



UNIVERSITY OF
LIVERPOOL

Digital Systems Design with Verilog

MIPS Processor Design Assignment 3

Assignment 3 – Two/Three Parts

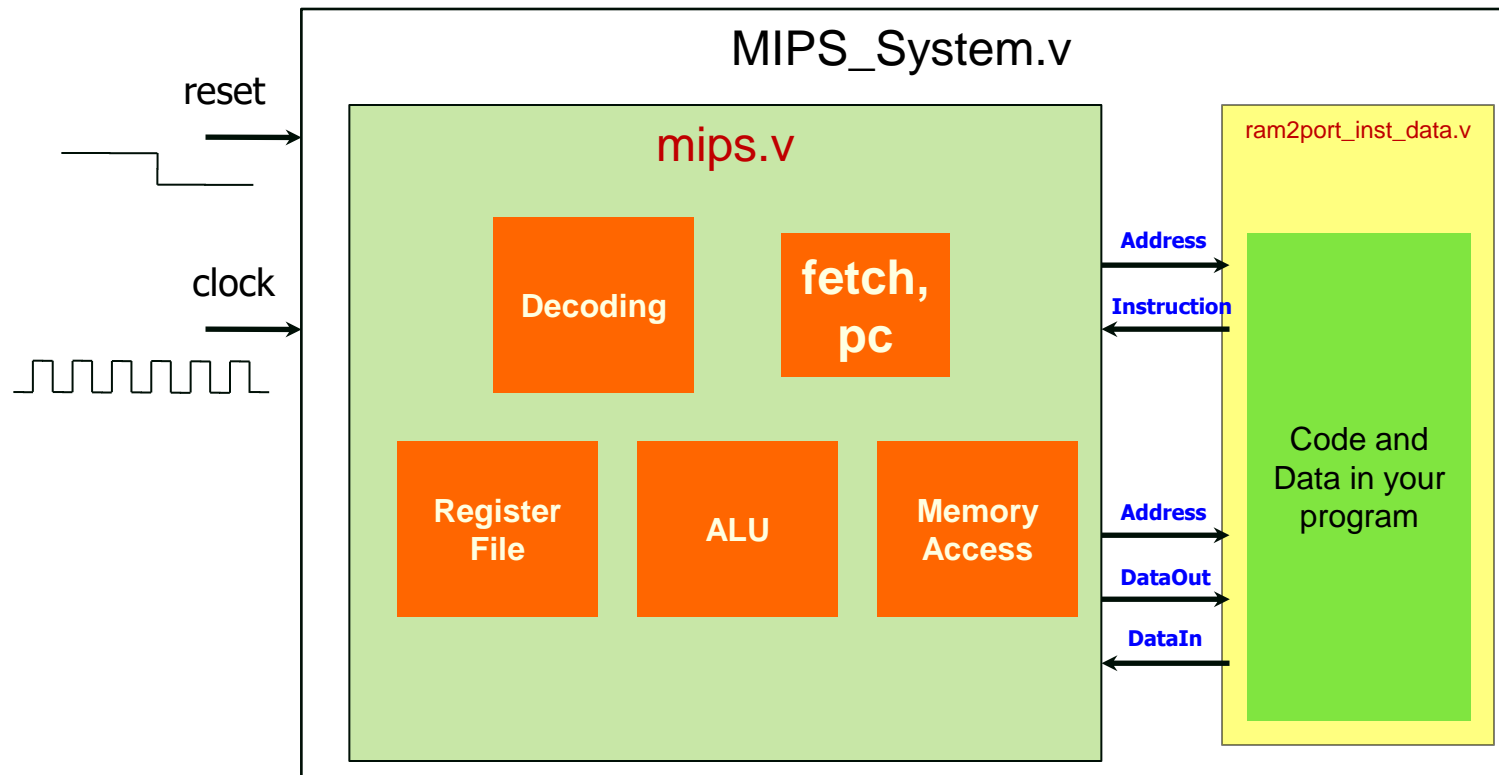
■ Assignment 3 Part A

1. Modify the MIPS assembly language program so that the program displays the lowest 8 digits of your ID on the DE2 board 7 segment display.
2. Show that your program functions correctly by taking a screen shot(s) of the ModelSim simulation or SignalTap Logic analyser.
3. In your report you should include your assembly language code and a screen dump of the ModelSim simulation or SignalTap Logic analyser. Also include a photograph of the 7 segment displays showing your ID if you used a DE2 Board.

Assignment 3 – Part B

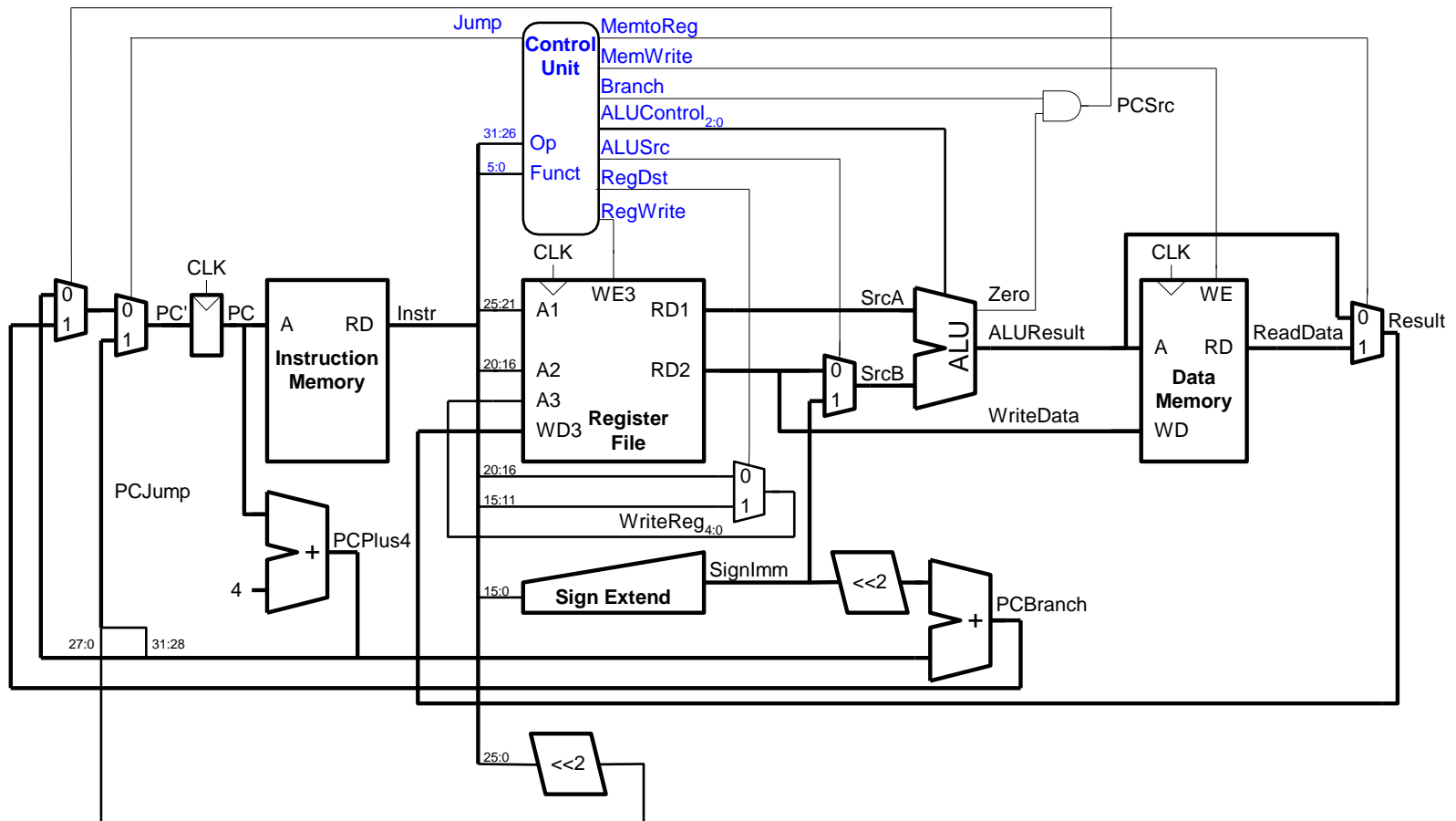
- The MIPS design presented in MIPS_System only implements a limited number of the MIPS instructions. For the R-Type instructions ADD, ADDU, SUB, SUBU, AND, OR and SLT are implemented. Your task is to modify the MIPS design so that it implements the additional instructions shown in Table 1.
- Once you have modified your design you need to write a program to demonstrate that your hardware correctly implements the instructions. Your results should include print outs of the SignalTap logic analyser showing your program operating. Annotate the print out to explain what is happening.
- (Instructions may include nor, xor, andi, xori, lb, lbu, lh) ³

