Computer Architecture

Single-cycle RISC-V CPU

B06901158 洪正維　　B06901061 王廷峻

CPU Architecture

Based on the CPU design in the lecture slide, we additionally support three instructions: `JAL`, `JALR`, `AUIPC`, and multi-cycled multiplication (`MUL`). In response to the first three instructions, we modify the control unit of the program counter (`PC`) and the `MemToReg` multiplexer. Meanwhile, ALU integrates a multi-cycled MulDiv module to support `MUL`. It will deliver an `alu_ready` signal to other modules when a multiplication operation is done. A detailed architecture is shown in Figure1.

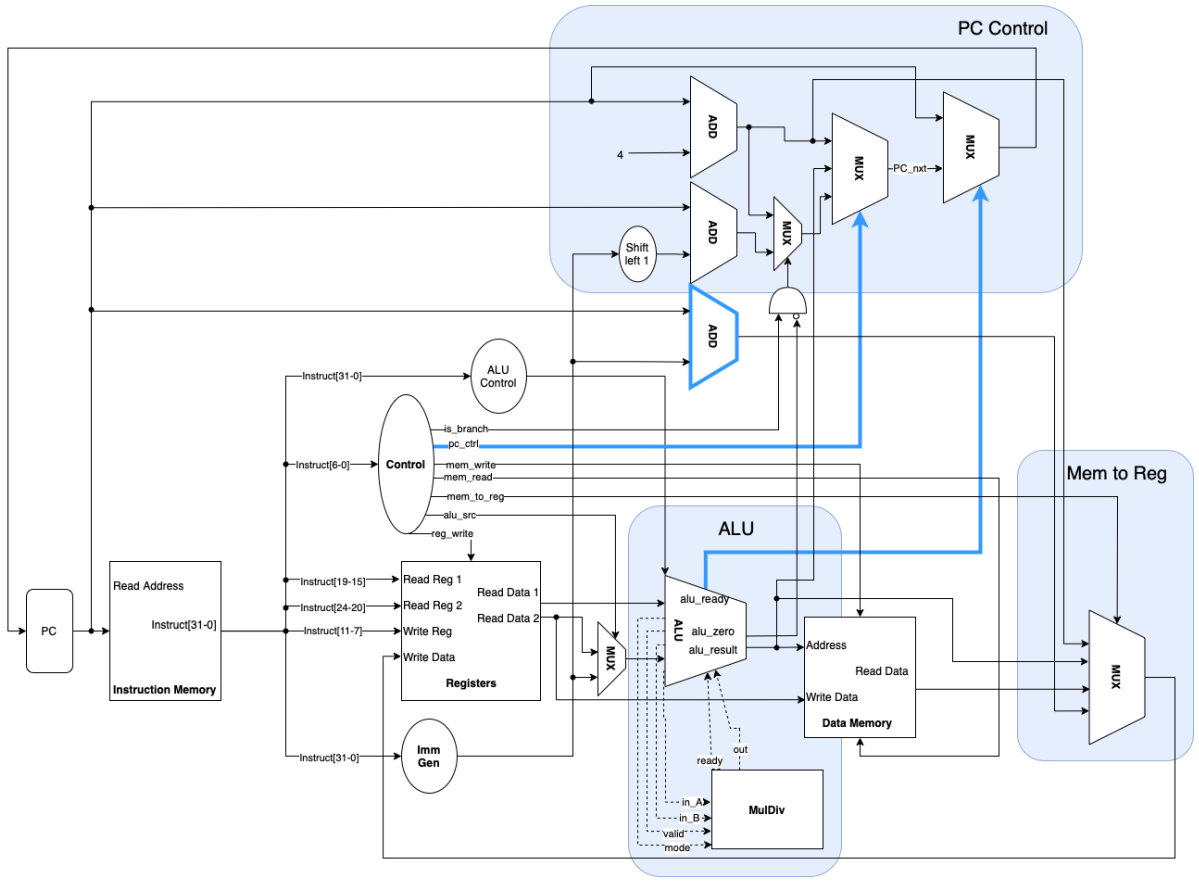

Figure1: CPU Architecture. Modules covered with blue blocks are modified and blue wires are added to support new instructions.

