# Data Hazards

## Forwarding

Data hazard on the input: when an operation has as input either rs1, rs2, or both.

Data hazard on the output: when an operation has output in rd.

|  |  |  |  |
| --- | --- | --- | --- |
| **Operation** | Rs1 | Rs2 | Rd |
| Arithmetic Operations | X | X | X |
| Immediate Arithmetic | X |  | X |
| LUI/AUIPC |  |  | X |
| NOP |  |  |  |
| JALR | X |  | X |
| JAL |  |  | X |
| BRANCH |  |  |  |
| LOAD | X |  | X |
| STORE | X | X |  |

The data hazard on Rd for most instructions can be resolved using Forwarding. (Exception is LOAD, where the value in memory cannot be retrieved in advance)

Using the previous code for the Forwarding Unit, we see the following conditions to activate the Forward.

|  |
| --- |
| rs1 = rs1\_EX; rs2 = rs2\_EX;  if (RegWrite\_MEM && (rd\_MEM != 0) && (rd\_MEM == rs1) && !NoFwd\_EX) begin ForwardA = 2'b10; end  else if (RegWrite\_WB && (rd\_WB != 0) && !(RegWrite\_MEM && (rd\_MEM != 0) && (rd\_MEM == rs1)) && (rd\_WB == rs1) && !NoFwd\_EX) begin ForwardA = 2'b01; end  else ForwardA = 2'b00;    if (RegWrite\_MEM && (rd\_MEM != 0) && (rd\_MEM == rs2) && !NoFwd\_EX) begin ForwardB = 2'b10; end  else if (RegWrite\_WB && (rd\_WB != 0) && !(RegWrite\_MEM && (rd\_MEM != 0) && (rd\_MEM == rs2)) && (rd\_WB == rs2) && !NoFwd\_EX) begin ForwardB = 2'b01; end  else ForwardB = 2'b00; |

Above, a value of (00) implies that the data comes from the register file (standard). A value of (01) indicates that the data comes from the writeback stage of the pipeline. A value of (10) indicates that the data comes from a prior result in the ALU from the EX/MEM register.

There are two steps for the new forwarding unit. First, we need to implement the same logic as in the box above. Then, we need to take the value of the first step and AND the values with a “Data Hazard” flag that confirms that the current instruction is indeed vulnerable to a data hazard.

## Stalling

To solve for the case of a LOAD from memory, we will need to stall the pipeline until the instruction is finished. To perform the stalling, we will use a stalling unit, and have another flag for the load instruction that tells the unit that it should proceed with the stalling.

|  |
| --- |
| Check if:  ID.MemRead; EX.MemRead;  Check if risk and rs matching with rd |

Stalling is also required for the BRANCH, JALR and JAL instructions.

**When should the stalling stop? (i.e., in which step has the instruction updated the pc value)**

The writeback for the pc value happens at the EX-stage, where the branch of the ALU is checked and the immediate gets added to the pc value. This could be shortened by moving the logic for the pc value to the ID stage.

To move the pc logic, we will need to take care of the JALR and JAL (which perform pc <- rs1 + imm and pc <- pc + imm\*2) and the BRANCH (which performs register comparisons). To move the JALR and JAL, we just implement the pc adder for both operations in the ID stage. To move the BRANCH, we move the flag and comparison to the output of the register file.

**! Be careful with data hazards.** Since the comparisons now occur at the ID stage, the forwarding logic implemented for the EX stage is not going to work. We will need to implement another forwarding unit to the comparator and adder for the JALR instruction.

Moving the forwarding logic earlier in our design implies that we also need to check for forwarding in the EX stage, but it is not a great problem, since we can implement the full forwarding in the ID stage.

**Assume Branch Not Taken.** Continue sequentially, if the branch is taken, Flush ID and IF.

# Register File

## Register

Below is the code for the register unit, using Verilog parameters to make it more reusable. It has a reset and an enable signal. It is negative edge triggered.

Graphical user interface, text, application, chat or text message

Description automatically generated

Diagram

Description automatically generated

## Register File

The code below is the register file. It uses a generate and a for loop to create the 32 registers needed for the implementation.

Text

Description automatically generated

## Immediate Generator

A picture containing text

Description automatically generated

# ALU

## ALU Operations Encoding

Text

Description automatically generatedFor the ALU, we need to redesign the way the control system works. The following table shows the different operations that the ALU needs to perform:

|  |  |
| --- | --- |
| Bitwise AND (&) | 0000 |
| Bitwise OR (|) | 0001 |
| ADD | 0010 |
| SUB | 0011 |
| XOR | 0100 |
| Shift Left | 0101 |
| Shift Right | 0110 |
| Arithmetic Shift Right | 0111 |
| Less than | 1000 |
| Less than unsigned | 1001 |

For this control signals going into the ALU, we will specify **ALUOp**.

## Comparison operators encoding

For the comparison operators, we will establish a control signal **Flag** to tell the comparison unit which flag it should output. To do this we follow the following table:

|  |  |
| --- | --- |
| Equality == | 000 |
| Not Equality != | 001 |
| Less than | 010 |
| Less than unsigned | 011 |
| Greater than | 100 |
| Greater than unsigned | 101 |

The code below is the Verilog implementation of the comparison unit.

Text

Description automatically generated

## Memory Length encoding

For the memory, there are different types of memory loads and stores, varying in the length of the memory loaded/stored and the way the load it is interpreted. The following table shows the encoding of the different instruction types and the corresponding **MemLen** flag.

|  |  |
| --- | --- |
| Byte | 000 |
| Half-word | 001 |
| Word | 010 |
| Byte unsigned | 011 |
| Half-word unsigned | 100 |
| Word unsigned | 101 |

# Instruction Memory

The code below is the instruction memory.

Graphical user interface, text, application

Description automatically generated

Some notes on stalling and forwarding. The PC register should be disabled when there is a MemStall signal high. Also, since there are two instructions in the pipeline before the MemStall is detected, the IF/ID register should also be disabled. This way both PC and IF/ID register keep the value and the load instruction can move through the pipeline. The IF/ID register should be reset when there is a Branch flag and the CompFlag is also high. This indicates that the Branch is TAKEN and the contents of the register are no longer valid (since we are assuming branch not taken). This is the only register that requires getting wiped out, since the PC gets updated with the Branch and the detection happens in the ID stage.

What about Jumps? (JALR and JAL) Jumps will require to reset the value of the IF/ID register (to “flush” the instructions). PC gets updated and everything else stays the same.