Hazard

Data hazard:

A data hazard occurs when instructions in a pipeline depend on the results of previous instructions that have not yet been completed.

Types of data hazards:

1. RAW (read after write)

For example:

Instr 1: R1 = R2 + R3

Instr 2: R4 = R1 + R5

Instruction 2 needs the result of instruction 1. Therefore, instruction 2 can only be executed after R1 is written back to register.

**Without memory:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| clk\_cycle | 1 | 2 | 3 | 4 |
| intruction1 | IF | ID | EX |  |
| intruction2 |  | IF | ID | EX |

**Two solutions:**

1. regfile in the instr\_decode module

forwarding, because instruction reading of instr2 happened in decode cycle, there is no time to write back the result of instr1 into regfile

1. regfile in the instr\_execution module

no forwarding, because there is no data hazard here.

For a 3-stage pipeline, this solution is not that efficient if processing bjp instructions in decode stage

**With memory**

Memory access takes one clock cycle.

For example:

Instr1: ld R1, 1

Instr2: R2=R1+R3

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| clk\_cycle | 1 | 2 | 3 | 4 |
| intruction1 | IF | ID | EX | MEM\_ACCESS |
| intruction2 |  | IF | ID | EX |

**Two solutions:**

1. forwarding
2. stall

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| clk\_cycle | 1 | 2 | 3 | 4 |  |
| intruction1 | IF | ID | EX | MEM\_ACCESS |  |
| intruction2 |  | IF | ID | STALL | EX |

1. WAR(write after read)

~~Without memory:~~

Lone pipeline:

For example:

1: mul x3, x4, x5 // 慢指令，要多周期执行

2: add x4, x1, x2 // 快指令，写回 x4

Save the result before the next instr write back to regfile

1. WAW(write after write)

~~Without memory:~~

Lone pipeline:

For example:

1: lw x10, 0(x5) // LOAD → 走 memory，进 OITF

2: addi x10, x10, 1 // 写 x10，可能更快执行

stall

1. ~~RAR~~

Common longpipe instruction groups:

LOAD instructions → lw, lh, lb, etc.

CSR instructions → csrrw, csrrs, etc

Multiply/Divide → mul, div, rem, etc.

Instructions that can trigger exceptions

Fence and system control → fence, fence.i

Not longpipe:

ALU ops like add, sub, sll, srl

Immediate ALU ops: addi, andi, ori

Branches: beq, bne, jal, jalr

Control hazard:

A control hazard (also called a branch hazard) occurs when the pipeline makes decisions based on the result of a branch instruction (e.g., jump, if-statement), and it's unclear which instruction should be executed next.

How to solve this problem:

1. Stalling

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| clk\_cycle | 1 | 2 | 3 | 4 | 5 | 6 |
| intruction1:beq, instr2 | IF | ID | EX |  |  |  |
| intruction2 |  |  |  | IF | ID | EX |
| intruction3 |  |  |  |  | IF | ID |
| intruction4 |  |  |  |  |  |  |
| intruction5 |  |  |  |  |  |  |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| clk\_cycle | 1 | 2 | 3 | 4 | 5 | 6 |
| intruction1:beq, instr2 | IF | ID | EX |  |  |  |
| intruction2 |  |  |  | IF | ID | EX |
| intruction3 |  |  |  |  | IF | ID |
| intruction4 |  |  |  |  |  |  |
| intruction5 |  |  |  |  |  |  |

1. branch prediction & flushing

right prediction

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| clk\_cycle | 1 | 2 | 3 | 4 |  |
| intruction1:beq, instr2 | IF | ID | EX |  |  |
| intruction2 |  | IF | ID | EX |  |
| intruction3 |  |  | IF | ID | EX |
| intruction4 |  |  |  |  |  |
| intruction5 |  |  |  |  |  |

wrong prediction:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| clk\_cycle | 1 | 2 | 3 | 4 | 5 | 6 |
| intruction1:beq, instr5 | IF | ID | EX |  |  |  |
| intruction2 |  | ~~IF~~ | ~~ID~~ |  |  |  |
| intruction3 |  |  | ~~IF~~ |  |  |  |
| intruction4 |  |  |  |  |  |  |
| intruction5 |  |  |  | IF | ID | EX |

* 1. How to improve this solution

**Processing the Branch in Decode (ID) Stage**

Normally, in a simple 3-stage CPU:

Branch decision is made in the EX stage. That means by the time you know whether to take the branch, you’ve already fetched 1–2 instructions that might need to be flushed.

By moving the branch logic earlier (into ID), you:

Detect and resolve the branch 1 cycle earlier.

Can potentially fetch the correct next instruction in the next cycle, saving bubbles

What should I do to implement this solution?

1. This means the register file must be **read during decode**
2. **Branch Decision Logic in ID**
3. If a previous instruction writes to a register used in the branch, you may have a **data hazard** on the branch operands.

For example

add x1, x2, x3

beq x1, x4, target