Chapter 4b - Pipeline Multiple Instructions

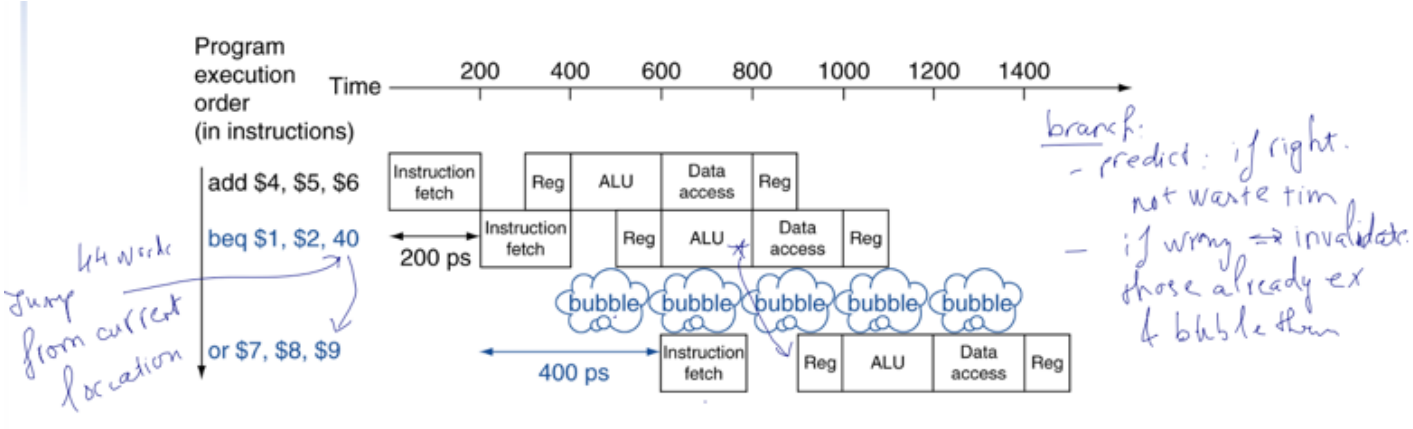
1. Pipeline

1.1. Summary

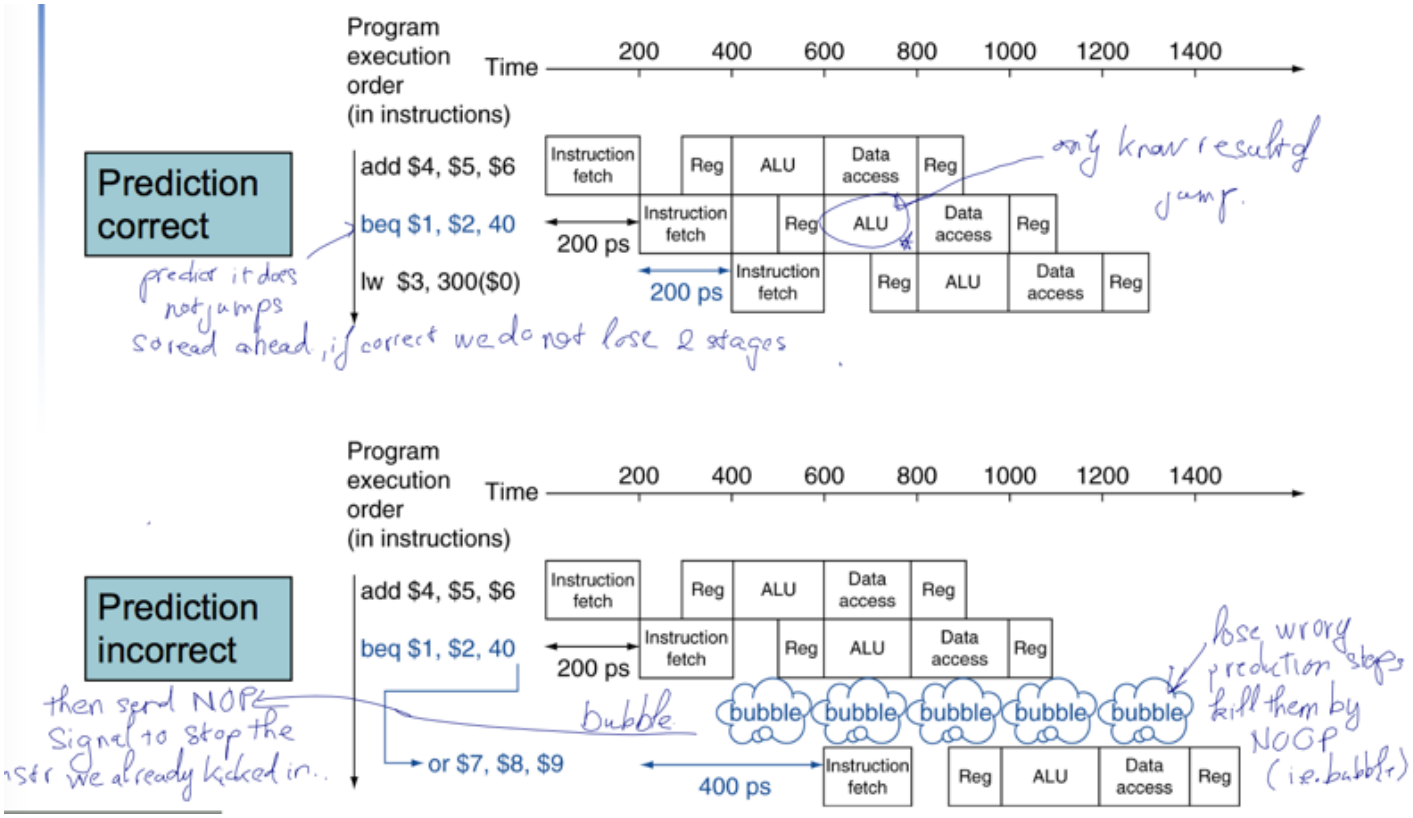
- Pipeline improves performance, increase throughput, reduce average latency (but same abs latency). But it also depends on structure hazards, data hazards, and control hazards.
- Pipeline implementation varies base on ISA design.
- Pipeline datapatch can be written in 2 ways. They are identical.
  - Logical: `IF->ID->EX->MEM->WB`
  - Physical: `IM->REG->ALU->DM->REG`

1.2. Possible Pipeline actions on Branch

- **Stall on branch**: Wait until branch outcome is determined before fetching next instructions.



- **Prediction on branch**: predict one possible outcome (could be base on past records). If it turns out correct, fine; if not, invalidate it by bubbles (noop).

