Lab 04:
HW 3

Hunjun Lee
hunjunlee@hanyang.ac.kr

Finally …

- You will be implementing a single cycle CPU that you learned in the previous class
- Use the ALU and register file you made in the previous assignments
- The target architecture: MIPS
  - https://opencores.org/projects/plasma/opcodes
  - Do not need to implement every single instruction (implement what's provided in the following slides)
  - I'll provide the constant variables in a global.h and GLOBAL.v files

Target MIPS Assembly - 1

Assembly	Action	Opcode bitfields					
ADDU rd, rs, rt	rd = rs + rt	000000	rs	rt	rd	00000	100001
ADDIU rt, rs, imm	rt = rs + sign(imm)	001001	rs	rt	imm		
AND rd, rs, rt	rd = rs & rt	000000	rs	rt	rd	00000	100100
ANDI rt, rs, imm	rt = rs & zero(imm)	001111	rs	rt	imm		
LUI rt,imm	rt = imm << 16	000000	rs	rt	imm		
NOR rd,rs,rt	rd = ~(rs | rt)	000000	rs	rt	rd	00000	100111
OR rd,rs,rt	rd = rs | rt	000000	rs	rt	rd	00000	100101
ORI rt,rs,imm	rt = rs | zero(imm)	001101	rs	rt	imm		
SLT rd,rs,rt	rd = rs < rt	000000	rs	rt	rd	00000	101010
SLTI rt,rs,imm	rt = rs < sign(imm)	001010	rs	rt	imm		
SLTIU rt,rs,imm	rt = rs < sign(imm)	001011	rs	rt	imm		
SLTU rd,rs,rt	rd = rs < rt	000000	rs	rt	rd	00000	101011
SUBU rd,rs,rt	rd = rs - rt	000000	rs	rt	rd	00000	100011
XOR rd,rs,rt	rd = rs ^ rt	000000	rs	rt	rd	00000	101010
XORI rt,rs,imm	rt = rs ^ zero(imm)	001010	rs	rt	imm		

