0xFFFF # $r13 <-- 0xFFFF5555

35 ad 55 55 // ori $13, 0x5555 # ^(pattern1)

3c 0c ff ff // lui $12, 0xFFFF # $r12 <-- 0xFFFFFAF5

35 8c fa f5 // ori $12, 0xFAF5 # ^(pattern2)

3c 0b ff ff // lui $11, 0xFFFF # $r11 <-- 0xFFFFFFFF

35 6b ff ff // ori $11, 0xFFFF # ^(pattern3)

3c 0a 00 00 // lui $10, 0x0000 # $r10 <-- 0x0000F0F0

35 4a f0 f0 // ori $10, 0xF0F0 # ^(pattern4)

39 a9 aa aa // xori $09, $13, 0xAAAA # $r09 <-- 0xFFFFFFFF

01 2b 40 22 // sub $08, $09, $11 # $r08 <-- 0

11 00 00 02 // beq $08, $00, xor\_p1 # should branch

20 0e ff f9 // addi $14, $00, -7 # $r14 <-- FFFF\_FFF9

00 00 00 0d // break # ^(fail flag7)

31 87 f5 fa // xor\_p1: andi $07, $12, 0xF5FA # $r07 <-- 0x0000F0F0

00 ea 40 22 // sub $08, $07, $10

11 00 00 02 // beq $08, $00, xor\_p2 # should branch

20 0e ff f8 // addi $14, $00, -8 # $r14 <-- FFFF\_FFF8

00 00 00 0d // break # ^(fail flag8)

ad e1 00 18 // xor\_p2: sw $01, 0x18($15) # M[D8] <-- FFFFFF8A

00 00 00 0d // break # should stop here,

# having completed all   
 # the tests

// sltiu\_tests:

2c 23 ff 8b // sltiu $03, $01, -117 # if r1 < se(0xFF8B) uns

14 60 00 02 // bne $03, $00, slt1\_p1 # should branch

20 0e ff ff // addi $14, $00, -1 # $r14 <-- FFFF\_FFFF

00 00 00 0d // break # ^(fail flag1)

20 04 00 c0 // slt1\_p1: addi $04, $00,0xC0 # pass flag1 M[C0]<--C0

ad e4 00 00 // sw $04, 0x00($15)