The PC control unit is a three-staged multiplexer:

Stage	Input	Control Signal	Supported Instruct.

1	PC+4, PC+(*imm<<1)	is_branch, alu_zero	Branch instruct., JAL
2	PC+4, alu_result, Stage-1 output	pc_ctrl	Others, JALR, (stage-1 supported instrut.)
3	PC, Stage-2 output	alu_ready	Halt until MUL is done

*imm refers to the extended immediates generated by the ImmGen module.

Table 1: Details of the three-staged PC control multiplexer

The MemToReg multiplexer is 4-to-1:

Input	Instruct.
PC+4	JAL, JALR
alu_result	others
memory read data	LW
PC+imm	AUIPC

Table 2: Details of the MemToReg multiplexer

Datapath of Instructions

Supported instructions are listed in Table3. Since R, I, I(LW), S, and B types instructions are mentioned in the lecture slides, we will skip their datapathes and focus on I(JALR), U, and UJ types instead.

Type	Instruct.
R	ADD, SUB, SLL, SLT, SLTU, XOR, SRL, SRA, OR, AND, MUL
I	ADDI, SLLI, SLTI, SLTIU, XORI, SRLI, SRAI, ORI, ANDI
I (JALR)	JALR
I (LW)	LW
S	SW
B	BEQ, BNE
U	AUIPC
UJ	JAL

Instructions colored blue are mandatory, those colored black are bonus.

Table 3: Supported instructions

I Type - `JALR`

When executing `JALR`, `PC_nxt` will be set to `RS1+imm`, and the CPU will write `PC+4` into the destination register as shown with the red wires in Figure 2.

