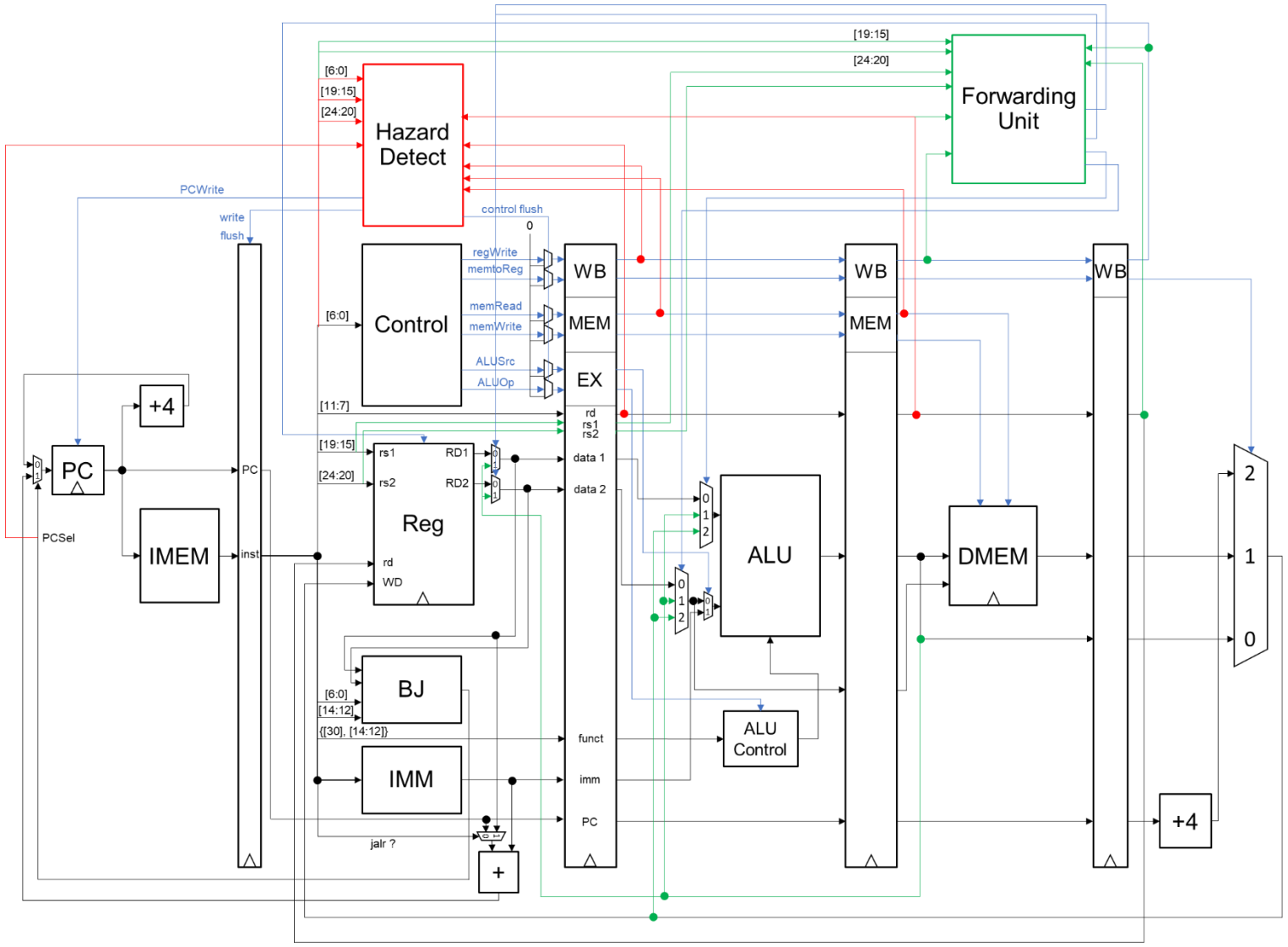
I. Architecture

This architecture works for add, addi, sub, and, andi, or, ori, slt, slti, lw, sw, beq, bne, blt, bge, jal, jalr. From single cycle CPU to pipeline CPU.



II. Stages and Pipeline Registers

Single Cycle CPU is ideal, some operations cannot be finished in a short time, such as memory access, so CPU is divided into stages. In this architecture, 5-stage pipeline CPU is used.

The CPU is divided into 5 stages to make the overall speed faster. Each stage operates on different instructions in parallel during a clock cycle. Pipeline registers are required to store all of the data generated in a stage but needed in the subsequent stages.

- Instruction Fetch (IF):

  PC output the instruction address to the instruction memory, and the new instruction is fetched. Store the instruction and its address to the pipeline register for later use.