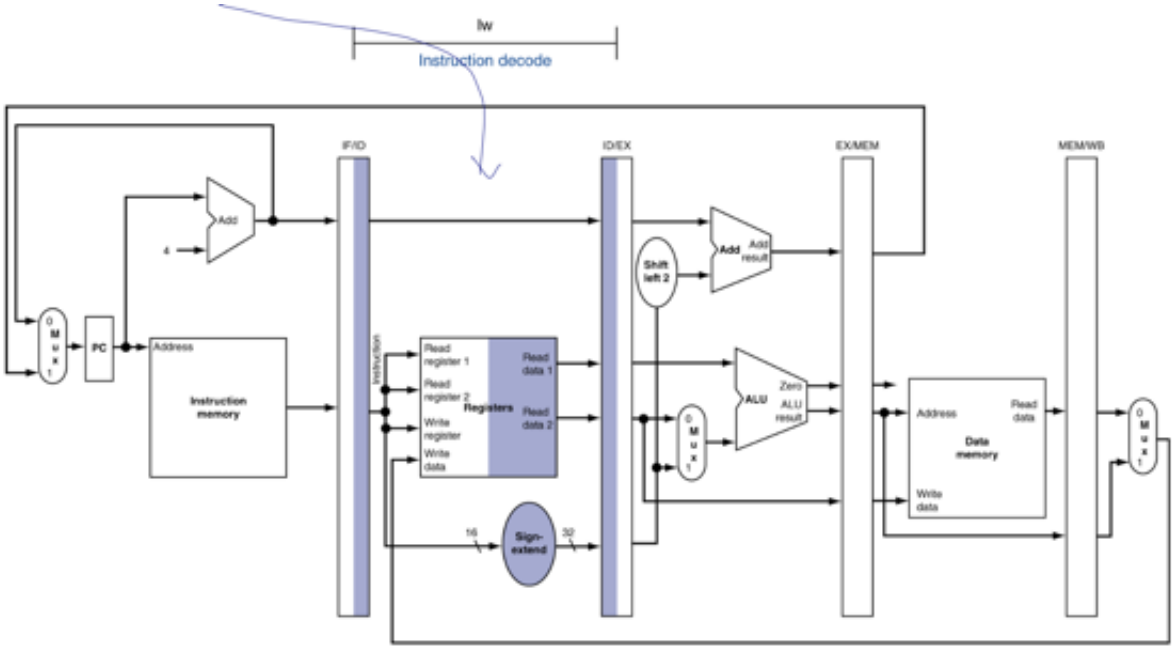


1.3. Pipeline Registers

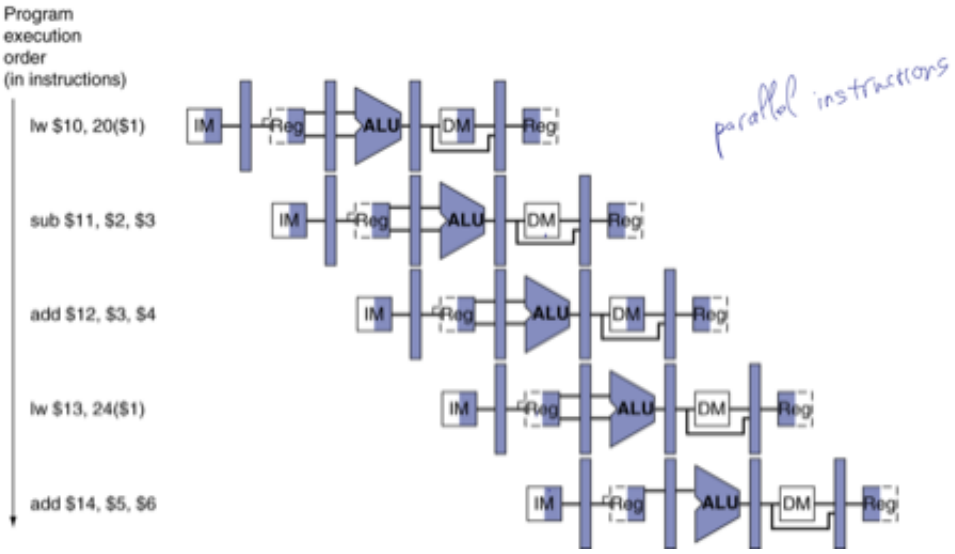
Between each stages (IF, ID, EX, MEM, WB) there is a pipe that stores **pipeline registers**, which is used to hold result from previous cycle. Pipeline uses pipeline control to control signals derived from instruction in each cycle.

There are 2 types of diagram used to describe pipeline datapath: **"single clock cycle" pipeline diagram, and "multi clock cycle" diagram**.