Overview of Our Design

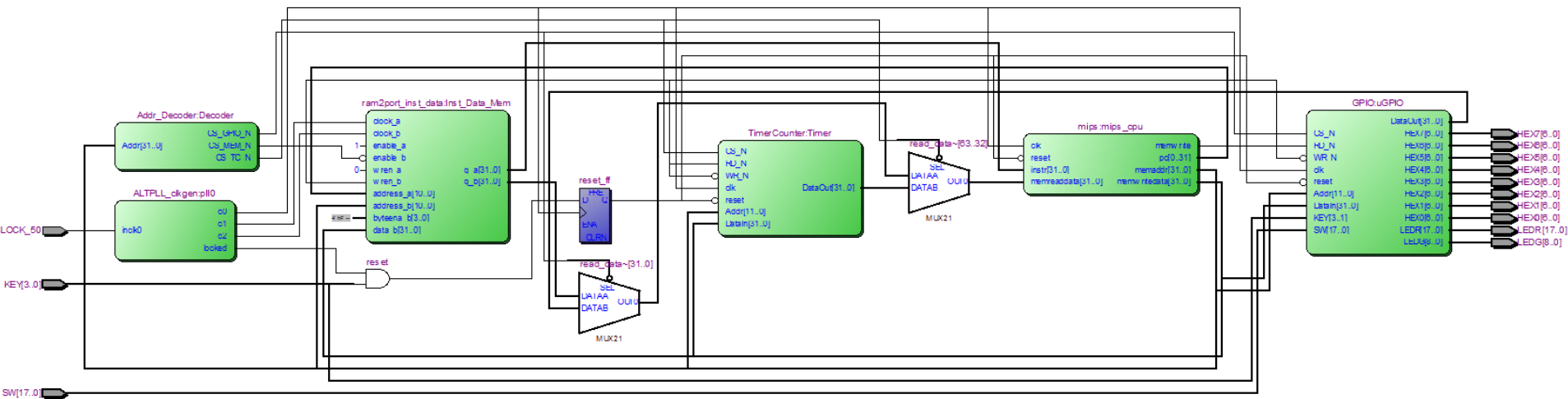


Design single cycle

– From previous lectures



RTL View of MIPS Design



Features of the MIPS Design

- Uses a PLL to generate 3 clock signals
- Includes Dual Port memory (for data and for instructions)
- Includes a Timer – not essential for you
- Includes GPIO to access the LEDs and Switches on the DE2 Board
- Has an address decoder to select between
 - Memory
 - GPIO
 - Timer

Why 3 clocks?

- It's supposed to be a single cycle design why have we three clocks?

Why three clocks?

- Remember the order of events after the main clock edge in this single cycle processor.
 - Latch the new PC into the PC latches
 - Read the Instruction from the Instruction memory
 - For a lw or sw instruction there is a data memory read or a data memory write once the address has been calculated
- As the Dual port memory accesses are synchronised with “clock edges” we need three clock “edges” within the one clock period.
 - Latch PC
 - Read Instruction
 - Read/Write data value
- We can use one of the Phase Locked Loops (PLLs) within the FPGA to generate the edges or split the clock into a number of cycles with a state machine.

Dual Port RAM module

MegaWizard Plug-In Manager [page 1 of 10]

RAM: 2-PORT

About Documentation

1 Parameter Settings 2 EDA 3 Summary

General Widths/Blk Type Clks/Rd, Byte En Regs/Clkens/Adrs Mem Init

Currently selected device family: Cyclone II
☒ Match project/default

How will you be using the dual port RAM?

☐ With one read port and one write port
☒ With two read/write ports

How do you want to specify the memory size?

☒ As a number of words
☐ As a number of bits

Block Type: AUTO

Resource Usage
16 M4K

ram2port_inst_data

data_a[31..0]
address_a[10..0]
wren_a
data_b[31..0]
address_b[10..0]
wren_b
byteena_b[3..0]
clock_a
enable_a
clock_b
enable_b

2048 Word(s) RAM

q_a[31..0]
q_b[31..0]

Cancel < Back Next > Finish

Can we remove the Registers on the inputs?

MegaWizard Plug-In Manager [page 5 of 10]

RAM: 2-PORT

About Documentation

1 Parameter Settings 2 EDA 3 Summary

General > Widths/Blk Type > Clks/Rd, Byte En > **Regs/Clkens/Aclr** > Mem Init >

ram2port_inst_data

data_a[31..0]
address_a[10..0]
wren_a
data_b[31..0]
address_b[10..0]
wren_b
byteena_b[3..0]
clock_a
enable_a
clock_b
enable_b

2048 Word(s) RAM

q_a[31..0]
q_b[31..0]

Block Type: AUTO

Resource Usage
16 M4K

Which ports should be registered?

- ☒ Write input ports
'data_a', 'waddress_a', and 'wren_a'
- ☐ Read input ports
'rdaddress' and 'rden'
- ☐ Read output port(s)
'q_a' and 'q_b'

☒ Create one clock enable signal for each clock signal

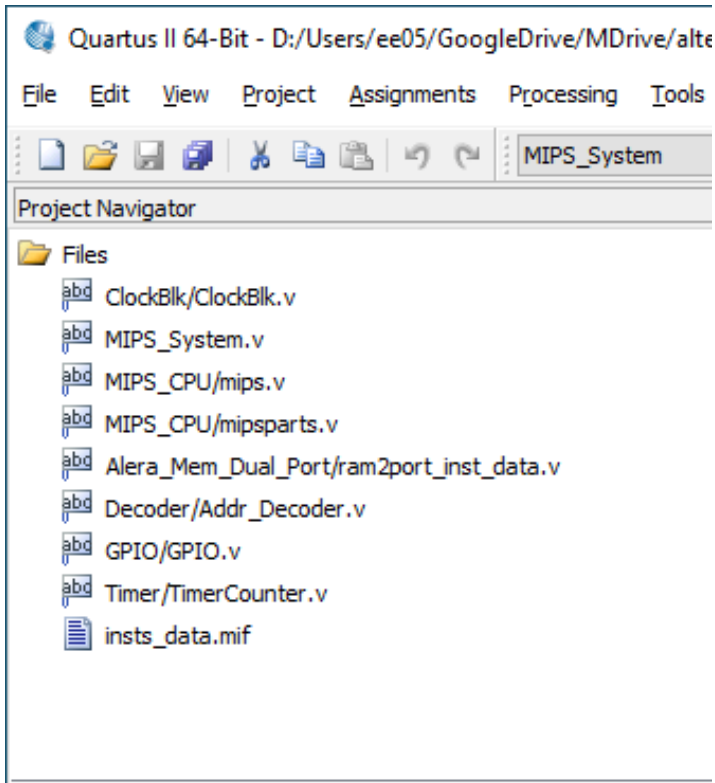
☐ Create an 'aclr' asynchronous clear for the registered ports

More Options... More Options... More Options...

Cancel < Back Next > Finish

Not on the cyclone II
The option is greyed out.

The Design Files



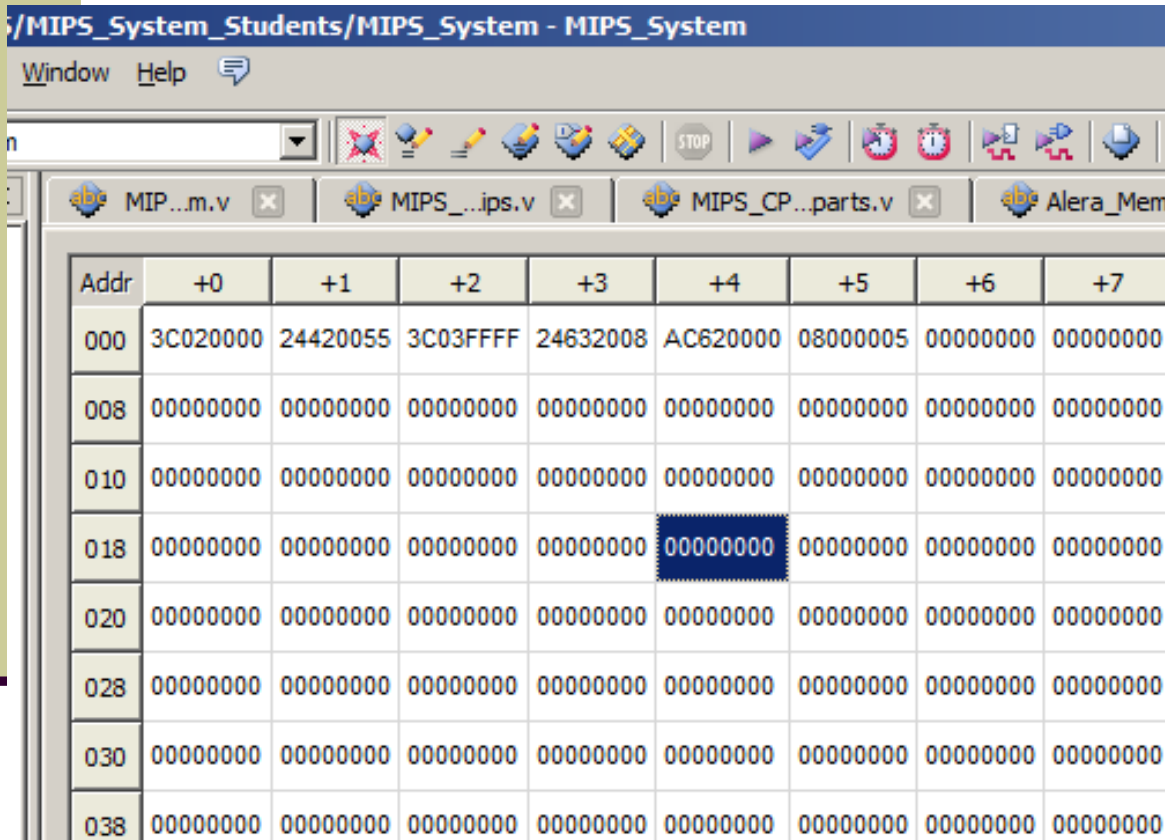
- MIPS_System.v is the top-level file that combines all the other modules.
- The Memory is implemented using an Altera Dual port ram megafunction.
 - The RAM is initialised using the “insts_data.mif” file. You need to edit this file with the hex code generated by the MIPS assembler.
- The ClockBlk generates the shifted clocks required by the Dual Port Memory.