BREAK INSTRUCTION FETCHED 1980.0 ps

R E G I S T E R ' S A F T E R B R E A K

t=1990.0 ps, $r0 = 0x00000000 || $r16 = 0xxxxxxxxx

t=2000.0 ps, $r1 = 0xffffff8a || $r17 = 0xxxxxxxxx

t=2010.0 ps, $r2 = 0x0000008a || $r18 = 0xxxxxxxxx

t=2020.0 ps, $r3 = 0x00000000 || $r19 = 0xxxxxxxxx

t=2030.0 ps, $r4 = 0x000000c0 || $r20 = 0xxxxxxxxx

t=2040.0 ps, $r5 = 0x000000c4 || $r21 = 0xxxxxxxxx

t=2050.0 ps, $r6 = 0x000000d4 || $r22 = 0xxxxxxxxx

t=2060.0 ps, $r7 = 0x0000f0f0 || $r23 = 0xxxxxxxxx

t=2070.0 ps, $r8 = 0x00000000 || $r24 = 0xxxxxxxxx

t=2080.0 ps, $r9 = 0xffffffff || $r25 = 0xxxxxxxxx

t=2090.0 ps, $r10 = 0x0000f0f0 || $r26 = 0xxxxxxxxx

t=2100.0 ps, $r11 = 0xffffffff || $r27 = 0xxxxxxxxx

t=2110.0 ps, $r12 = 0xfffffaf5 || $r28 = 0xxxxxxxxx

t=2120.0 ps, $r13 = 0xffff5555 || $r29 = 0x000003fc

t=2130.0 ps, $r14 = 0x00000000 || $r30 = 0xxxxxxxxx

t=2140.0 ps, $r15 = 0x100100c0 || $r31 = 0x00000014

Memory Locations 0xC0 to 0xFF

t=2150.0 ps, M[000000c0] = 0x000000c0

t=2160.0 ps, M[000000c4] = 0x000000c4

t=2170.0 ps, M[000000c8] = 0x000000c8

t=2180.0 ps, M[000000cc] = 0x000000cc

t=2190.0 ps, M[000000d0] = 0x000000d0

t=2200.0 ps, M[000000d4] = 0x000000d4

t=2210.0 ps, M[000000d8] = 0xffffff8a

t=2220.0 ps, M[000000dc] = 0xxxxxxxxx

t=2230.0 ps, M[000000e0] = 0xxxxxxxxx

t=2240.0 ps, M[000000e4] = 0xxxxxxxxx

t=2250.0 ps, M[000000e8] = 0xxxxxxxxx

t=2260.0 ps, M[000000ec] = 0xxxxxxxxx

t=2270.0 ps, M[000000f0] = 0xxxxxxxxx

t=2280.0 ps, M[000000f4] = 0xxxxxxxxx

t=2290.0 ps, M[000000f8] = 0xxxxxxxxx

t=2300.0 ps, M[000000fc] = 0xxxxxxxxx

t=2310.0 ps, PC = 0x00000064, IR = 0x0000000d

**Instruction Memory Module 11**

**Testing Instructions:** XORI, SUB, ANDI, SLTIU

2c 23 ff 89 // sltiu $03, $01, -119 # if r1 !<se(0xFF89) uns

10 60 00 02 // beq $03, $00, slt\_p2 # should branch

20 0e ff fe // addi $14, $00, -2 # $r14 <-- FFFF\_FFFE

00 00 00 0d // break # ^(fail flag2)

20 05 00 c4 // slt\_p2: addi $05, $00, 0xC4 # pass flag2 M[C4]<-- C4

ad e5 00 04 // sw $05, 0x04($15)

2c 23 ff 8a // sltiu $03, $01, -118 # if r1 !< se(0xFF8A) uns

10 60 00 02 // beq $03, $00, slt\_p3 # should branch

20 0e ff fd // addi $14, $00, -3 # r14 <-- FFFF\_FFFD

00 00 00 0d // break # ^(fail flag3)

20 06 00 c8 // slt\_p3: addi $06, $00, 0xC8# pass flag3 M[C8] <-- C8

ad e6 00 08 // sw $06, 0x08($15)

2c 43 00 8b // sltiu $03, $02, 0x008B # if r2 < se(0x008B) uns

14 60 00 02 // bne $03, $00, slt1\_p4 # should branch

20 0e ff fc // addi $14, $00, -4 # r14 <-- FFFF\_FFFC

00 00 00 0d // break # ^(fail flag4)

20 07 00 cc // slt1\_p4: addi $07, $00, 0xCC# pass flag4 M[CC] <-- CC

ad e7 00 0c // sw $07, 0x0C($15)

2c 43 00 89 // sltiu $03, $02, 0x0089 # if r2 !< se(0x0089) uns

10 60 00 02 // beq $03, $00, slt\_p5 # should branch

20 0e ff fb // addi $14, $00, -5 # r14 <-- FFFF\_FFFB

00 00 00 0d // break # ^(fail flag5)

20 08 00 d0 // slt\_p5: addi $08, $00, 0xD0# pass flag5 M[D0] <-- D0

ad e8 00 10 // sw $08 0x10($15)

2c 43 00 8a // sltiu $03, $02, 0x008A # if r2 !< se(0x008A) uns

10 60 00 02 // beq $03, $00, slt\_p6 # should branch

20 0e ff fa // addi $14, $00, -6 # r14 <-- FFFF\_FFFA

00 00 00 0d // break # ^(fail flag6)

20 06 00 d4 // slt\_p6: addi $06, $00, 0xD4 # pass flag6 M[D4] <-- D4

ad e6 00 14 // sw $06, 0x14($15)

20 0e 00 00 // addi $14, $00, 0 # set $r14 to 0000\_0000

03 e0 00 08 // jr $31 # return from subroutine

**-** The SLTIU instruction is checked in its own subroutine reached by the JAL   
 instruction and confirmed to be working correctly according to the pass flags in memory   
 locations 0xC0 to 0xD4

- The XORI, SUB and ANDI instructions are verified with memory location 0xD8 having   
 the value 0xFFFFFF8A and the register $r14 having the value 0x00000000

**Instruction Memory Module 12**

**Testing Instructions:** BLEZ, BGTZ

@0

3c 0f 10 01 // main: lui $15, 0x1001

35 ef 00 c0 // ori $15, 0x00C0 # $r15 <-- 0x100100C0 (dest pointer)