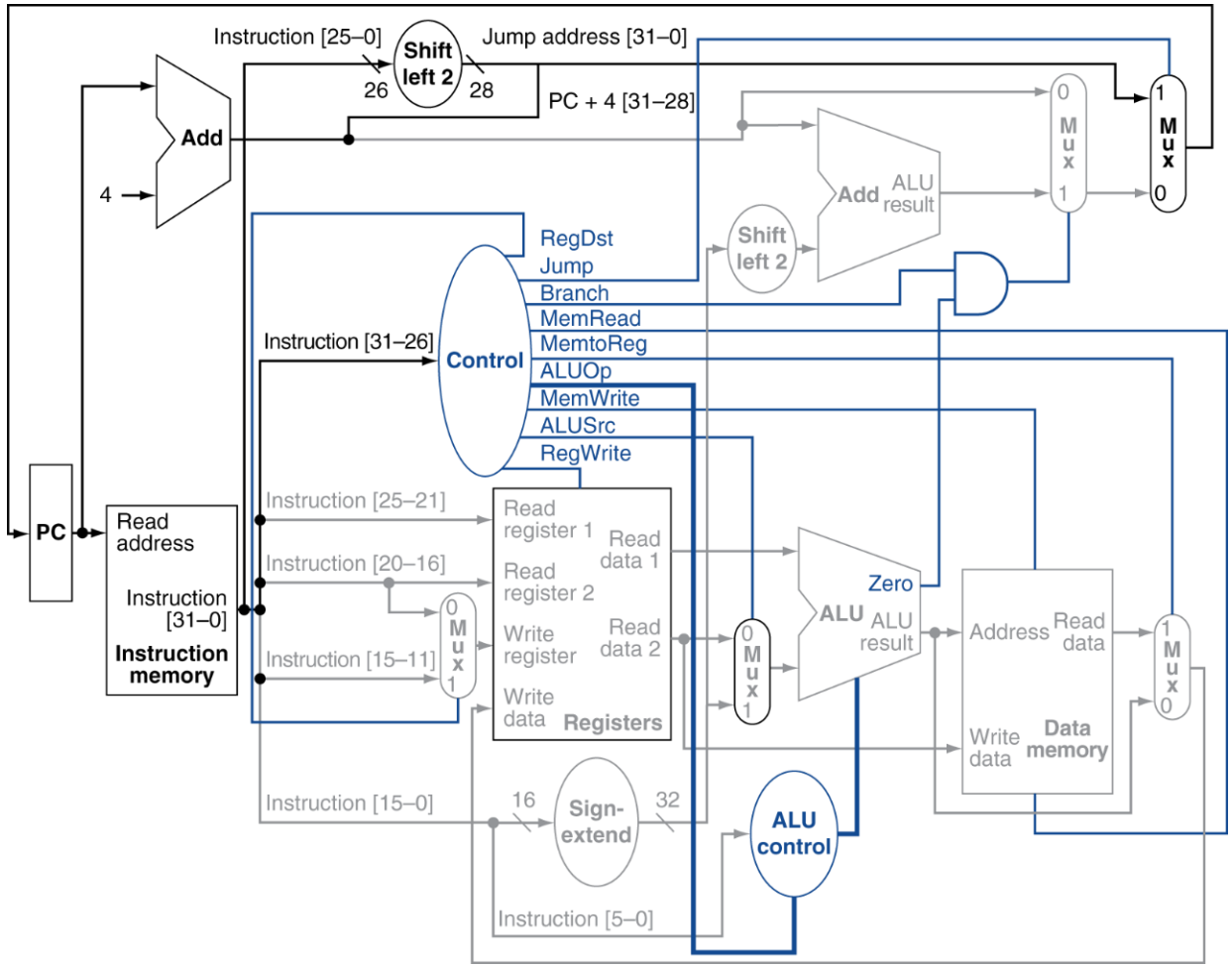
Target MIPS Assembly - 2

Assembly	Action	Opcode bitfields					
SLL rd, rt, sa	rd = rs << sa	000000	rs	rt	rd	sa	000000
SRA rd, rt, sa	rd = rt >> sa (arithmetic)	000000	rs	rt	rd	sa	000011
SRL rd, rt, sa	rd = rt >> sa (logical)	000000	rs	rt	rd	sa	000010
BEQ rs, rt, offset	if(rs == rt) pc += sign(offset) << 2	000100	rs	rt	offset		
BNE rs, rt, offset	if(rs != rt) pc += sign(offset) << 2	000101	rs	rt	offset		
J target	pc=pc_upper|(target << 2)	000010	target				
JAL target	r31 = pc; pc=pc_upper|(target << 2)	000011	target				
JR rs	pc = rs	000000	rs	000000000000000			001000
LW rt, offset(rs)	rt = MEM[sign(offset) + rs]	100011	rs	rt	offset		
SW rt, offset(rs)	MEM[sign(offset) + rs] = rt	101011	rs	rt	offset		

Implement a CPU that supports these ISAs

Target CPU Microarchitecture

- Implement what you learned in the class, but w/ **additional support for jal and jr**



Assignment

- **Files:**
  - **global.h + GLOBAL.v** **(provided)**
    - Include constant values (opcode, funct …)
  - **ALU.cpp/h + ALU.v** **(HW1)**
    - You can copy & paste the code from HW1
  - **RF.cpp/h + RF.v** **(HW2)**
    - You can copy & paste the code from HW2
  - **MEM.cpp/h + MEM.v** **(provided)**
    - I did it for you
  - **CTRL.cpp/h + CTRL.v** **(TODO)**
    - A controller to determine control signals
  - **CPU.cpp/h + CPU.v** **(TODO)**
    - The top module to combine these modules
  - **main.cpp + CPU_tb.v** **(provided)**
    - A testbench to test the CPU module

Memory module implementation

- I implemented a memory module that can simultaneously access instruction and data
- You will need to modify this code for multicycle CPU, but for now … use the provided file

```
 3 module MEM(
 4     input               clk,
 5     input               rst,
 6
 7     input [31:0]        inst_addr,
 8     output reg [31:0]   inst,
 9
10     input [31:0]        mem_addr,
11     input               MemWrite,
12     input               MemRead,
13     input [31:0]        mem_write_data,
14     output reg [31:0]   mem_read_data
15     );
16
17     reg [31:0] memory [0:8191];
18
19     initial begin
20         $readmemh("initial_mem.mem", memory);
21     end
22
23     // Read instruction + memory data
24     always @(*) begin
25         inst = memory[(inst_addr >> 2)];
26         if (MemRead)
27             mem_read_data = memory[(mem_addr >> 2)];
28         else
29             mem_read_data = 32'hz;
30     end
31
32     always @(posedge clk) begin
33         if (!rst)
34             if (MemWrite)
35                 memory[(mem_addr >> 2)] <= mem_write_data;
36     end
37
38 endmodule
```

End-to-End Debugging

- **Process #1:** Use MARS simulator to check how the architectural state changes for every instruction
- **Process #2:** Use CPP simulator to verify if the architectural state changes the same way as the MARS simulator ➔ Then, generate the debugging file for your Verilog code
  - I have implemented my own debugging procedure for you … of course you can modify this if you want
- **Process #3:** Use the debugging files from CPP simulator to debug your Verilog files

Submission

- Submission (Zip all the files)
  - For a two-people team: lab3_student_id1_student_id2.zip
  - For a one-person team: lab3_student_id.zip
    - You must follow the format (-10% for wrong file format)
    - Example:
      - lab3_2020102030.zip
      - lab3_2020102030_2022103040.zip
  - The zip file should contain:
    - every cpp/h/v files
    - lab3_report.pdf

Submission

- You need to write a 4+-page report
  - Explain the overall structure and how you implemented each program
  - Write down the difficulties if you had one
  - Draw the hardware modules if needed …
  - How did you implement JAL / JR instruction and modified the uarchitecture?

- Due: Fri. April 18th
  - 1 week delay: -20%
  - 2 week delay: -50%
  - Further delay: 0 point