The Memory Map

```
1 module Addr_Decoder (input [31:0] Addr,
2                       output reg CS_MEM_N,
3                       output reg CS_TC_N,
4                       output reg CS_UART_N,
5                       output reg CS_GPIO_N);
6
7 //=====
8 // Address      Peripheral      Peripheral Name      Size
9 // OxFFFF_FFFF -----
10 //
11 //              Reserved
12 //
13 // OxFFFF_3000 -----
14 //              GPIO          General Purpose IO      4KB
15 // OxFFFF_2000 -----
16 //              UART          Universal
17 //              Asynchronous   Receive/ Transmitter    4KB
18 //
19 // OxFFFF_1000 -----
20 //              TC            Timer Counter            4KB
21 // OxFFFF_0000 -----
22 //
23 //              Reserved
24 //
25 // Ox0000_2000 -----
26 //              mem           Instruction & Data Memory  8KB
27 // Ox0000_0000 -----
28 //=====
29
30 always @(*)
31 begin
32     if (Addr[31:13] == 19'h0000) // Instruction & Data Memory
33     begin
34         CS_MEM_N  <=0;
35         CS_TC_N   <=1;
36         CS_UART_N <=1;
37         CS_GPIO_N <=1;
38     end
39
40     else if (Addr[31:12] == 20'hFFFF0) // Timer
41     begin
42         CS_MEM_N  <=1;
43         CS_TC_N   <=0;
44         CS_UART_N <=1;
45         CS_GPIO_N <=1;
46     end
47
48     else if (Addr[31:12] == 20'hFFFF1) // UART
```

The Address Decoder uses the upper bits of the address bus to decide which device should be selected. Its normal for devices and control signals to be “active-low” i.e. selected when the control signal is low.

Given Insts_data.mif



Addr	+0	+1	+2	+3	+4	+5	+6	+7
000	3C020000	24420055	3C03FFFF	24632008	AC620000	08000005	00000000	00000000
008	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
010	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
018	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
020	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
028	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
030	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
038	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000

0x3C020000
0x24420055
0x3C03FFFF
0x24632008
0xAC620000
0x08000005

Manual Decoding - 0x3C020000

- 3 Instruction formats: all **32 bits** wide

opcode	rs	rt	rd	sa	funct	R format
opcode	rs	rt	immediate			I format
opcode	jump target					J format

What is the opcode (upper 6 bits)?

0x3C020000 = 0b0011_1100_0000_0010_0000_0000_0000_0000

Opcode = 0011_11

What format instruction?

Manual Decoding - 0x3C020000

opcode	rs	rt	rd	sa	funct	R format
opcode	rs	rt	immediate			I format
opcode	jump target					J format

0x3C020000 = 0b0011_1100_0000_0010_0000_0000_0000_0000

Opcode = 0011_11

An I type instruction: lui rt, immediate

Load upper immediate, set the upper 16 bits in register rt to the immediate value:

What register is rt?

Bits 20->16 = 0_0010 rt = register 2

What is the immediate value?

0x0000 (last 16 bits of the instruction)

Instruction is: lui \$2, 0x0000

Manual Decoding - 0x24420055

opcode	rs	rt	rd	sa	funct	R format
opcode	rs	rt	immediate			I format
opcode	jump target					J format

0x24420055 = 0b0010_0100_0100_0010_0000_0000_0101_0101

Opcode = 0010_01

An I type instruction : addiu rt, rs, immediate

Add immediate unsigned, add the 16 bit immediate value to rs and place the result in rt value:

What register is rs?

Bits 25->21 = 00_010 rs = register 2

What register is rt?

Bits 20->16 = 0_0010 rt = register 2

What is the immediate value?

0x0055 Bits 15->0 of the instruction

Instruction is: addiu \$2, \$2, 0x0055

Manual Decoding - 0x3C03FFFF

opcode	rs	rt	rd	sa	funct	R format
opcode	rs	rt	immediate			I format
opcode	jump target					J format

0x3C03FFFF = 0b0011_1100_0000_0011_1111_1111_1111_1111

Opcode = 0011_11 (upper 6 bits)

An I type instruction “lui rt, immediate”

Load upper immediate, set the upper 16 bits in register rt to the immediate value stored in the lower 16 bits:

What register is rt?

Bits 20->16 = 0_0011 rt = register 3

What is the immediate value?

0xFFFF (low 16 bits of the instruction)

Instruction is: lui \$3, 0xFFFF

Manual Decoding - 0x24632008

opcode	rs	rt	rd	sa	funct	R format
opcode	rs	rt	immediate			I format
opcode	jump target					J format

0x24632008 = 0b0010_0100_0110_0011_0010_0000_0000_1000

Opcode = 0010_01

An I type instruction : addiu rt, rs, immediate

Add immediate unsigned, add the 16 bit immediate value to rs and place the result in rt value:

What register is rs?

Bits 25->21 = 00_011 rs = register 3

What register is rt?

Bits 20->16 = 0_0011 rt = register 3

What is the immediate value?

0x2008, Low 16 bits of the instruction

Instruction is: addiu \$3, \$3, 0x2008

Manual Decoding - 0xAC620000

opcode	rs	rt	rd	sa	funct	R format
opcode	rs	rt	immediate			I format
opcode	jump target					J format

0xAC620000 = 0b1010_1100_0110_0010_0000_0000_0000_0000

Opcode = 1010_11

An I type instruction : sw rt, immediate(rs)

Store register rt at the location specified by rs offset by the immediate value:

What register is rs?

Bits 25->21 = 00_011 rs = register 3

What register is rt?

Bits 20->16 = 0_0010 rt = register 2

What is the immediate value?

0x0000 (Low 16 bits of the instruction)

Instruction is: sw \$2, 0x0000(\$3)

Manual Decoding – 0x08000005

opcode	rs	rt	rd	sa	funct	R format
opcode	rs	rt	immediate			I format
opcode	jump target					J format

0x08000005 = 0b0000_1000_0000_0000_0000_0000_0000_0101

Opcode = 0000_10

A J type instruction : j coded address of label

Jump to address

What address is label?