20 01 ff 8a // addi $01, $00, -118 # $r01 <-- 0xFFFFFF8A

20 02 00 8a // addi $02 $00, 138 # $r02 <-- 0x0000008A

0c 10 00 08 // jal blt\_tests

ad e1 00 18 // sw $01, 0x18($15) # M[D8] <-- 0xFFFFFF8A

ad e2 00 1c // sw $02, 0x1C($15) # M[DC] <-- 0x0000008A

00 00 00 0d // break

18 20 00 02 // blt\_tests: blez $01, blez\_p1 # this should branch

20 0e ff ff // addi $14, $00, -1 # r14 <-- FFFF\_FFFF

00 00 00 0d // break ^(fail flag)

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 # r14 <-- FFFF\_FFFE

00 00 00 0d // break ^(fail flag2)

18 00 00 02 // blez\_p2: blez $0, blez\_p3 # this should branch

20 0e ff fd // addi $14, $00, -3 #failflag3 r14<-FFFFFFFD

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 # r14 <-- FFFF\_FFFC

00 00 00 0d // break ^(fail flag3)

20 06 00 cc // bgtz\_p1: addi $06, $00, 0xCC # pass flag4 M[CC]<-- 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 # r14 <-- FFFF\_FFFB

00 00 00 0d // break ^(fail flag5)

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[D4]<-- D4

ad e8 00 14 // sw $08, 0x14($15)

08 10 00 29 // j bgtz\_p3

20 0e ff fa // bgtz\_f3: addi $14, $00, -6 # r14 <-- FFFF\_FFFA

00 00 00 0d // break ^(fail flag6)

20 0e 00 00 // bgtz\_p3: addi $14, $00, 0 # set $r14 to 0000\_0000

03 e0 00 08 // jr $31 # return from subroutine

BREAK INSTRUCTION FETCHED 1310.0 ps

R E G I S T E R ' S A F T E R B R E A K

t=1320.0 ps, $r0 = 0x00000000 || $r16 = 0xxxxxxxxx

t=1330.0 ps, $r1 = 0xffffff8a || $r17 = 0xxxxxxxxx

t=1340.0 ps, $r2 = 0x0000008a || $r18 = 0xxxxxxxxx

t=1350.0 ps, $r3 = 0x000000c0 || $r19 = 0xxxxxxxxx

t=1360.0 ps, $r4 = 0x000000c4 || $r20 = 0xxxxxxxxx

t=1370.0 ps, $r5 = 0x000000c8 || $r21 = 0xxxxxxxxx

t=1380.0 ps, $r6 = 0x000000cc || $r22 = 0xxxxxxxxx

t=1390.0 ps, $r7 = 0x000000d0 || $r23 = 0xxxxxxxxx

t=1400.0 ps, $r8 = 0x000000d4 || $r24 = 0xxxxxxxxx

t=1410.0 ps, $r9 = 0xxxxxxxxx || $r25 = 0xxxxxxxxx

t=1420.0 ps, $r10 = 0xxxxxxxxx || $r26 = 0xxxxxxxxx

t=1430.0 ps, $r11 = 0xxxxxxxxx || $r27 = 0xxxxxxxxx

t=1440.0 ps, $r12 = 0xxxxxxxxx || $r28 = 0xxxxxxxxx

t=1450.0 ps, $r13 = 0xxxxxxxxx || $r29 = 0x000003fc

t=1460.0 ps, $r14 = 0x00000000 || $r30 = 0xxxxxxxxx

t=1470.0 ps, $r15 = 0x100100c0 || $r31 = 0x00000014

Memory Locations 0xC0 to 0xFF

t=1480.0 ps, M[000000c0] = 0x000000c0

t=1490.0 ps, M[000000c4] = 0x000000c4

t=1500.0 ps, M[000000c8] = 0x000000c8

t=1510.0 ps, M[000000cc] = 0x000000cc

t=1520.0 ps, M[000000d0] = 0x000000d0

t=1530.0 ps, M[000000d4] = 0x000000d4

t=1540.0 ps, M[000000d8] = 0xffffff8a

t=1550.0 ps, M[000000dc] = 0x0000008a

t=1560.0 ps, M[000000e0] = 0xxxxxxxxx

t=1570.0 ps, M[000000e4] = 0xxxxxxxxx

t=1580.0 ps, M[000000e8] = 0xxxxxxxxx

t=1590.0 ps, M[000000ec] = 0xxxxxxxxx

t=1600.0 ps, M[000000f0] = 0xxxxxxxxx

t=1610.0 ps, M[000000f4] = 0xxxxxxxxx

t=1620.0 ps, M[000000f8] = 0xxxxxxxxx

t=1630.0 ps, M[000000fc] = 0xxxxxxxxx

t=1640.0 ps, PC = 0x00000020, IR = 0x0000000d

**-** The BLEZ and BGTZ instructions are tested after the JAL instruction and are confirmed to   
 be working correctly according to the pass flags in memory locations 0xC0 to 0xD4. After   
 jumping back to the original PC before the JAL instruction, 0xD8 and 0xDC gets the values   
 0xFFFFFF8A and 0x0000008A, respectively, and further proves the functionality of the   
 instructions.