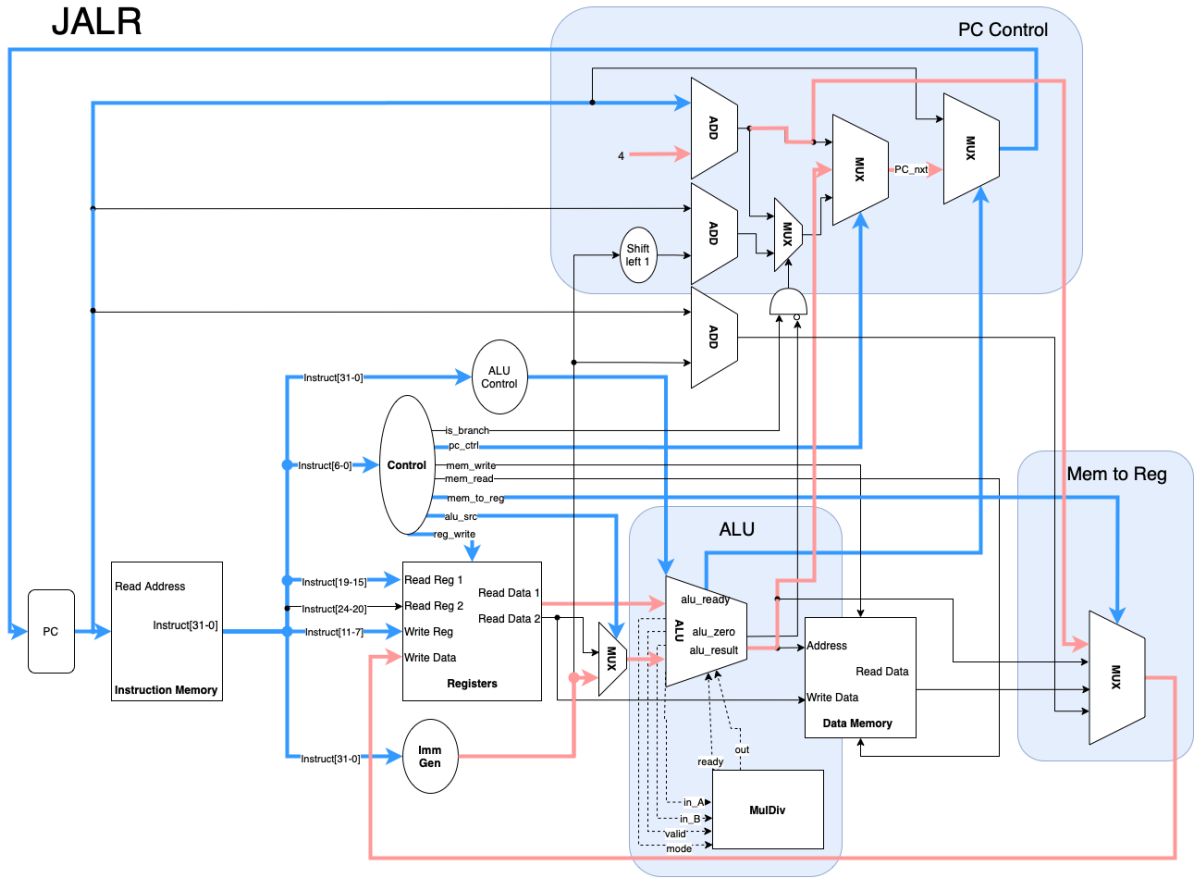

Figure 2: Datapath of `JALR`. Wires colored blue and red are activated while the CPU executes `JALR`.

UJ Type - `JAL`

When executing `JAL`, `PC_nxt` will be set to `PC+(imm << 1)`, and CPU will write `PC+4` into the destination register as shown with the red wires in Figure 3.