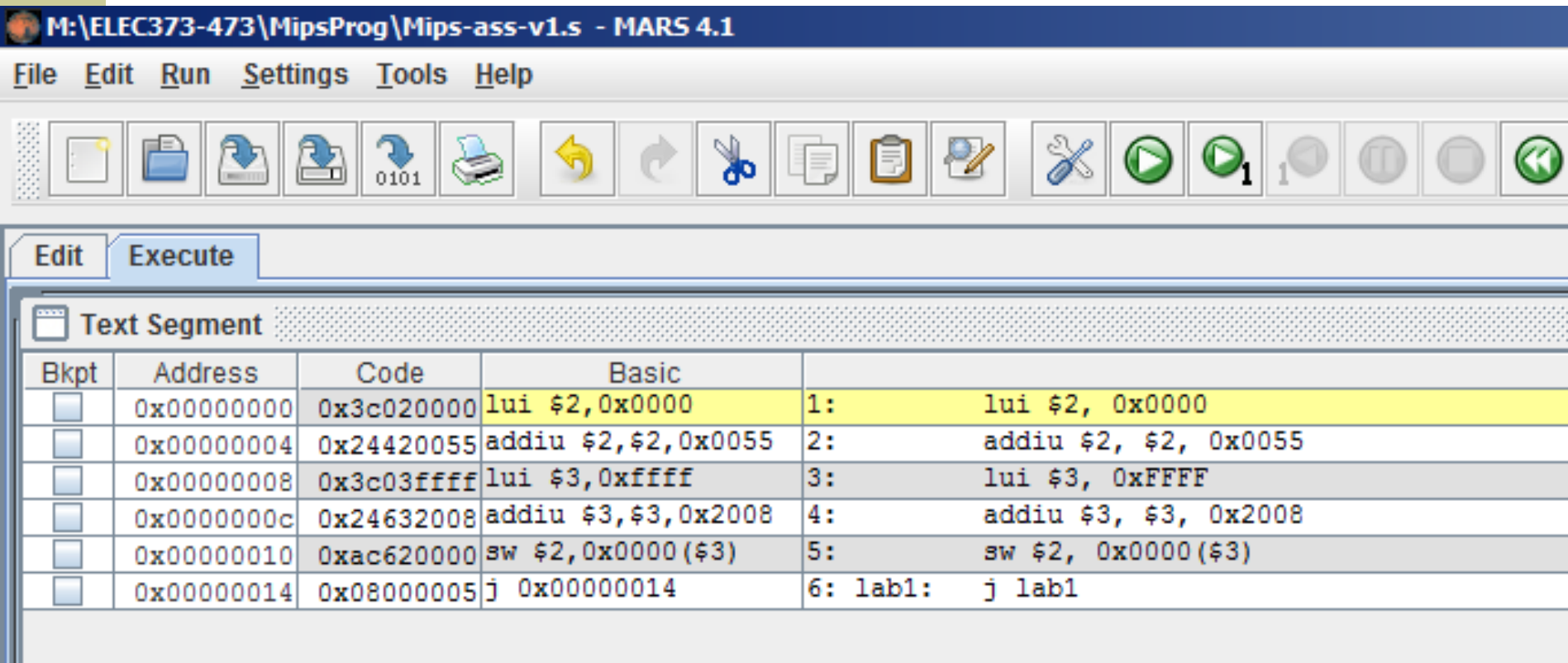
label = 0x00000005

Instruction is: j 0x00000014 (this is a byte address, $4 * 5 = 20d = 0x14$)

MARS Assembler

- If the Mars java file does not run when you click on it you can start it with the command line
- `Java -jar mars.jar` (if you saved it as mars.jar)

Assembled with MARS



To get the assembly code starting at location 0x00000000, select
“Settings->Memory Configuration->Compact, Text at Address 0”

What is stored where?

What does \$2 contain?

\$2 = 0x00000055

What does \$3 contain?

\$3 = 0xFFFF_2008

```
wire sw14_pressed;
wire sw15_pressed;
wire sw16_pressed;
wire sw17_pressed;
```

```
// Register
```

```
//
```

```
// FFFF_202C    HEX7_R
```

```
// FFFF_2028    HEX6_R
```

```
// FFFF_2024    HEX5_R
```

```
// FFFF_2020    HEX4_R
```

```
// FFFF_201C    HEX3_R
```

```
// FFFF_2018    HEX2_R
```

```
// FFFF_2014    HEX1_R
```

```
// FFFF_2010    HEX0_R
```

```
// FFFF_200C    LEDG_R
```

```
// FFFF_2008    LEDR_R
```

```
// FFFF_2004    SW_StatusR
```

```
// FFFF_2000    KEY_StatusR
```

```
//
```

```
// LEDG register (32bit)
```

```
// ZZZZ_ZZZ|LEDG8|_|LEDG7|LEDG6|LEDG5|LEDG4|_
```

```
//          |LEDG3|LEDG2|LEDG1|LEDG0|
```

```
//
```

```
// LEDR register(32 bit)
```

```
// ZZZZ_ZZZZ_ZZZZ_ZZ|LEDR17|LEDR16|_
```

```
//          |LEDR15|LEDR14|LEDR13|LEDR12|_
```

```
//          |LEDR11|LEDR10|LEDR9|LEDR8|_
```

```
//          |LEDR7|LEDR6|LEDR5|LEDR4|_
```

```
//          |LEDR3|LEDR2|LEDR1|LEDR0|
```

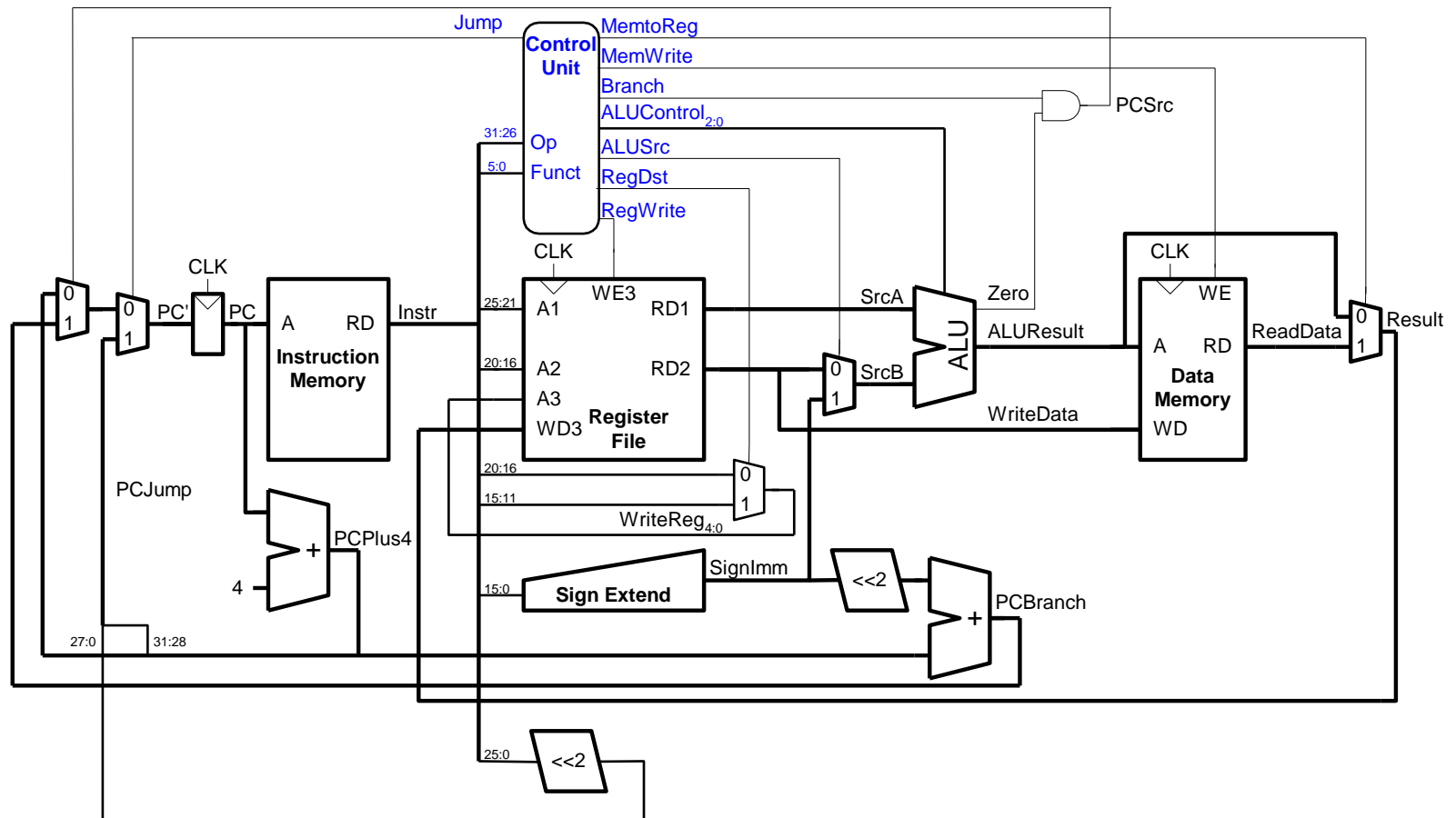
```
//
```

```
// SW Status register(32 bit)
```