**Instruction Memory Module 13**

**Testing Instructions:** SETIE, INPUT, OUTPUT, SRA(Barrel Shift)

@0

00 00 00 1f // main: setie

3c 01 12 34 // lui $01, 0x1234

34 21 56 78 // ori $01, 0x5678 # LI R01, 0x12345678

3c 02 87 65 // lui $02, 0x8765

34 42 43 21 // ori $02, 0x4321 # LI R02, 0x87654321

3c 03 ab cd // lui $03, 0xABCD

34 63 ef 01 // ori $03, 0xEF01 # LI R03, 0xABCDEF01

3c 04 01 fe // lui $04, 0x01FE

34 84 dc ba // ori $04, 0xDCBA # LI R04, 0x01FEDCBA

3c 05 5a 5a // lui $05, 0x5A5A

34 a5 5a 5a // ori $05, 0x5A5A # LI R05, 0x5A5A5A5A

3c 06 ff ff // lui $06, 0xFFFF

34 c6 ff ff // ori $06, 0xFFFF # LI R06, 0xFFFFFFFF

3c 07 ff ff // lui $07, 0xFFFF

34 e7 ff 00 // ori $07, 0xFF00 # LI R07, 0xFFFFFF00

00 c7 40 20 // add $08, $06, $07

00 c8 48 20 // add $09, $06, $08

00 c9 50 20 // add $10, $06, $09

00 ca 58 20 // add $11, $06, $10

00 cb 60 20 // add $12, $06, $11

00 cc 68 20 // add $13, $06, $12

00 cd 70 20 // add $14, $06, $13

00 ce 78 20 // add $15, $06, $14

3c 07 10 01 // lui $07, 0x1001

34 e7 03 f0 // ori $07, 0x03F0 # LI R07, 0x100103F0

ac ef 00 00 // sw $15, 0($07) # ST [R07], R15

00 00 00 0d // break

@200

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// In this ISR, we will implement writing

// some patterns to the IO space, and then

// reading them back.

// Note: this "ISR" expects the "return addr”

// to have been saved in $ra (not the stack)