- Instruction Decode (ID):

  The instruction is decoded into pieces to determine what to do next.
  The control unit generates control signals based on the option code (instruction [6:0]).
  2 data at specified address in the register file are extracted, but they may not be up-to-date, the solution to this problem is in the following pages.
  Immediate is also generated here. It is added to the instruction address at this stage if the option code indicates that the instruction is a branch or jal. It is added to the "updated" data 1 if the instruction is jalr. It is transmitted to EX for I-type instructions.
  The address of next instruction is determined here.

- Execution (EX):

  A multiplexer selects the input B of the ALU, the selection bit (`ALUSrc`) can be obtained from the control unit in the ID stage after passed through a pipeline register.
  ALU Control Unit determines the operation the ALU should perform based on `ALUOp`.
  The simple ALU supplies only 5 operations, `+`, `-`, `&`, `|`, `<`.

- Memory / Cache Access (MEM):

  Access the data memory as needed. 2 control signals are used: `memRead` and `memWrite`.

- Write Back (WB):

  Write back data to register file. 2 control signals are needed: `regWrite` indicates whether there is data to be written, `memtoReg` is responsible for selecting the written data.

Although all pipeline registers behave like tunnels, it's easier to implement it as 4 modules, IF_ID, ID_EX, EX_MEM, MEM_WB.

III. Modules

i. Program Counter

Add a control signal: `PCWrite`. The PC can be updated only if `PCWrite` is asserted.

ii. Immediate Generator

For simplicity, I put the “shift left 1” operation for branch instructions and jal into this module.

iii. Branch and Jump Controller - BranchJump

If the option code is corresponding to jump (jal or jalr), select input 1 for PC (`PCSel = 1'b1`).
If it the option code for branch is received, determine whether the branch should be taken.
Else, select PC + 4 (`PCSel = 1'b0`).

IV. Pipeline Hazard, Cases, and Solution

i. Structural Hazard

Structural hazard is already solved by splitting IMEM and DMEM and the scheme to write data to register file in the first half of the clock cycles and read data from it in the second half.

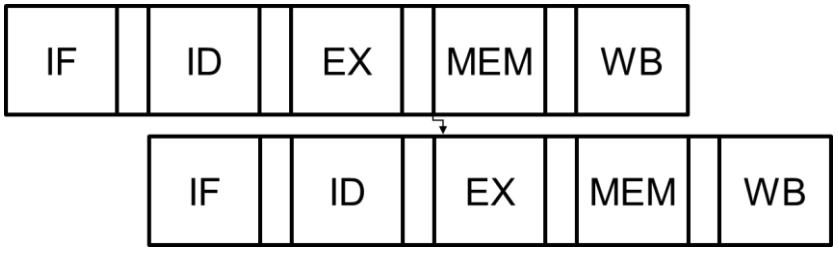
ii. Data Hazard

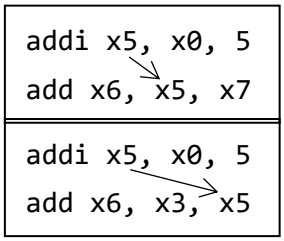
The data have not been stored to the register file but the next instruction needs it.

Case 1: R/I - R/I

The correct data is the output of the ALU, and it will be written to register file in the stage WB, but next instruction access it in the stage ID and use it in the stage EX. The data should be forwarded from pipeline register EX/MEM to stage EX, and 2 MUX are added to select the correct data for ALU input 1 and ALU input 2 because we don’t know which input require the data.

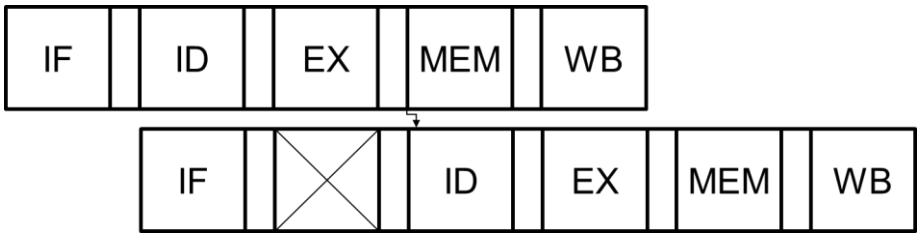
Note that we are not connecting a wire directly from the output of ALU to input of ALU because the data is used in the next clock cycle and should pass through a pipeline register first.