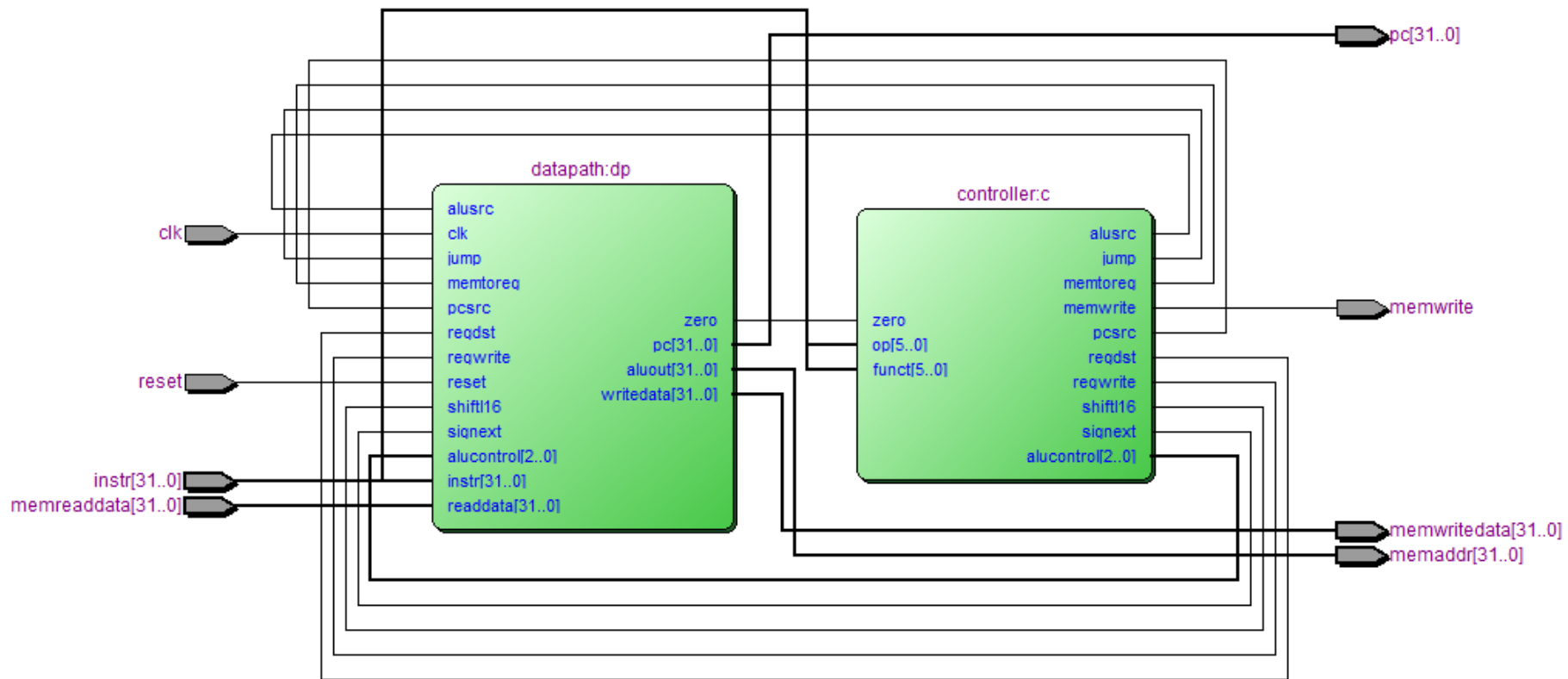
```
// ZZZZ_ZZZZ_ZZZZ_ZZ|SW17|SW16|
```

In the gpio.v file

What signals should be shown with ModelSim?



What signals should be shown with ModelSim?

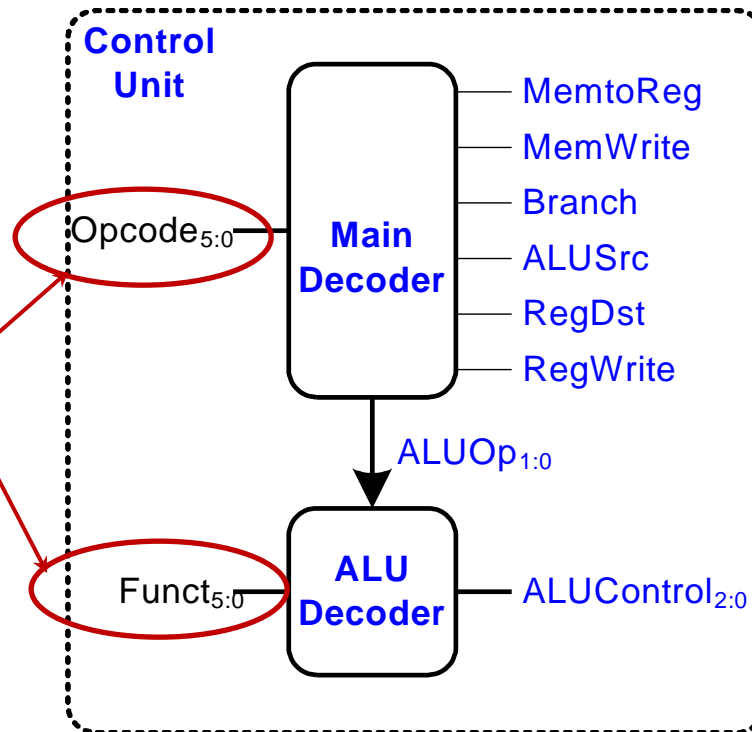


Part B – What to change

- Part B requires you to encode some new instructions.
- What instruction type is your “instruction 1”?

Control Unit

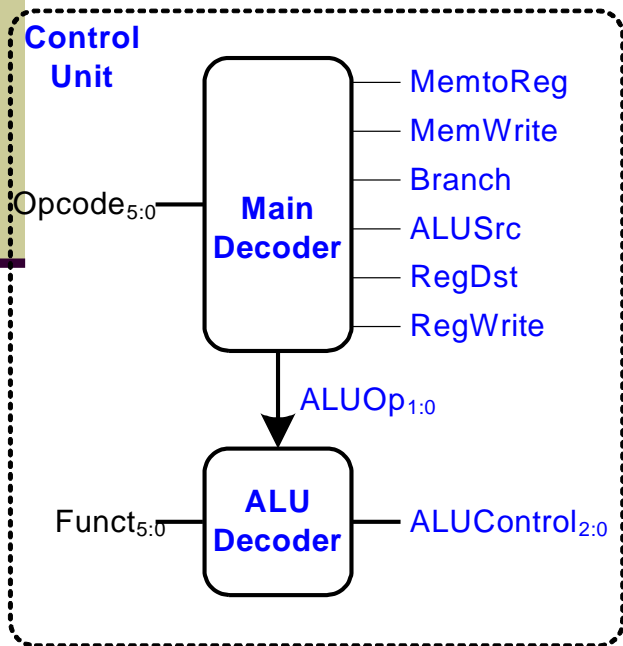
Opcode and **funct** fields come from the fetched instruction



Control Unit - ALU Control

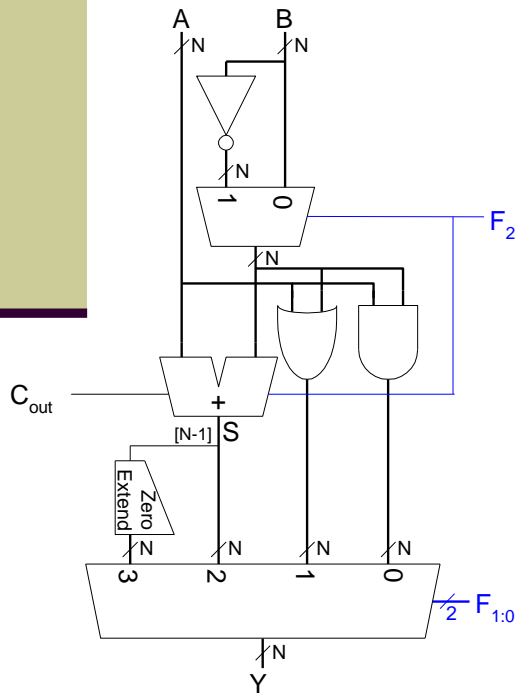
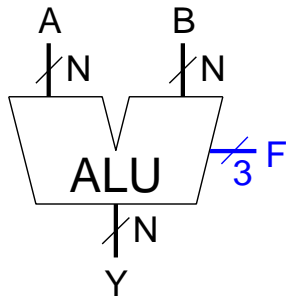
- Implementation is completely dependent on hardware designers
- But, the designers should make sure the implementation is reasonable enough
- Memory access instructions (`lw`, `sw`) need to use ALU to calculate memory target address (**addition**)
- Branch instructions (`beq`, `bne`) need to use ALU for the equality check (**subtraction**)

ALUOp _{1:0}	Meaning
00	Add
01	Subtract
10	Look at Funct
11	Not Used



ALUOp _{1:0}	Funct	ALUControl _{2:0}
00	X	010 (add)
X1	X	110 (subtract)
1X	100000 (add)	010 (add)
1X	100010 (sub)	110 (subtract)
1X	100100 (and)	000 (and)
1X	100101 (or)	001 (or)
1X	101010 (slt)	111 (slt)

Verilog Code – ALU



```

module alu(input      [31:0] a, b,
           input      [2:0] alucont,
           output reg [31:0] result,
           output      zero);

  wire [31:0] b2, sum, slt;

  assign b2 = alucont[2] ? ~b:b;
  // addition (sub)
  assign sum = a + b2 + alucont[2];
  assign slt = sum[31]; // SLT

  always@(*)
  begin
    case(alucont[1:0])
      2'b00: result <= a & b2; // A & B
      2'b01: result <= a | b2; // A | B
      2'b10: result <= sum;    // A + B, A - B
      2'b11: result <= slt;    // SLT
    endcase
  end

  // for branch
  assign zero = (result == 32'b0);

endmodule