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

3c 10 10 01 // isr: lui $16, 0x1001 #load destination IO addr

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

BREAK INSTRUCTION FETCHED 5030.0 ps

R E G I S T E R ' S A F T E R B R E A K

t=5040.0 ps, $r0 = 0x00000000 || $r16 = 0x100100c0

t=5050.0 ps, $r1 = 0x12345678 || $r17 = 0xffffffff

t=5060.0 ps, $r2 = 0x87654321 || $r18 = 0x00000000

t=5070.0 ps, $r3 = 0xabcdef01 || $r19 = 0x8000ffff

t=5080.0 ps, $r4 = 0x01fedcba || $r20 = 0xe0003fff

t=5090.0 ps, $r5 = 0x5a5a5a5a || $r21 = 0xf8000fff

t=5100.0 ps, $r6 = 0xffffffff || $r22 = 0xfe0003ff

t=5110.0 ps, $r7 = 0x100103f0 || $r23 = 0xff8000ff

t=5120.0 ps, $r8 = 0xfffffeff || $r24 = 0xffe0003f

t=5130.0 ps, $r9 = 0xfffffefe || $r25 = 0xxxxxxxxx

t=5140.0 ps, $r10 = 0xfffffefd || $r26 = 0xxxxxxxxx

t=5150.0 ps, $r11 = 0xfffffefc || $r27 = 0xxxxxxxxx

t=5160.0 ps, $r12 = 0xfffffefb || $r28 = 0xxxxxxxxx

t=5170.0 ps, $r13 = 0xfffffefa || $r29 = 0x000003fc

t=5180.0 ps, $r14 = 0xfffffef9 || $r30 = 0xxxxxxxxx

t=5190.0 ps, $r15 = 0xfffffef8 || $r31 = 0x00000020

Memory Location 0x3F0

t=5200.0 ps, M[000003f0] = 0xfffffef8

IO Memory Locations 0xC0 to 0xFF

t=5210.0 ps, IO[000000c0] = 0x8000ffff

t=5220.0 ps, IO[000000c4] = 0xe0003fff

t=5230.0 ps, IO[000000c8] = 0xf8000fff

t=5240.0 ps, IO[000000cc] = 0xfe0003ff

t=5250.0 ps, IO[000000d0] = 0xff8000ff

t=5260.0 ps, IO[000000d4] = 0xffe0003f

t=5270.0 ps, IO[000000d8] = 0xfff8000f

t=5280.0 ps, IO[000000dc] = 0xfffe0003

t=5290.0 ps, IO[000000e0] = 0xffff8000

t=5300.0 ps, IO[000000e4] = 0xffffe000

t=5310.0 ps, IO[000000e8] = 0xfffff800

t=5320.0 ps, IO[000000ec] = 0xfffffe00

t=5330.0 ps, IO[000000f0] = 0xffffff80

t=5340.0 ps, IO[000000f4] = 0xffffffe0

t=5350.0 ps, IO[000000f8] = 0xfffffff8

t=5360.0 ps, IO[000000fc] = 0xfffffffe

t=5370.0 ps, PC = 0x0000006c, IR = 0x0000000d

**Instruction Memory Module 13 (page 2)**

**Testing Instructions:** SETIE, INPUT, OUTPUT, SRA(Barrel Shift)

76 11 00 00 // out\_IO: output $17, 0($16) # output [R16], R17

00 11 88 83 // sra $17, $17, 2 # shift twice

22 10 00 04 // addi $16, $16, 4 # inc memory pointer by 4

22 52 ff ff // addi $18, $18, -1 # dec the loop counter

16 40 ff fb // bne $18, $00, out\_IO # and jmp to top if   
 // not done

3c 10 10 01 // lui $16, 0x1001 #load source IO addr

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)

- The SETIE instruction is called as the very first instruction in the Instruction Memory.  
- After 300ns, an interrupt happens and the processor jumps to the Interrupt Service Routine.  
 In the ISR, the OUTPUT instruction is called 16 times and writes to IO memory locations  
 0xC0 to 0xFF while doing the SRA(2) instruction in every iteration of the loop.

- After the loop, the INPUT instruction writes to registers $r19 to $r24 from IO memory   
 locations 0xC0 to 0xD4

- The ISR then calls the JR instruction to jump back to the main routine and at the end, the   
 value of register $r15, 0xFFFFFEF8 is stored into memory location 0x3F0.

**Instruction Memory Module 14**

**Testing Instructions:** RETI

@0

00 00 00 1f // main: setie

3c 01 12 34 // lui $01, 0x1234

34 21 56 78 // ori $01, 0x5678 # LI R01, 0x12345678

3c 02 87 65 // lui $02, 0x8765

34 42 43 21 // ori $02, 0x4321 # LI R02, 0x87654321

3c 03 ab cd // lui $03, 0xABCD

34 63 ef 01 // ori $03, 0xEF01 # LI R03, 0xABCDEF01

3c 04 01 fe // lui $04, 0x01FE

34 84 dc ba // ori $04, 0xDCBA # LI R04, 0x01FEDCBA

3c 05 5a 5a // lui $05, 0x5A5A

34 a5 5a 5a // ori $05, 0x5A5A # LI R05, 0x5A5A5A5A

3c 06 ff ff // lui $06, 0xFFFF

34 c6 ff ff // ori $06, 0xFFFF # LI R06, 0xFFFFFFFF

3c 07 ff ff // lui $07, 0xFFFF

34 e7 ff 00 // ori $07, 0xFF00 # LI R07, 0xFFFFFF00

00 c7 40 20 // add $08, $06, $07

00 c8 48 20 // add $09, $06, $08

00 c9 50 20 // add $10, $06, $09

00 ca 58 20 // add $11, $06, $10

00 cb 60 20 // add $12, $06, $11

00 cc 68 20 // add $13, $06, $12

00 cd 70 20 // add $14, $06, $13

00 ce 78 20 // add $15, $06, $14

3c 07 10 01 // lui $07, 0x1001

34 e7 03 f0 // ori $07, 0x03F0 # LI R07, 0x100103F0

ac ef 00 00 // sw $15, 0($07) # ST [R07], R15

00 00 00 0d // break

@200   
 //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// In this ISR, we will implement writing

// some patterns to the IO space, and then

// reading them back.