Single clock cycle diagram



Multi-clock cycle (or multicycle) diagram



2. Hazards

2.1. Structural Hazards

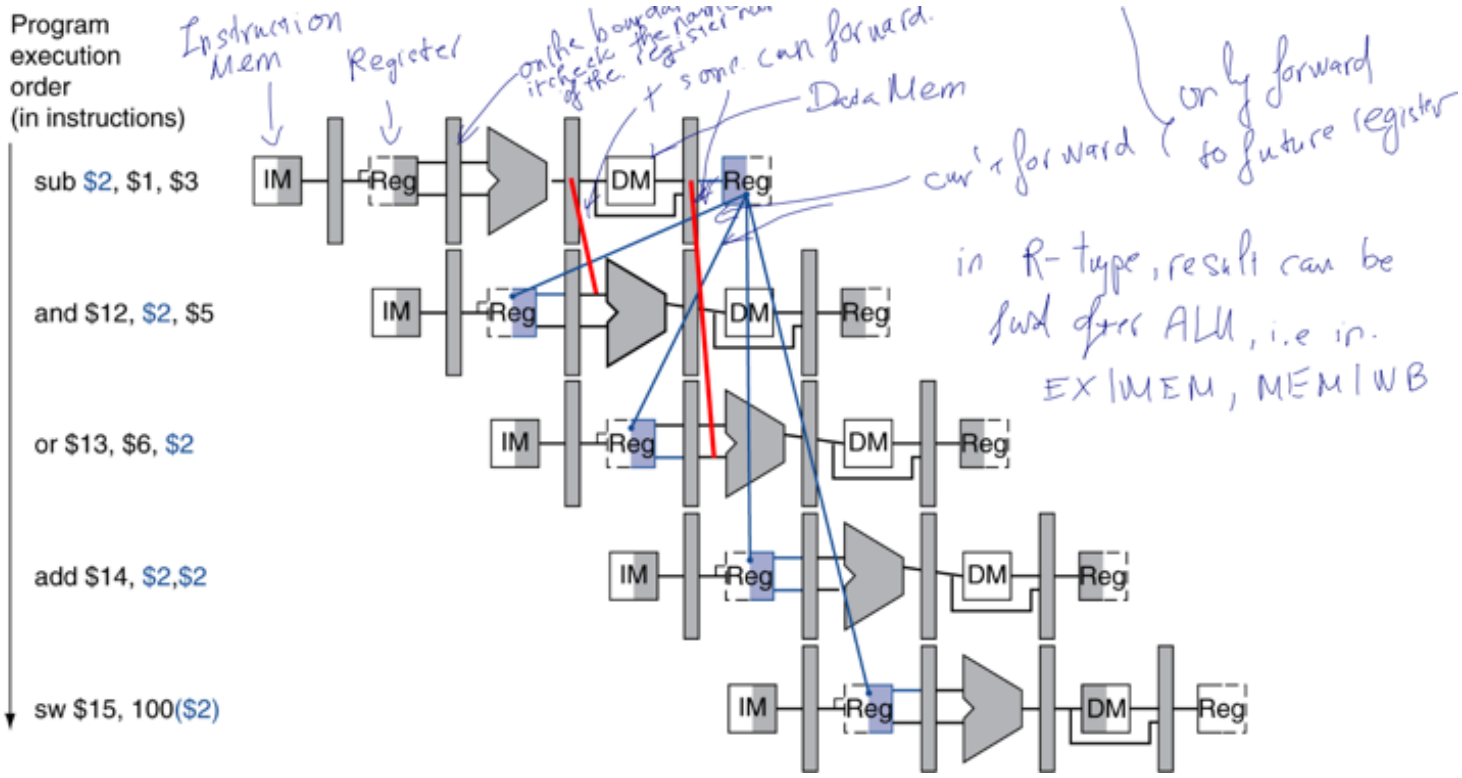
- Required resource is busy (IO, load, store)
- Stalling instruction fetch for that cycle cause pipeline buble
- Fixed by writing to register in first half of clock, then reading from register during second half.

2.2. Data Hazards

- Need to wait for previous instruction to complete its data read/write.
- Forward result from one stage to another (although load will still cause stall)

2.2.1. Data Required In ALU Stage

"Newer" instruction require data available in registers before going through ALU. In some cases it can be resolved by **Forwarding (aka ByPassing)** data from older instruction.