```

F _{2:0}	Function
000	A & B
001	A B
010	A + B
011	not used
100	A & ~B
101	A ~B
110	A - B
111	SLT

Where are “R-Type” Instructions decoded?

```
assign {signext, shiftl16, regwrite, regdst,
       alusrc, branch, memwrite,
       memtoreg, jump, aluop} = controls;

always @(*)
case (op)
6'b000000: controls <= 11'b001100000011; // Rtype
6'b100011: controls <= 11'b10101001000; // LW
6'b101011: controls <= 11'b10001010000; // SW
6'b000100: controls <= 11'b10000100001; // BEQ
6'b001000,
6'b001001: controls <= 11'b10101000000; // ADDI, ADDIU: only difference is exception
6'b001101: controls <= 11'b00101000010; // ORI
6'b001111: controls <= 11'b01101000000; // LUI
6'b000010: controls <= 11'b00000000100; // J
default: controls <= 11'bxxxxxxxxxxx; // ???
endcase

endmodule

module aludec(input [5:0] funct,
             input [1:0] aluop,
             output reg [2:0] alucontrol);

always @(*)
case (aluop)
2'b00: alucontrol <= 3'b010; // add
2'b01: alucontrol <= 3'b110; // sub
2'b10: alucontrol <= 3'b001; // or
default: case (funct) // RTYPE
6'b100000,
6'b100001: alucontrol <= 3'b010; // ADD, ADDU: only difference is exception
6'b100010,
6'b100011: alucontrol <= 3'b110; // SUB, SUBU: only difference is exception
6'b100100: alucontrol <= 3'b000; // AND
6'b100101: alucontrol <= 3'b001; // OR
6'b101010: alucontrol <= 3'b111; // SLT
default: alucontrol <= 3'bxxx; // ???
endcase
endcase
endmodule
```

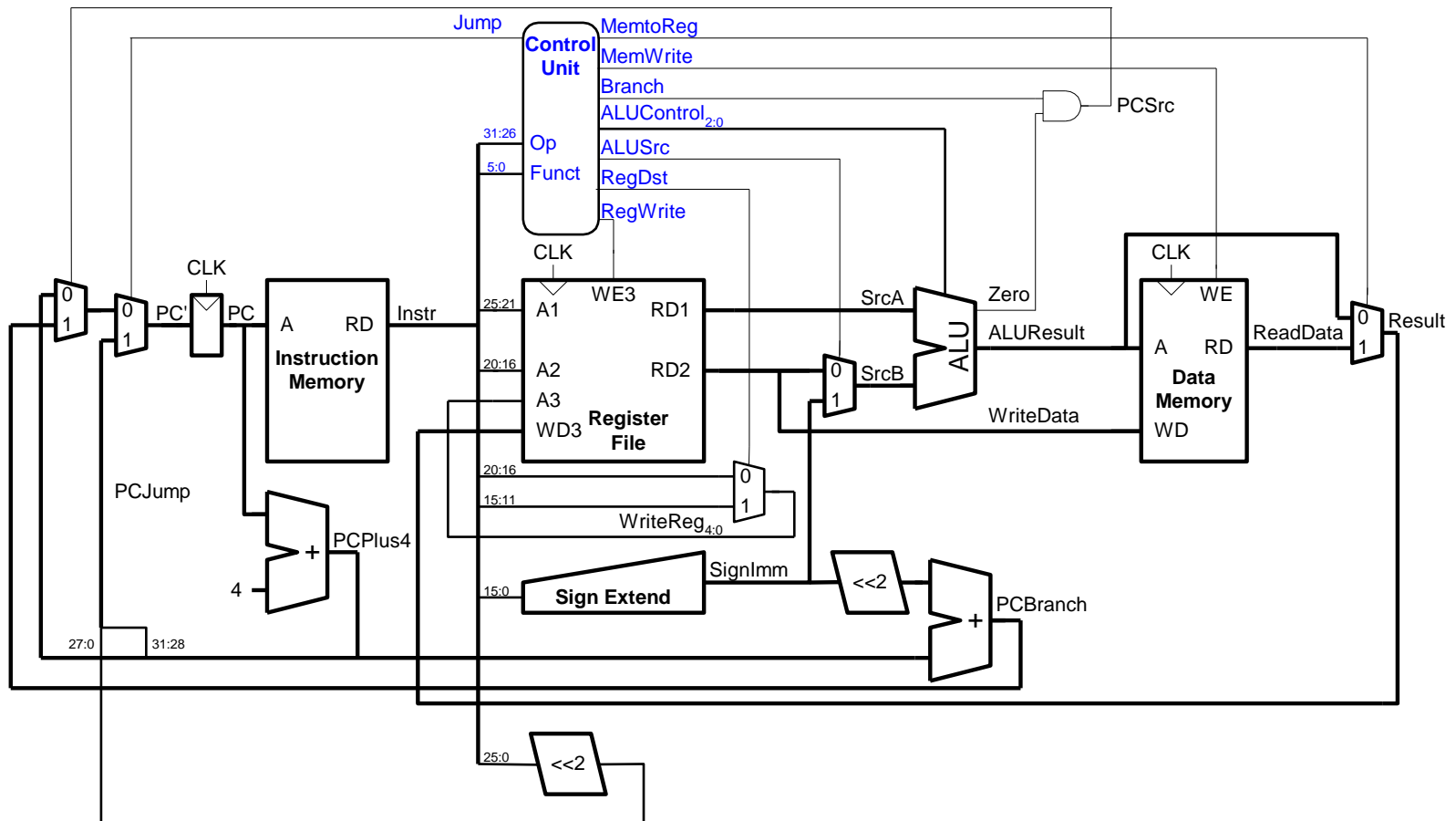
What do lb, lbu, lh, lhu do?

- lb = load byte
 - Load a single 8 bit byte from *any* memory location (not just word aligned) and place it in the lower 8 bits of the specified register. Upper 24 bits should be sign extension of bit 7
- lbu = load byte unsigned
 - Load a single 8 bit byte from *any* memory location (not just word aligned) and place it in the lower 8 bits of the specified register. Upper 24 bits should be set to zero

What do lb, lbu, lh, lhu do?

- lh = load half word
 - Load a 16 bit value **from any half word** memory location (not just word aligned) and place it in the lower 16 bits of the specified register. Upper 16 bits should be sign extension of bit 15
- lhu = load half word unsigned
 - Load a 16 bit byte value **from any half word** memory location (not just word aligned) and place it in the lower 16 bits of the specified register. Upper 16 bits should be set to zero

What extra hardware is needed for lb etc.?



Currently which address lines go to the Memory?

Quartus II 64-Bit - D:/Users/ee05/GoogleDrive/MDrive/altera/13.0sp1/MIPS/MIPS_System_Students_nopll_ts/MIPS_System - MIPS_System

File Edit View Project Assignments Processing Tools Window Help

Project Navigator

Entity

- Cyclone II: EP2C35F672C6
 - MIPS_System
 - Addr_Decoder:Decoder
 - ram2port_inst_data:Inst_Data_Mem
 - TimerCounter:Timer
 - mips:mips_cpu
 - ClockBlk:pll0
 - GPIO:uGPIO

Tasks

Flow: Compilation Customize...

Task	Status
Compile Design	✓
Analysis & Synthesis	✓
Edit Settings	
View Report	
Analysis & Elaboration	✓
Partition Merge	
Netlist Viewers	
RTL Viewer	

```
88 .memwrite      (data_we), // data_we: active high
89 .memaddr      (data_addr),
90 .memwritedata  (write_data),
91 .memreaddata   (read_data));
92
93 assign data_re = ~data_we;
94
95
96 // Port A: Instruction
97 // Port B: Data
98 ram2port_inst_data Inst_Data_Mem (
99     .address_a    (inst_addr[12:2]),
100    .address_b     (data_addr[12:2]),
101    .byteena_b     (4'b1111),
102    .clock_a       (clk90),          //was clk90
103    .clock_b       (clk180),        //was clk180
104    .data_a        (),
105    .data_b        (write_data),
106    .enable_a      (1'b1),
107    .enable_b      (~cs_mem_n),
108    .wren_a        (1'b0),
109    .wren_b        (data_we),
110    .q_a           (inst),
111    .q_b           (read_data_mem));
112
113
114 Addr_Decoder Decoder (
115     .Addr         (data_addr),
116     .CS_MEM_N     (cs_mem_n),
117     .CS_TC_N      (cs_timer_n),
118     .CS_UART_N    (),
```

Dual Port RAM Megafunction

MegaWizard Plug-In Manager [page 3 of 10]

RAM: 2-PORT

Parameter Settings | EDA | Summary

General > Widths/Blk Type > Clks/Rd, Byte En > Regs/Clocks/Adrs > Mem Init >

ram2port_inst_data

data_a[31..0] address_a[10..0] wren_a

data_b[31..0] address_b[10..0] wren_b

byteena_b[3..0]

clock_a enable_a

clock_b enable_b

Block Type: AUTO

2048 Word(s) RAM

q_a[31..0]

q_b[31..0]

What clocking method do you want to use?