// Note: this "ISR" expects the "return address"

// to have been saved on the stack (not $ra)

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

3c 10 10 01 // isr: lui $16, 0x1001 #load dest 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 #shift right by 2

22 10 00 04 // addi $16, $16, 4 # inc memory pointer by 4

22 52 ff ff // addi $18, $18, -1 # dec the loop counter

16 40 ff fb // bne $18, $00,out\_IO# jmp to top if not done

BREAK INSTRUCTION FETCHED 5100.0 ps

R E G I S T E R ' S A F T E R B R E A K

t=5110.0 ps, $r0 = 0x00000000 || $r16 = 0x100100c0

t=5120.0 ps, $r1 = 0x12345678 || $r17 = 0xffffffff

t=5130.0 ps, $r2 = 0x87654321 || $r18 = 0x00000000

t=5140.0 ps, $r3 = 0xabcdef01 || $r19 = 0x8000ffff

t=5150.0 ps, $r4 = 0x01fedcba || $r20 = 0xe0003fff

t=5160.0 ps, $r5 = 0x5a5a5a5a || $r21 = 0xf8000fff

t=5170.0 ps, $r6 = 0xffffffff || $r22 = 0xfe0003ff

t=5180.0 ps, $r7 = 0x100103f0 || $r23 = 0xff8000ff

t=5190.0 ps, $r8 = 0xfffffeff || $r24 = 0xffe0003f

t=5200.0 ps, $r9 = 0xfffffefe || $r25 = 0xxxxxxxxx

t=5210.0 ps, $r10 = 0xfffffefd || $r26 = 0xxxxxxxxx

t=5220.0 ps, $r11 = 0xfffffefc || $r27 = 0xxxxxxxxx

t=5230.0 ps, $r12 = 0xfffffefb || $r28 = 0xxxxxxxxx

t=5240.0 ps, $r13 = 0xfffffefa || $r29 = 0x000003fc

t=5250.0 ps, $r14 = 0xfffffef9 || $r30 = 0xxxxxxxxx

t=5260.0 ps, $r15 = 0xfffffef8 || $r31 = 0x00000000

Memory Location 0x3F0 to 0x404

t=5270.0 ps, M[000003f0] = 0xfffffef8 //SW $15, 0($07)

t=5280.0 ps, M[000003f4] = 0xxxxxxxxx

t=5290.0 ps, M[000003f8] = 0x00000000

t=5300.0 ps, M[000003fc] = 0x00000200 //ISR address

t=5310.0 ps, M[00000400] = 0x00000020 //PC

t=5320.0 ps, M[00000404] = 0x00000011 //Status Flags

IO Memory Locations 0xC0 to 0xFF

t=5330.0 ps, IO[000000c0] = 0x8000ffff

t=5340.0 ps, IO[000000c4] = 0xe0003fff

t=5350.0 ps, IO[000000c8] = 0xf8000fff

t=5360.0 ps, IO[000000cc] = 0xfe0003ff

t=5370.0 ps, IO[000000d0] = 0xff8000ff

t=5380.0 ps, IO[000000d4] = 0xffe0003f

t=5390.0 ps, IO[000000d8] = 0xfff8000f

t=5400.0 ps, IO[000000dc] = 0xfffe0003

t=5410.0 ps, IO[000000e0] = 0xffff8000

t=5420.0 ps, IO[000000e4] = 0xffffe000

t=5430.0 ps, IO[000000e8] = 0xfffff800

t=5440.0 ps, IO[000000ec] = 0xfffffe00

t=5450.0 ps, IO[000000f0] = 0xffffff80

t=5460.0 ps, IO[000000f4] = 0xffffffe0

t=5470.0 ps, IO[000000f8] = 0xfffffff8

t=5480.0 ps, IO[000000fc] = 0xfffffffe

t=5490.0 ps, PC = 0x0000006c, IR = 0x0000000d

**Instruction Memory Module 14 (page 2)**

**Testing Instructions:** RETI

3c 10 10 01 // lui $16, 0x1001 #load source IO address

36 10 00 c0 // ori $16, 0x00C0 # 0x100100C0 into r16

72 13 00 00 // input $19, 0($16) # and input from 6

72 14 00 04 // input $20, 4($16) # the IO locations,

72 15 00 08 // input $21, 8($16) # starting from 0xC0

72 16 00 0c // input $22, 12($16)

72 17 00 10 // input $23, 16($16)

72 18 00 14 // input $24, 20($16)

7B A0 00 00 // reti # return from interrupt (v2, using M[sp]  
 # as saved PC)

# Note opcode=0x1E and $rs=0x1D, i.e. $sp

- The SETIE instruction is called as the very first instruction in the Instruction Memory.  
- After 300ns, an interrupt happens and the processor jumps to the Interrupt Service Routine.  
 The INTR instruction from the Control Unit then writes the PC to memory location   
 0x400and current status flags to memory location 0x404 before jumping to the ISR

- In the ISR, the OUTPUT instruction is called 16 times and writes to IO memory locations  
 0xC0 to 0xFF while doing the SRA(2) instruction in every iteration of the loop.