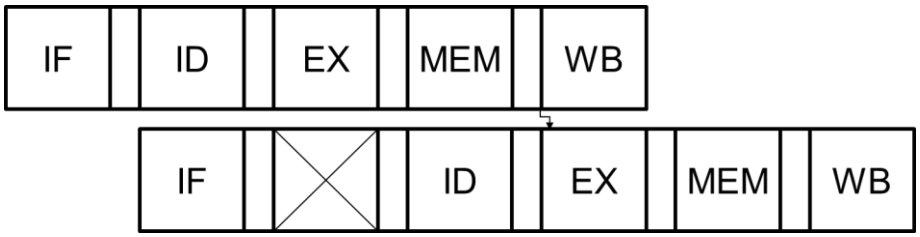
Case 2: R/I - B/jalr

In this architecture, whether a branch is taken or not is determined in the ID stage, so the result should be forwarded from EX/MEM to ID. But we still need to delay the branch instruction for 1 clock cycle (1 stall) to wait for the execution of ALU. 2 MUX is added to select the correct input of branch and jump controller. As for jalr, PC = reg[rs1] + imm is executed in ID.



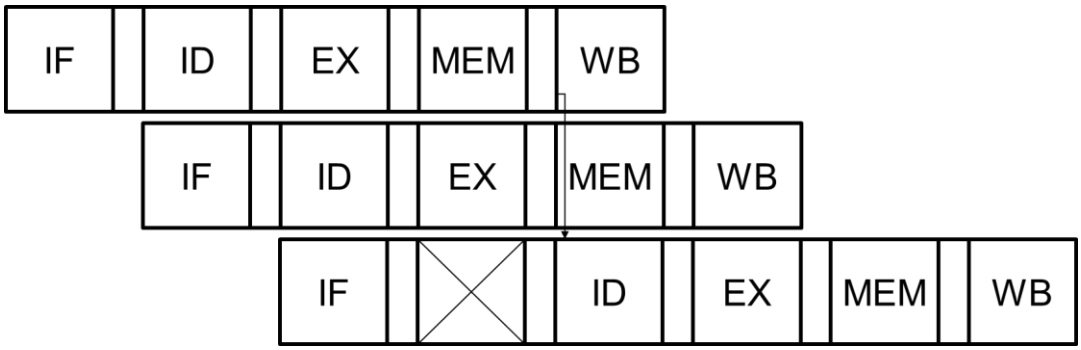
Case 3: load - R/I

The correct data is fetched in MEM for load instructions, R/I-type instructions operate the data in EX. The data is forwarded from MEM/WB to EX after 1 stall.



Case 4: load - B/jalr

In case 3, this problem is simplified to load - nop - B/jalr. Another stall is needed.



iii. Control Hazard and Jump

For branch instructions, it is not possible to determine whether or not to take the branch in the IF stage, so the CPU does not know which instruction should be fetched next.

For jump instructions, the address to jump to can be known in ID stage in this architecture, but we cannot fetch the next instruction in the same clock cycle.

Case 5: B (taken) or J - any

Flush the data stored in the IF/ID registers because the instruction should not be executed and its address is not needed now, and don't write to the IF/ID registers. But the correct instruction address is known, so PC should store the address of the next instruction.

V. Implementation of Hazard Detection and Forwarding

i. Forwarding Unit

Forwarding technique is applied in case 1, 2, 3, and 4. When the correct data has not been written to the register file, we should insert some nop(s) or forward the correct data to meet the requirement. So, a judgement would be whether `regWrite` is asserted.

`rd` must not be 0 because x0 cannot store anything other than 0, and this becomes a clue of whether the control signal is flushed or not.

Lastly, we check whether the data is dependent, in other words, instruction progressing to execution or judging may have got the wrong data from the register file. The data is dependent if the executed data is the result of the previous instruction. Take stall(s) into consideration and determine where previous instruction is.

ii. Hazard Detector

4 control signals are required to control the behavior of the registers:
`PCWrite`: whether the program counter should be updated.
`IFID_write`: whether IF/ID should be updated.
`IFID_flush` is asserted means the data in the registers are discarded.
`controlFlush` is asserted when the control signals generated in the ID stage should be abandoned.

After an instruction progresses to ID, observe specific control signal(s) or option code of previous and current instructions to identify them, and check the data addresses to determine whether a hazard will occur or not.

List the cases above:

In case 1, forwarding data solves the problem.

In case 2, 3, and 4, insert a nop, or stall for a clock cycle, and set the 4 control signals to stop the instruction from progressing to next stage. PC and IF/ID registers should not be updated, but the data is still stored in them. Control signals generated in the control unit should be flushed to 0 to stop the instruction.

In case 5, flush the entire instruction. The IF/ID registers should be flushed to 0 and no data is stored in this clock cycle. PC is updated to the address of the new instruction. Control signals generated in the control unit should not be flushed because they belong to the previous instruction.