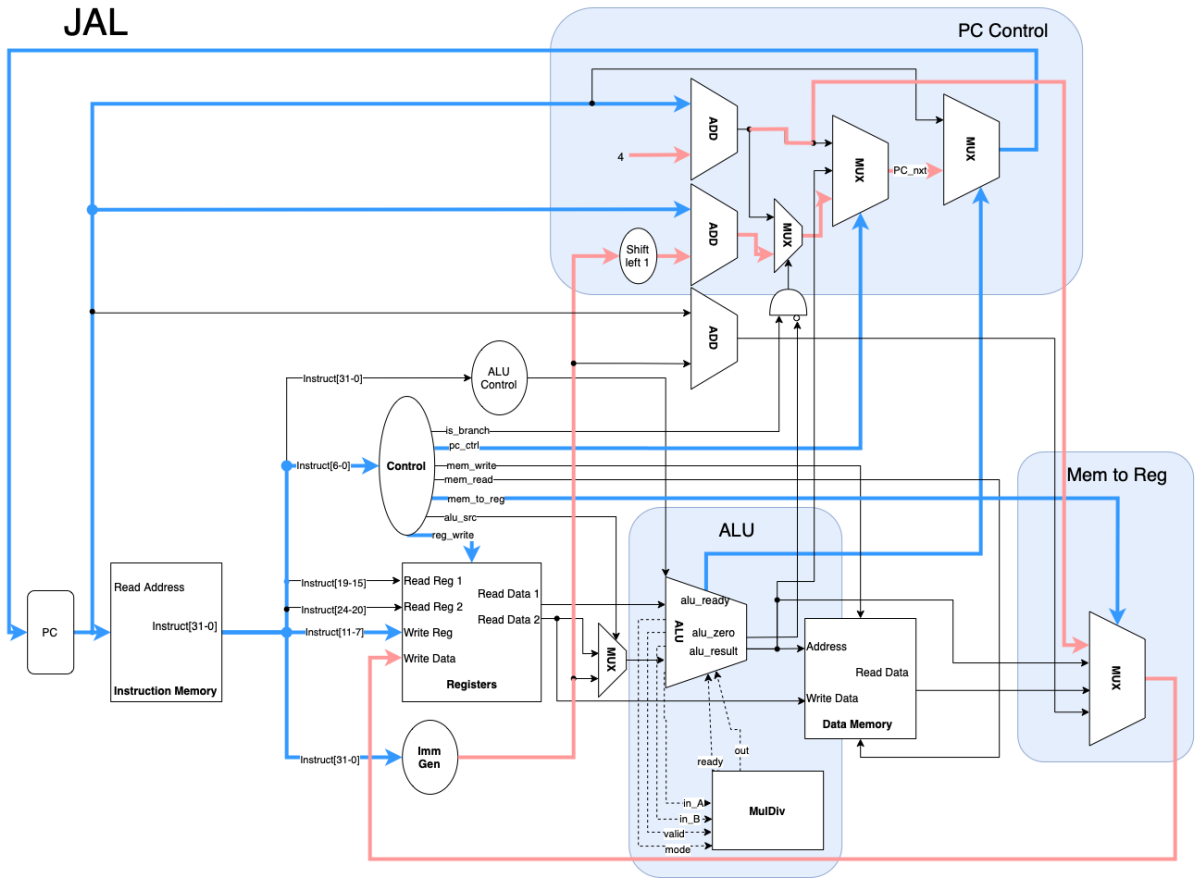

Figure 3: Datapath of `JAL`

U Type - `AUIPC`

When executing `AUIPC`, `PC_nxt` will be set to `PC+4`, and CPU will write `PC+imm` into the destination register as shown with the red wires in Figure 4.

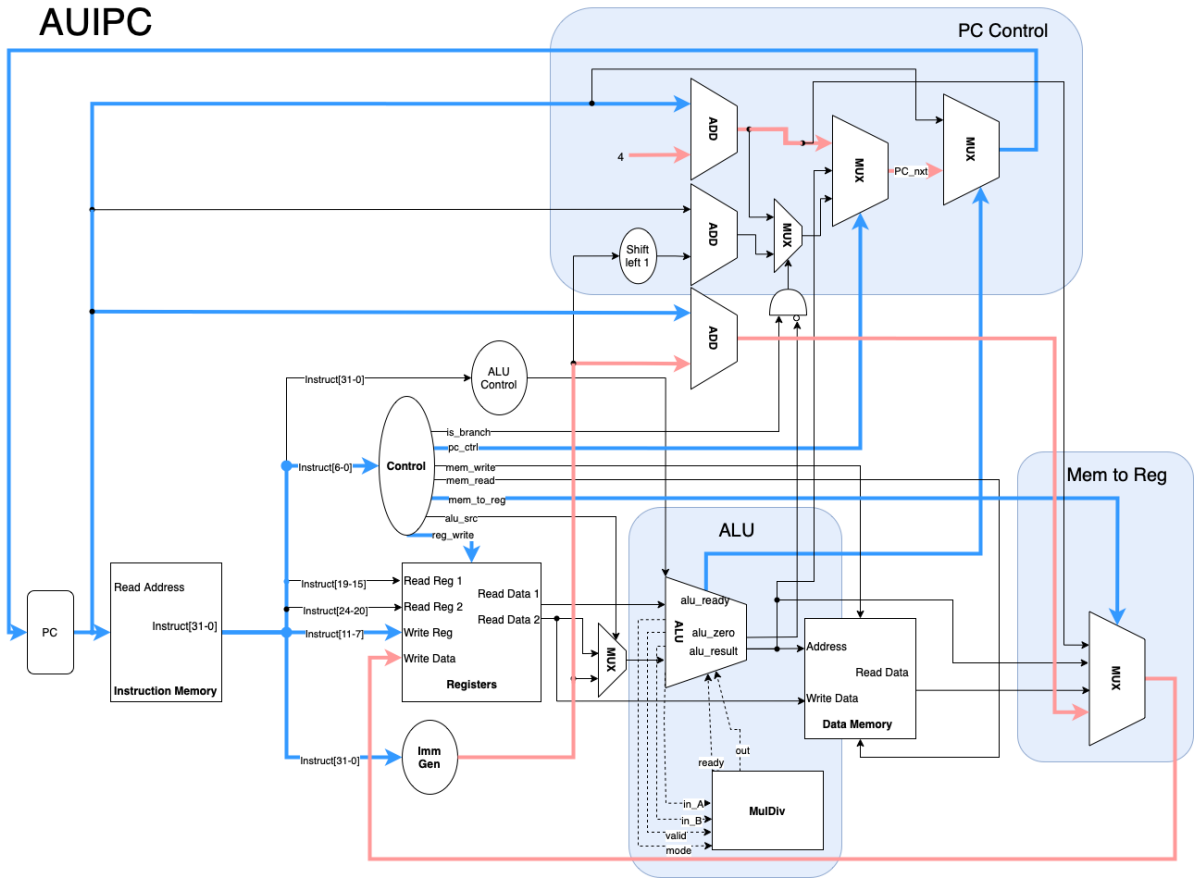

Figure 4: Datapath of `AUIPC`

Design of ALU (Handling multi-cycle instructions)