- After the loop, the INPUT instruction writes to registers $r19 to $r24 from IO memory   
 locations 0xC0 to 0xD4

- The ISR then calls the RETI instruction to get the old PC from memory location 0x400 and   
 jump back to it (0x00000020) and resets the stack pointer to 0x3FC on $sp

**Instruction Memory Enhanced Instruction Module 1**

**Testing Instructions:** ROTLEFT

@0

3c 01 ff fF // main: lui $01, 0xFFFF

34 21 00 00 // ori $01, 0x0000 # LI R01, 0xFFFF0000

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 ac // top: rotleft $01, $01, 2 # rotate left 2 bits

ad e1 00 00 // sw $01, 0($15) # ST [R15], R01

21 ef 00 04 // addi $15, $15, 4 # inc memory pointer by 4

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 done

08 10 00 0c // j exit # jump around a halt instr

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

$r2 is decremented by 1, sixteen times until it reaches 0 and JUMPS to the exit.

$r15 from 0x100100C0 incremented by 4, sixteen times and resulted into 0x10010100  
- Register $r2 having the value 0x00000000 proves that the ADDI function works correctly.

- Since register $r14 has 0x5A5A3C3C, it shows that all the instructions are working as   
 intended

- Values inside the Memory Locations 0xC0 to 0xFF shows the original value of $r1 of   
 0xFFFF0000 rotating 2 bits to the left using the instruction **ROTLEFT** each loop and after   
 16 loops, goes back to its original value.

BREAK INSTRUCTION FETCHED 3650.0 ps

R E G I S T E R ' S A F T E R B R E A K

t=3660.0 ps, $r0 = 0x00000000 || $r16 = 0xxxxxxxxx

t=3670.0 ps, $r1 = 0xffff0000 || $r17 = 0xxxxxxxxx

t=3680.0 ps, $r2 = 0x00000000 || $r18 = 0xxxxxxxxx

t=3690.0 ps, $r3 = 0xxxxxxxxx || $r19 = 0xxxxxxxxx

t=3700.0 ps, $r4 = 0xxxxxxxxx || $r20 = 0xxxxxxxxx

t=3710.0 ps, $r5 = 0xxxxxxxxx || $r21 = 0xxxxxxxxx

t=3720.0 ps, $r6 = 0xxxxxxxxx || $r22 = 0xxxxxxxxx

t=3730.0 ps, $r7 = 0xxxxxxxxx || $r23 = 0xxxxxxxxx

t=3740.0 ps, $r8 = 0xxxxxxxxx || $r24 = 0xxxxxxxxx

t=3750.0 ps, $r9 = 0xxxxxxxxx || $r25 = 0xxxxxxxxx

t=3760.0 ps, $r10 = 0xxxxxxxxx || $r26 = 0xxxxxxxxx

t=3770.0 ps, $r11 = 0xxxxxxxxx || $r27 = 0xxxxxxxxx

t=3780.0 ps, $r12 = 0xxxxxxxxx || $r28 = 0xxxxxxxxx

t=3790.0 ps, $r13 = 0xxxxxxxxx || $r29 = 0x000003fc

t=3800.0 ps, $r14 = 0x5a5a3c3c || $r30 = 0xxxxxxxxx

t=3810.0 ps, $r15 = 0x10010100 || $r31 = 0x00000000

Memory Locations 0xC0 to 0xFF

t=3820.0 ps, M[000000c0] = 0xfffc0003

t=3830.0 ps, M[000000c4] = 0xfff0000f

t=3840.0 ps, M[000000c8] = 0xffc0003f

t=3850.0 ps, M[000000cc] = 0xff0000ff

t=3860.0 ps, M[000000d0] = 0xfc0003ff

t=3870.0 ps, M[000000d4] = 0xf0000fff

t=3880.0 ps, M[000000d8] = 0xc0003fff

t=3890.0 ps, M[000000dc] = 0x0000ffff

t=3900.0 ps, M[000000e0] = 0x0003fffc

t=3910.0 ps, M[000000e4] = 0x000ffff0

t=3920.0 ps, M[000000e8] = 0x003fffc0

t=3930.0 ps, M[000000ec] = 0x00ffff00

t=3940.0 ps, M[000000f0] = 0x03fffc00

t=3950.0 ps, M[000000f4] = 0x0ffff000

t=3960.0 ps, M[000000f8] = 0x3fffc000

t=3970.0 ps, M[000000fc] = 0xffff0000

t=3980.0 ps, PC = 0x0040003c, IR = 0x0000000d

**Instruction Memory Enhanced Instruction Module 2**

**Testing Instructions:** ROTRIGHT

@0

3c 01 ff fF // main: lui $01, 0xFFFF

34 21 00 00 // ori $01, 0x0000 # LI R01, 0xFFFF0000

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 ad // top: rotright $01, $01, 2 # rotate right 2 bits