2.2.1.1. Forward Conditions for R-Type

> R-Type returns the data **right after ALU**, i.e. result in EX|MEM or MEM|WB.

Forward data from old EX|MEM to new ID|EX

```
old EX|MEM.RegisterRd == new ID|EX.RegisterRs
old EX|MEM.RegisterRd == new ID|EX.RegisterRt
```

Forward data from old MEM|WB to new ID|EX

```
old MEM|WB.RegisterRd == new ID|EX.RegisterRs
old MEM|WB.RegisterRd == new ID|EX.RegisterRt
```

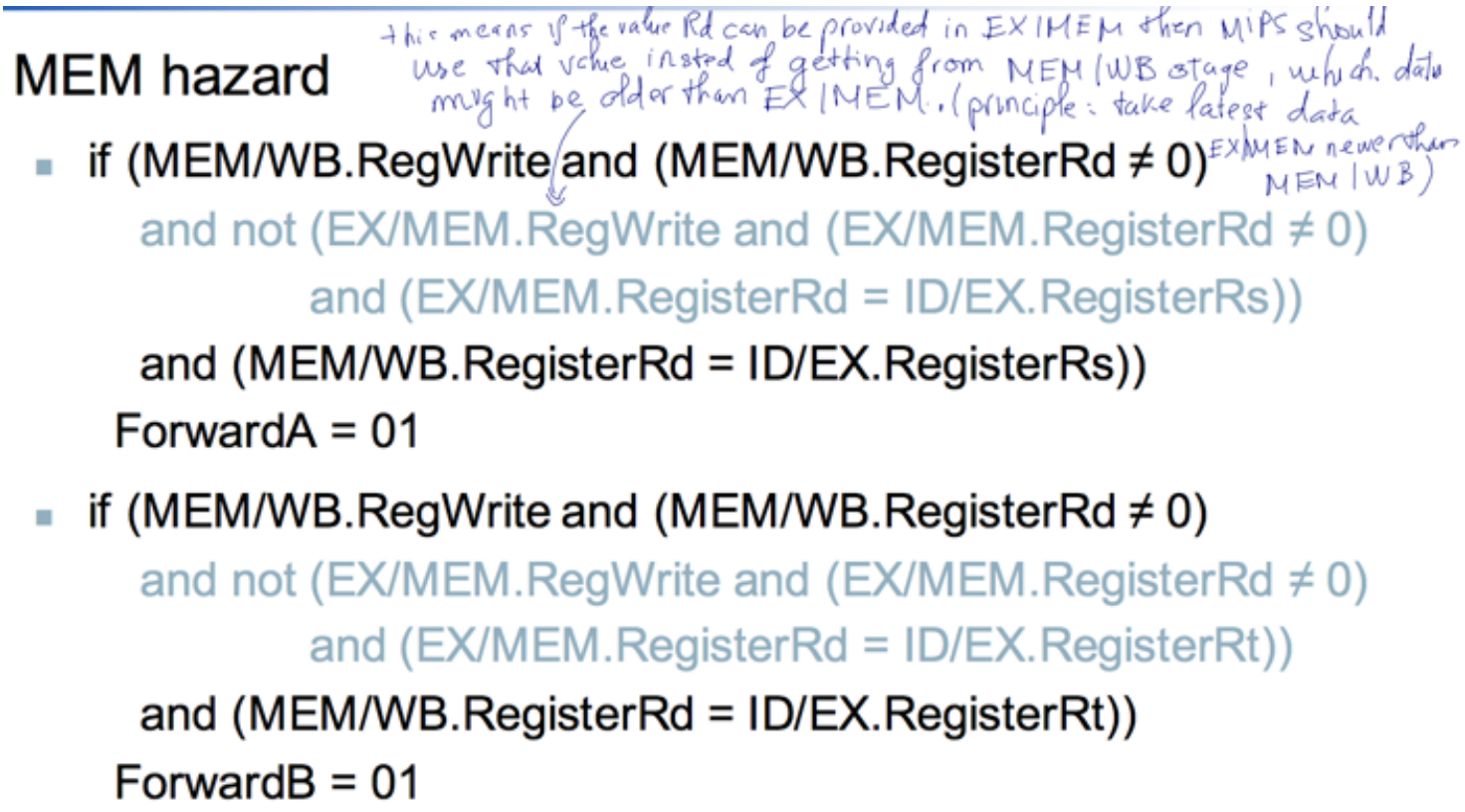
AND the old instruction outputs to registers

```
EX|MEM.RegWrite, MEM|WB.RegWrite
```