ALU integrates a MulDiv module and delegates the multiplication operation to it. To simplify the explanation of ALU operations, we will skip MulDiv's part. Please see the Figure 5 below for more details about MulDiv.

ALU has five inputs: `clock`, `rst_n`, `input1`, `input2`, and `alu_ctrl`, and three outputs: `alu_zero`, `alu_ready`, and `alu_result`. When the CPU executes instructions other than `MUL`, `alu_ready` is always True, and `alu_result` is always ready. On the other hand, when it executes `MUL`, it will generate two inputs: `valid` and `mode`, to feed into MulDiv. `alu_ready` changes to False and `alu_result` isn't ready until the ready signal from MulDiv rises (Specifically, MulDiv requires 32 cycles for calculation).

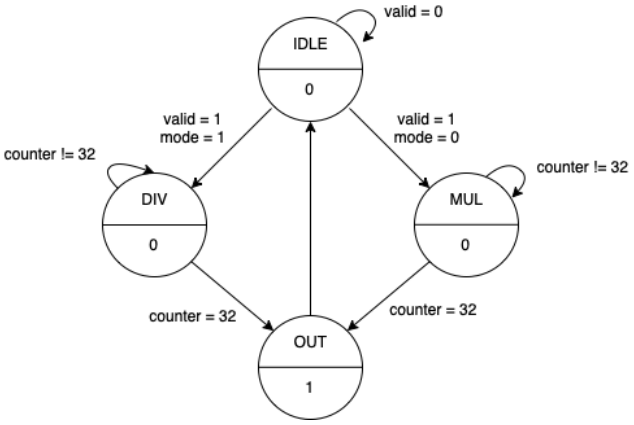

Figure 5: State diagram for MulDiv

As soon as `alu_ready` rises, the last stage of PC Control multiplexer selects `PC+4` and the program continues to execute the next instruction.

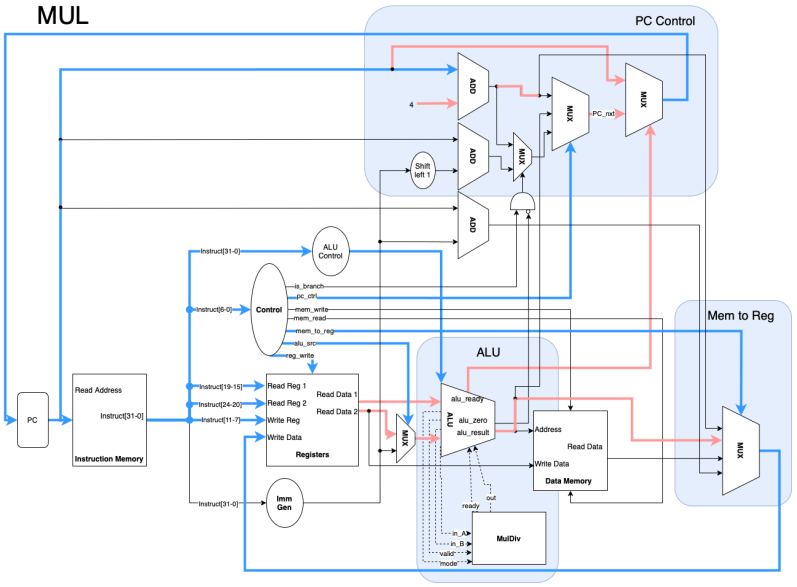

Figure 6: Datapath of `MUL`

Simulation

Record total simulation time (CYCLE = 10 ns)
Leaf: a = 5, b = 6, c = 8, d = 0

```
Simulation complete via $finish(1) at time 255 NS + 0
```