- ☐ Single clock
- ☐ Dual clock: use separate 'read' and 'write' clocks
- ☐ Dual clock: use separate 'input' and 'output' clocks
- ☐ No clock (fully asynchronous)
- ☒ Dual clock: use separate clocks for A and B ports

☐ Create a 'rden' read enable signal

Byte Enable Ports

- ☐ Create byte enable for port A
- ☒ Create byte enable for port B

What is the width of a byte for byte enables? 8 bits

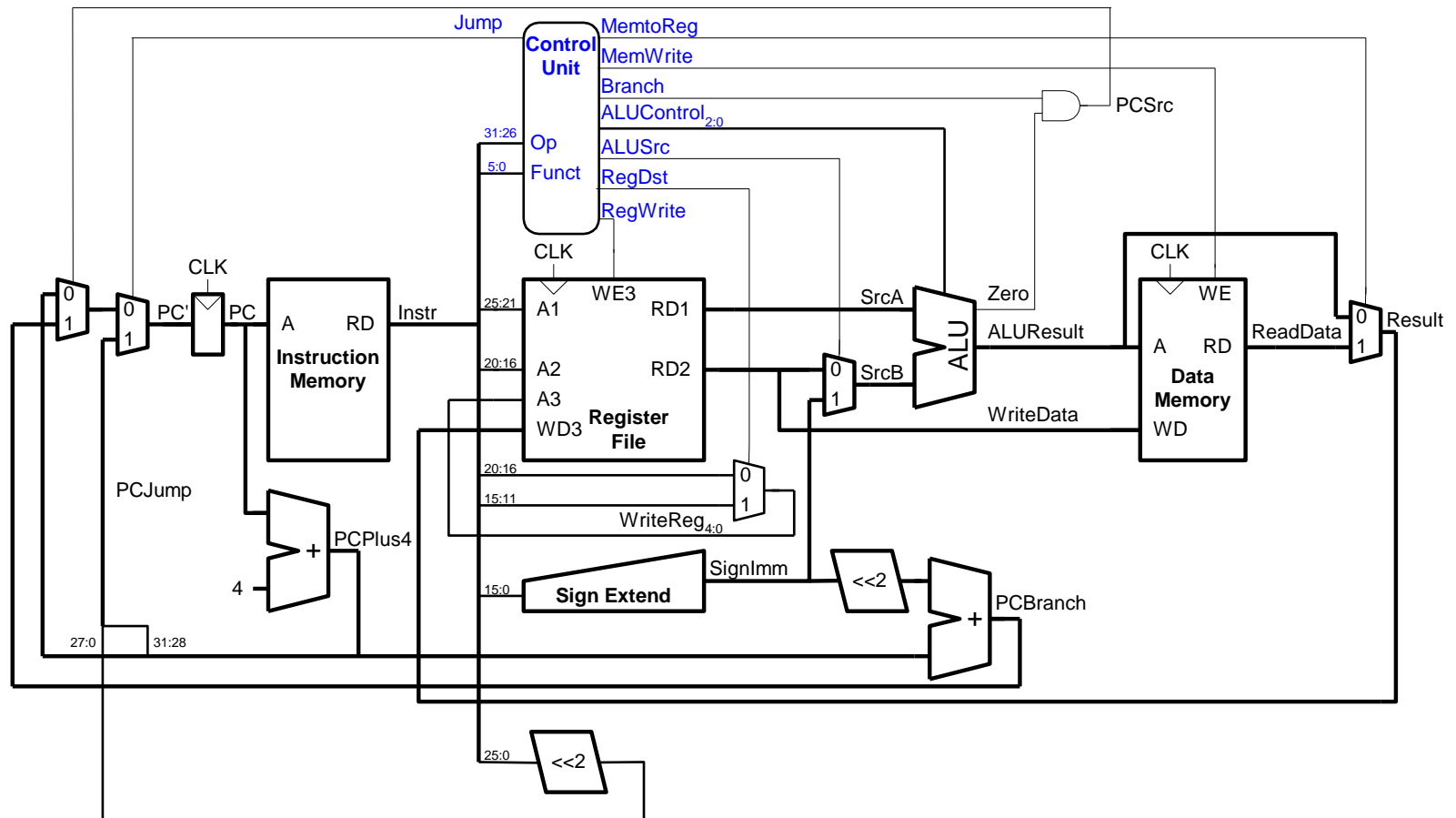
Resource Usage

16 M4K

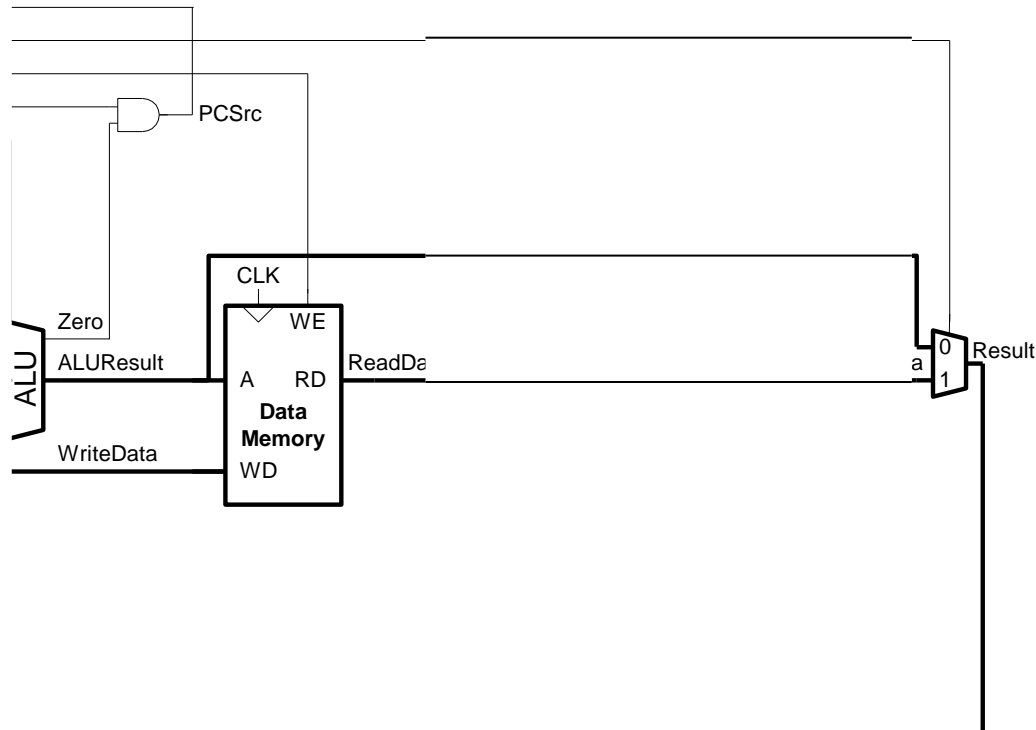
Cancel < Back Next > Finish

A: Instruction
B: Data

Full Data Path



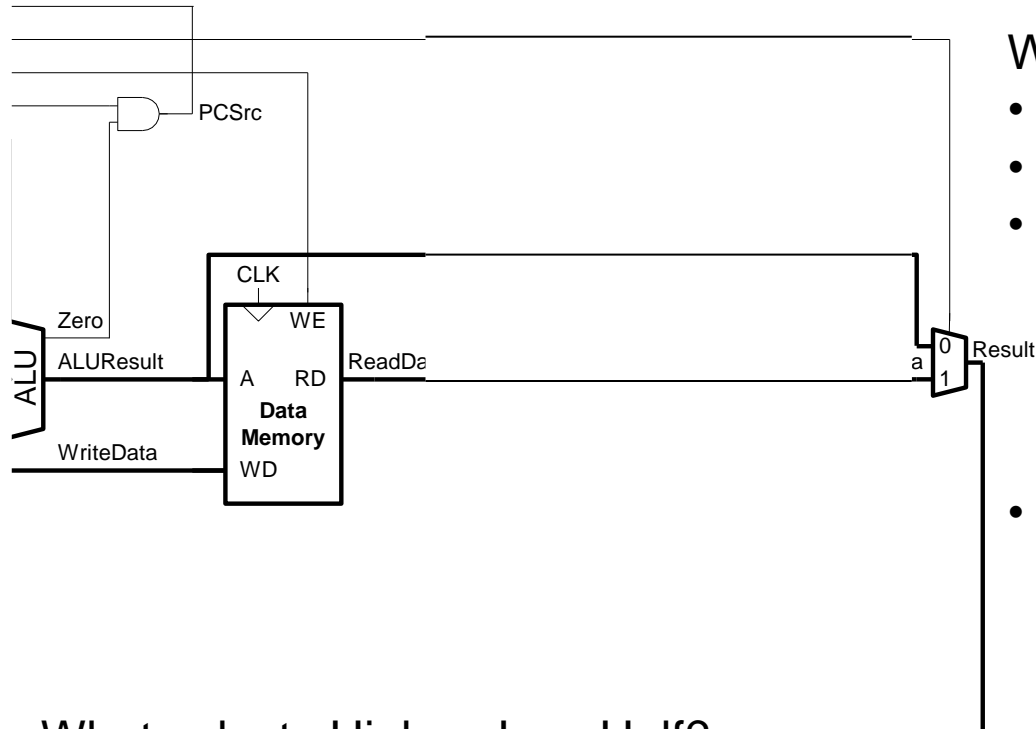
Current Data Path



What should the “Result” be:

- Could be ALU Result
- Could be Read 32 bit word
- For LH/LHU
 - Could be Sign/Zero Low Half Word
 - Could be Sign/Zero High Half Word
- For LB/LBU
 - Could be Sign/Zero Byte[0]
 - Could be Sign/Zero Byte[1]
 - Could be Sign/Zero Byte[2]
 - Could be Sign/Zero Byte[3]

Current Data Path

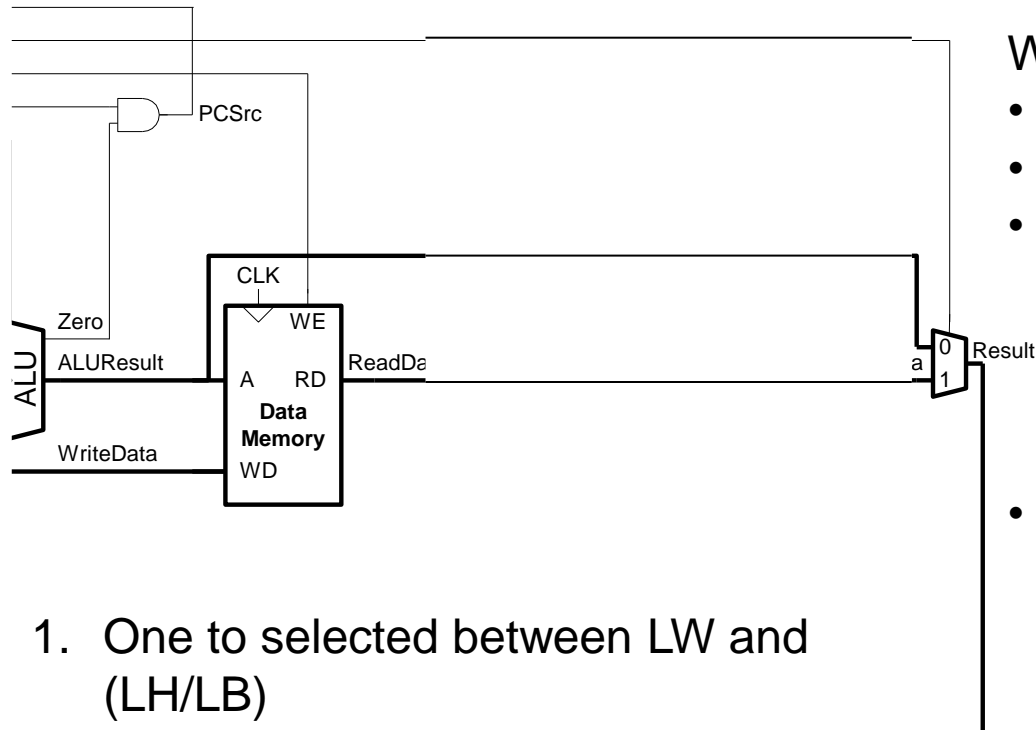


What selects High or Low Half?
What selects which Byte?

What should the “Result” be:

- Could be ALU Result
- Could be Read 32 bit word
- For LH/LHU
 - Could be Sign/Zero Ext. Low Half Word
 - Could be Sign/Zero Ext. High Half Word
- For LB/LBU
 - Could be Sign/Zero Ext. Byte[0]
 - Could be Sign/Zero Ext. Byte[1]
 - Could be Sign/Zero Ext. Byte[2]
 - Could be Sign/Zero Ext. Byte[3]

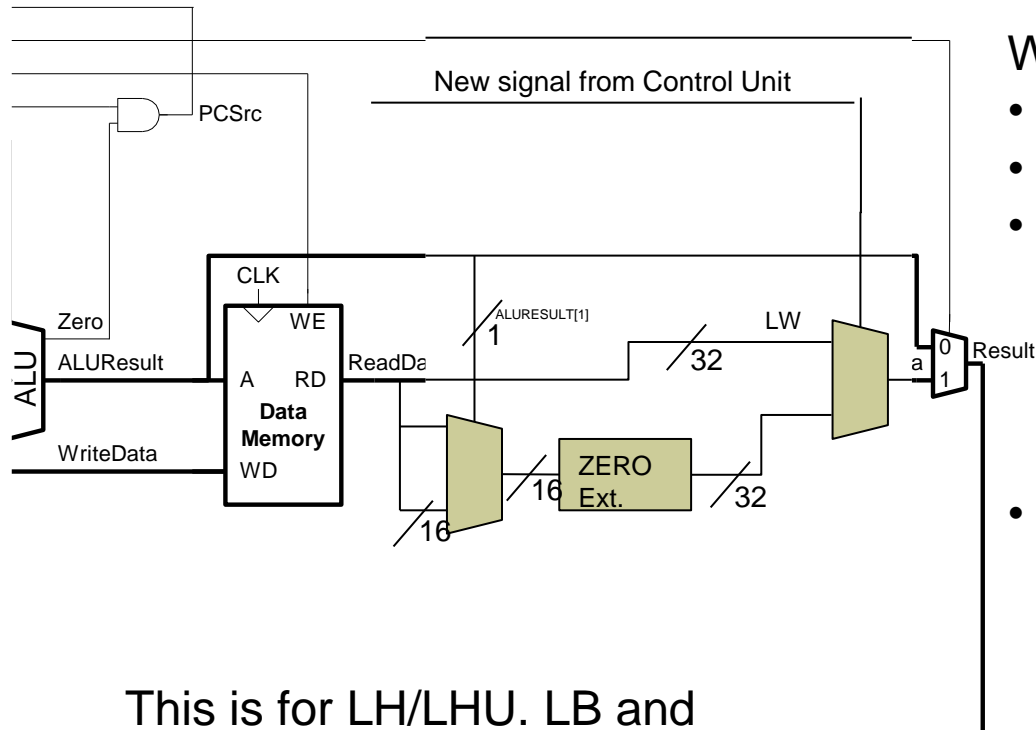
How many multiplexers are needed?



What should the “Result” be:

- Could be ALU Result
 - Could be Read 32 bit word
 - For LH/LHU
 - Could be Sign/Zero Ext. Low Half Word
 - Could be Sign/Zero Ext. High Half Word
 - For LB/LBU
 - Could be Sign/Zero Ext. Byte[0]
 - Could be Sign/Zero Ext. Byte[1]
 - Could be Sign/Zero Ext. Byte[2]
 - Could be Sign/Zero Ext. Byte[3]
1. One to selected between LW and (LH/LB)
 - What is its selection signal?
 2. One to select the appropriate Half Word or BYTE
 - What is its selection signal(s)?
 - Also need a Sign or Zero Extender

How many multiplexers are needed?



This is for LH/LHU. LB and LBU would be 4 input 8 bit mux and with two select lines to select the correct byte.

What should the “Result” be:

- Could be ALU Result
- Could be Read 32 bit word
- For LH/LHU
 - Could be Sign/Zero Ext. Low Half Word
 - Could be Sign/Zero Ext. High Half Word
- For LB/LBU
 - Could be Sign/Zero Ext. Byte[0]
 - Could be Sign/Zero Ext. Byte[1]
 - Could be Sign/Zero Ext. Byte[2]
 - Could be Sign/Zero Ext. Byte[3]

Code for testing instructions

1. Your code should cover all combinations of inputs
 - For Boolean operators this includes:
 - 0,0 : 0,1 : 1,0 : 1,1
 - Your operations should be able to be easily spotted in the ModelSim / Signaltap Waveforms
 - The numbers used shouldn't require too much thought to work out whether it produces the correct answer:
 - Which is easier to calculate
 - 0x01234567 & 0x76543210
 - 0xFFFF0000 & 0xFF00FF00

Testing LB, LBU, LH, LHU

- You need to load a 32 bit value into an appropriate memory location.
 - The memory is mapped between 0x0 -> 0x2000.
 - You should pick a word aligned location higher than where your program is stored
 - Something like 0x200
 - You should use SW to store an appropriate word aligned 32 bit value.
 - The value used should allow the result to differentiate between LB & LBU (or LH & LHU)
 - For LB/LBU you need to load 4 bytes using four LB/LBU instructions
 - For LH/LHU you need to load 2 half-words using two LH/LHU instructions