AND if the control signal (or output) is not writing to to $zero

```
EX|MEM.RegisterRd ≠ 0
MEM|WB.RegisterRD ≠ 0
```

In Combination:



2.2.1.2. Stall Conditions for Load (I-type)

> Stall aka ~~Bubble~~
>
> Load only returns the data **right after MEM**, i.e. result appearing in MEM|WB only. Thus to work with Load, we need to detect earlier at **ID|EX** (for old instr) and **IF|ID** (for new instr).

If this condition meets, **stall and insert buble**.

IF ID EX MEM WB

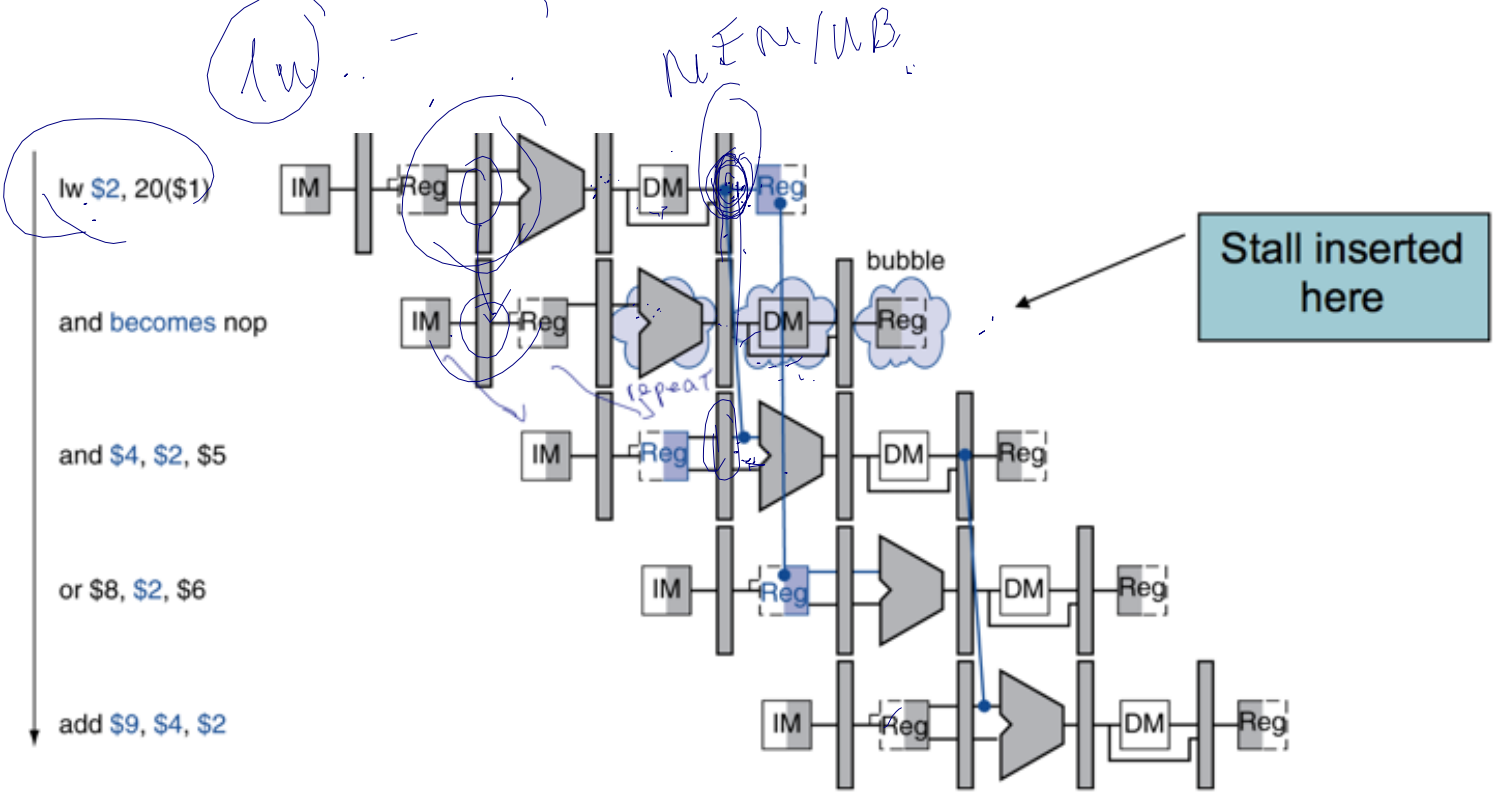
```
Old ID|EX.MemRead AND
    ((ID|EX.RegisterRt == IF|ID.RegisterRs) OR
    (ID|EX.RegisterRt == IF|ID.RegisterRt)
    )
```