Fact: n = 8

```
Simulation complete via $finish(1) at time 3875 NS + 0
```

(Bonus) HW1: n = 10

```
Simulation complete via $finish(1) at time 675 NS + 0
```

Coding Style Check

Snapshot the “Register table” in Design Compiler

```
Inferred memory devices in process
        in routine MulDiv line 95 in file
                'MulDiv.v'.
===============================================================================
|    Register Name    |   Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
===============================================================================
|      shreg_reg      | Flip-flop |  64   |  Y  | N  | N  | N  | N  | N  | N  |
|     alu_in_reg      | Flip-flop |  32   |  Y  | N  | N  | N  | N  | N  | N  |
|      state_reg      | Flip-flop |   3   |  Y  | N  | Y  | N  | N  | N  | N  |
|     counter_reg     | Flip-flop |   5   |  Y  | N  | N  | N  | N  | N  | N  |
===============================================================================
```

```
Inferred memory devices in process
        in routine CHIP line 214 in file
                '/home/raid7_2/userb06/b06158/CA2021/CA-Final-Project-mike-implem
===============================================================================
|    Register Name    |   Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
===============================================================================
|       PC_reg        | Flip-flop |  31   |  Y  | N  | Y  | N  | N  | N  | N  |
|       PC_reg        | Flip-flop |   1   |  N  | N  | N  | Y  | N  | N  | N  |
===============================================================================
Warning:  /home/raid7_2/userb06/b06158/CA2021/CA-Final-Project-mike-implement/c1/
Warning:  /home/raid7_2/userb06/b06158/CA2021/CA-Final-Project-mike-implement/c1/

Inferred memory devices in process
        in routine reg_file line 247 in file
                '/home/raid7_2/userb06/b06158/CA2021/CA-Final-Project-mike-implem
===============================================================================
|    Register Name    |   Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
===============================================================================
|       mem_reg       | Flip-flop |  995  |  Y  | N  | Y  | N  | N  | N  | N  |
|       mem_reg       | Flip-flop |  29   |  Y  | N  | N  | Y  | N  | N  | N  |
===============================================================================
```

Observation

Unlike C++ or other software programming languages, it's harder to debug in a hardware description language like Verilog. It’s important for us to choose which signal to display in nWave.

If we get an error "Simulation time exceeds" when we run our simulation, Usually there is a problem with the branch instruction or the program counter, which causes the simulation to be stuck in a certain cycle and cannot continue. We’d better check mem_rdata_I or the PC signal; that’s how we know that program first evaluates + then <<(left shift) in “PC = PC + immediate << 1” when it takes a branch in JAL instruction.

Then in the factorial test pattern,we use the muldiv submodule in our ALU submodule. We need to be careful with the state in the ALU unit and the state in our multiple unit in hw2, especially stalling the PC when fetching MUL instruction and deciding when to output alu_result in a MUL instruction and other instructions. Checking state signal, input signal, ready signal and output signal both in ALU unit and muldiv submodule would be helpful.

When writing assembly code in our bonus problem, we need to change the register name in our hw1 code to store the result in x10 because there’s a “sw x10, 4(t0)” instruction in the do-not-modify part. Also we should avoid bge instruction by using equivalent beq instruction to represent it since our processor does not recognize it. Because operands of multiple are powers of 2(2,8), we can replace mul by slli to accelerate running time. If we find that the result is not stored in the right place in the memory, we can add some dummy code when it’s stored in a lower address.

In conclusion, we learn how to find errors in Verilog and fix them in this project. We need to simulate running the program in our brain many times and try to find out which part went wrong when debugging. It is better for us to draw a flowchart completely on paper before starting coding our program.

Workload Distribution

Item	B06901061	B06901158
Impl. CHIP.v	v	
Impl. module ALU	v	
Impl. module Control	v	
Impl. module ALUControl	v	
Impl. module IMMGEN	v	
Debug: CHIP.v		v

Testing: Pattern 1 Leaf		v
Testing: Pattern 2 Fact		v
impl. hw1.s/hw1_text.txt		v
Testing: Pattern 3 HW1		v
Report	v	v