ad e1 00 00 // sw $01, 0($15) # ST [R15], R01

21 ef 00 04 // addi $15, $15, 4 # inc memory pointer by 4

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 done

08 10 00 0c // j exit # jump around a halt instr

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

$r2 is decremented by 1, sixteen times until it reaches 0 and JUMPS to the exit.

$r15 from 0x100100C0 incremented by 4, sixteen times and resulted into 0x10010100  
- Register $r2 having the value 0x00000000 proves that the ADDI function works correctly.

- Since register $r14 has 0x5A5A3C3C, it shows that all the instructions are working as   
 intended

- Values inside the Memory Locations 0xC0 to 0xFF shows the original value of $r1 of   
 0xFFFF0000 rotating 2 bits to the right using the instruction **ROTRIGHT** each loop and   
 after 16 loops, goes back to its original value.

BREAK INSTRUCTION FETCHED 3650.0 ps

R E G I S T E R ' S A F T E R B R E A K

t=3660.0 ps, $r0 = 0x00000000 || $r16 = 0xxxxxxxxx

t=3670.0 ps, $r1 = 0xffff0000 || $r17 = 0xxxxxxxxx

t=3680.0 ps, $r2 = 0x00000000 || $r18 = 0xxxxxxxxx

t=3690.0 ps, $r3 = 0xxxxxxxxx || $r19 = 0xxxxxxxxx

t=3700.0 ps, $r4 = 0xxxxxxxxx || $r20 = 0xxxxxxxxx

t=3710.0 ps, $r5 = 0xxxxxxxxx || $r21 = 0xxxxxxxxx

t=3720.0 ps, $r6 = 0xxxxxxxxx || $r22 = 0xxxxxxxxx

t=3730.0 ps, $r7 = 0xxxxxxxxx || $r23 = 0xxxxxxxxx

t=3740.0 ps, $r8 = 0xxxxxxxxx || $r24 = 0xxxxxxxxx

t=3750.0 ps, $r9 = 0xxxxxxxxx || $r25 = 0xxxxxxxxx

t=3760.0 ps, $r10 = 0xxxxxxxxx || $r26 = 0xxxxxxxxx

t=3770.0 ps, $r11 = 0xxxxxxxxx || $r27 = 0xxxxxxxxx

t=3780.0 ps, $r12 = 0xxxxxxxxx || $r28 = 0xxxxxxxxx

t=3790.0 ps, $r13 = 0xxxxxxxxx || $r29 = 0x000003fc

t=3800.0 ps, $r14 = 0x5a5a3c3c || $r30 = 0xxxxxxxxx

t=3810.0 ps, $r15 = 0x10010100 || $r31 = 0x00000000

Memory Locations 0xC0 to 0xFF

t=3820.0 ps, M[000000c0] = 0x3fffc000

t=3830.0 ps, M[000000c4] = 0x0ffff000

t=3840.0 ps, M[000000c8] = 0x03fffc00

t=3850.0 ps, M[000000cc] = 0x00ffff00

t=3860.0 ps, M[000000d0] = 0x003fffc0

t=3870.0 ps, M[000000d4] = 0x000ffff0

t=3880.0 ps, M[000000d8] = 0x0003fffc

t=3890.0 ps, M[000000dc] = 0x0000ffff

t=3900.0 ps, M[000000e0] = 0xc0003fff

t=3910.0 ps, M[000000e4] = 0xf0000fff

t=3920.0 ps, M[000000e8] = 0xfc0003ff

t=3930.0 ps, M[000000ec] = 0xff0000ff

t=3940.0 ps, M[000000f0] = 0xffc0003f

t=3950.0 ps, M[000000f4] = 0xfff0000f

t=3960.0 ps, M[000000f8] = 0xfffc0003

t=3970.0 ps, M[000000fc] = 0xffff0000

t=3980.0 ps, PC = 0x0040003c, IR = 0x0000000d