How to stall

following ID|EX

> Set control values in ~~old instr ID|EX register~~ to 0 (bubble/no-op in EX, MEM, WB)
>
> Prevent updating PC and IF/ID registers in new instr. This result in refetch and re-decode in new instr.

or reorder it.

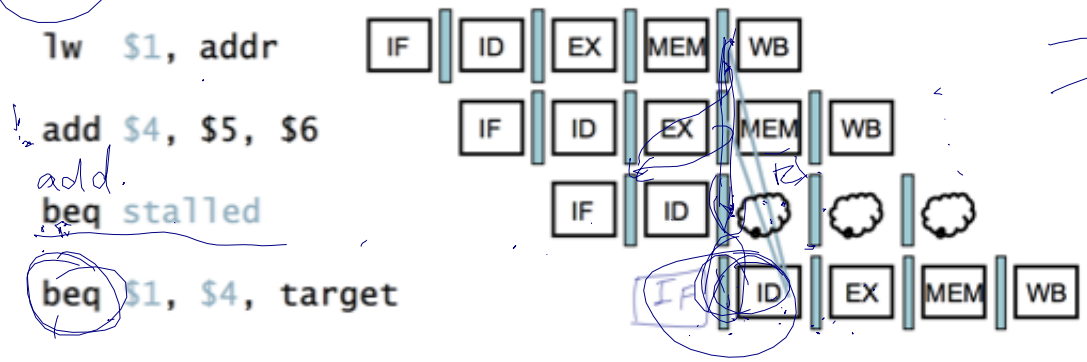


Stall characteristics: stall reduces performance, but essential to get correct results.

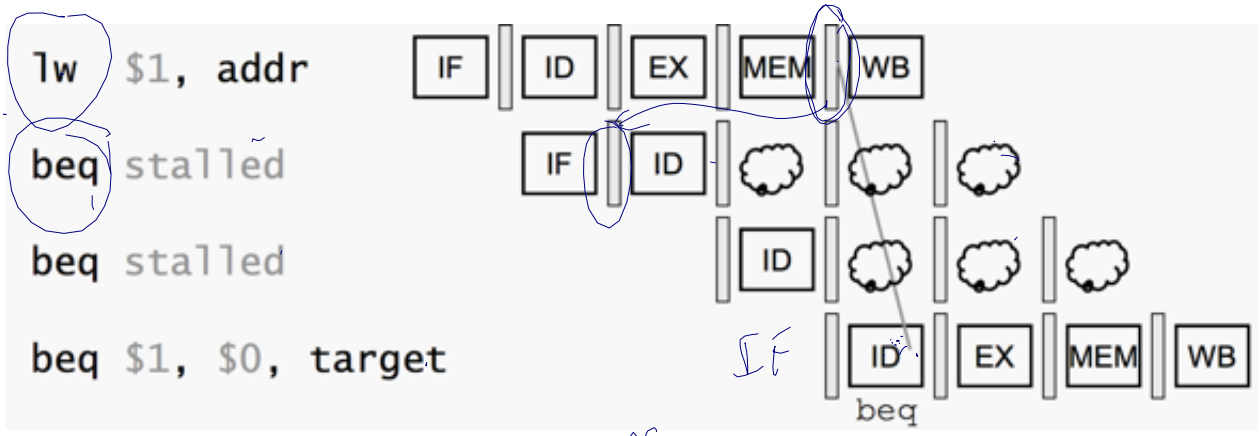
2.2.2. Data Required in ID stage

> This happens for **Branch as the newer instruction**.

- If a comparison register in beq is a destination of 2nd or 3rd preceding ALU instruction, **it needs 1 stall cycle** (BEQ needs data at ID, not EX).



- If a comparison register is a destination of immediately preceding load instruction, **it need 2 stall cycle**.



Stall more if need more steps

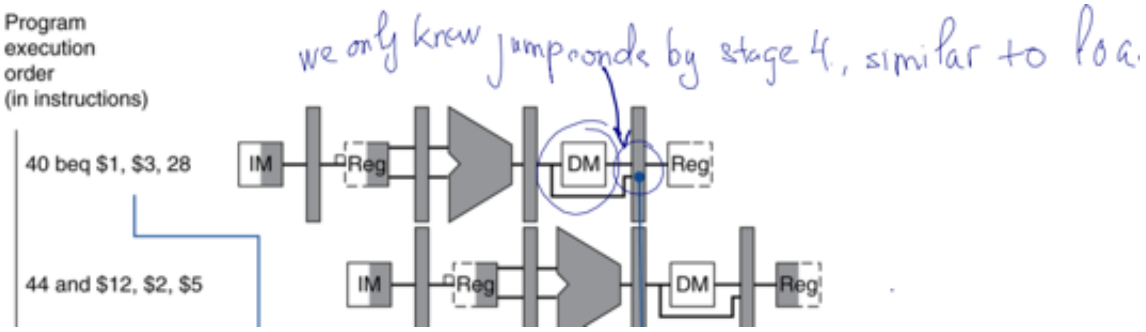
2.3. Control Hazard

Fetching next instruction depends on branch outcome

Options are move branch comparator to stage 2, predict branch outcome, and flush pipeline if wrong

2.3.1. Stall in Branch

> Similar to load, if branch is the preceding instruction, **Branch** outcome only **determined in MEM**, i.e. MEMIWB, DMIREG



2.3.2. Dynamic Branch Prediction

- Using prediction buffer, aka branch history table.
- Stores outcome, if wrong, flush pipeline and flip prediction.

2.3.2.1. 1-bit Predictor

Will waste 2 times if wrong.

2.3.2.2. 2-bit Predictor

Only change prediction on 2 successive mispredictions.

3. Exceptions and Interrupt

3.1. Exception

- **Exception**: unexpected events require change in flow of control, usually arise within the CPU in opposed to Interrupt arise from ext IO controller.
- **Handling**:
  - Use **EPC (Exception Program Counter)** to save the Exception instruction address.
  - Use **Cause Register** to save Exception reason.
  - Use **Rescue** register to jump to rescue handler at `800000180`
- Alternative mechanism
  - Use vectored Interrupts, labeling them with specified opcode.
- **Handler actions**:
  - Find the cause
  - Find next action
    - if restartable: take corrective action and use EPC to return to program.
    - otherise: terminate program, report error
- If multiple exceptions: deal with earliest instruction