Pipeline Control Hazards

Caiwen Ding
Department of Computer Science and Engineering
University of Connecticut

CSE3666: Introduction to Computer Architecture

Admin

- We will continue to learn processor pipeline. After reviewing data hazards, we study control hazards.

Control Hazards

- When the flow of instruction is not sequential (i.e., the next instruction is not located at PC + 4), processor has to wait for PC
  - Dependence through PC: PC is needed in IF, but produced in later stages

PC+offset
PC+4 ?

- Due to change of instruction flow
  - Branches (e.g., `beq`)
  - Jumps (e.g., `jal`)
  - Exceptions

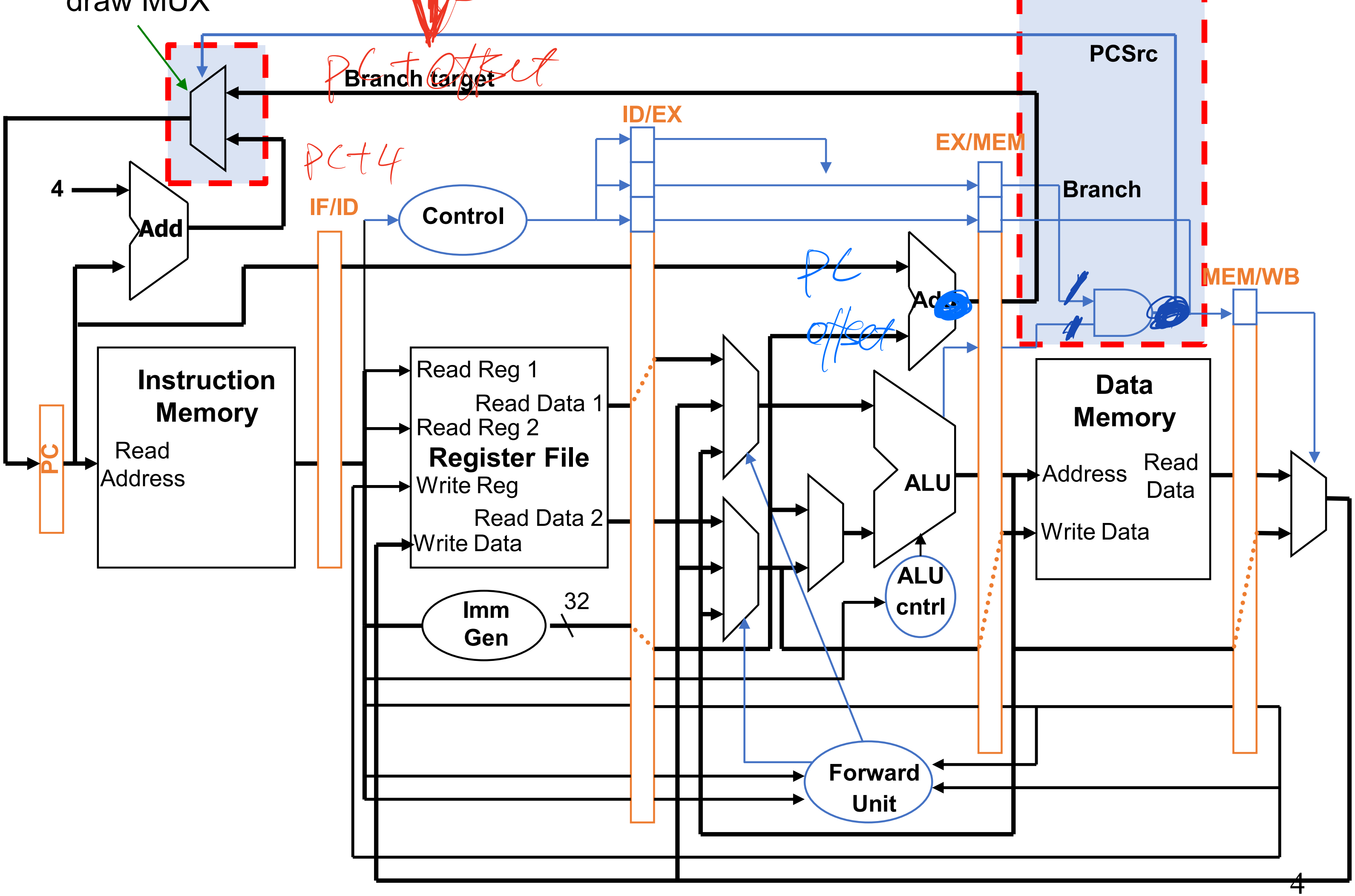
Reading: Sections 4.6 (on control hazards) and 4.9.

Pipeline with Branch

ID

MEM

Another way to draw MUX

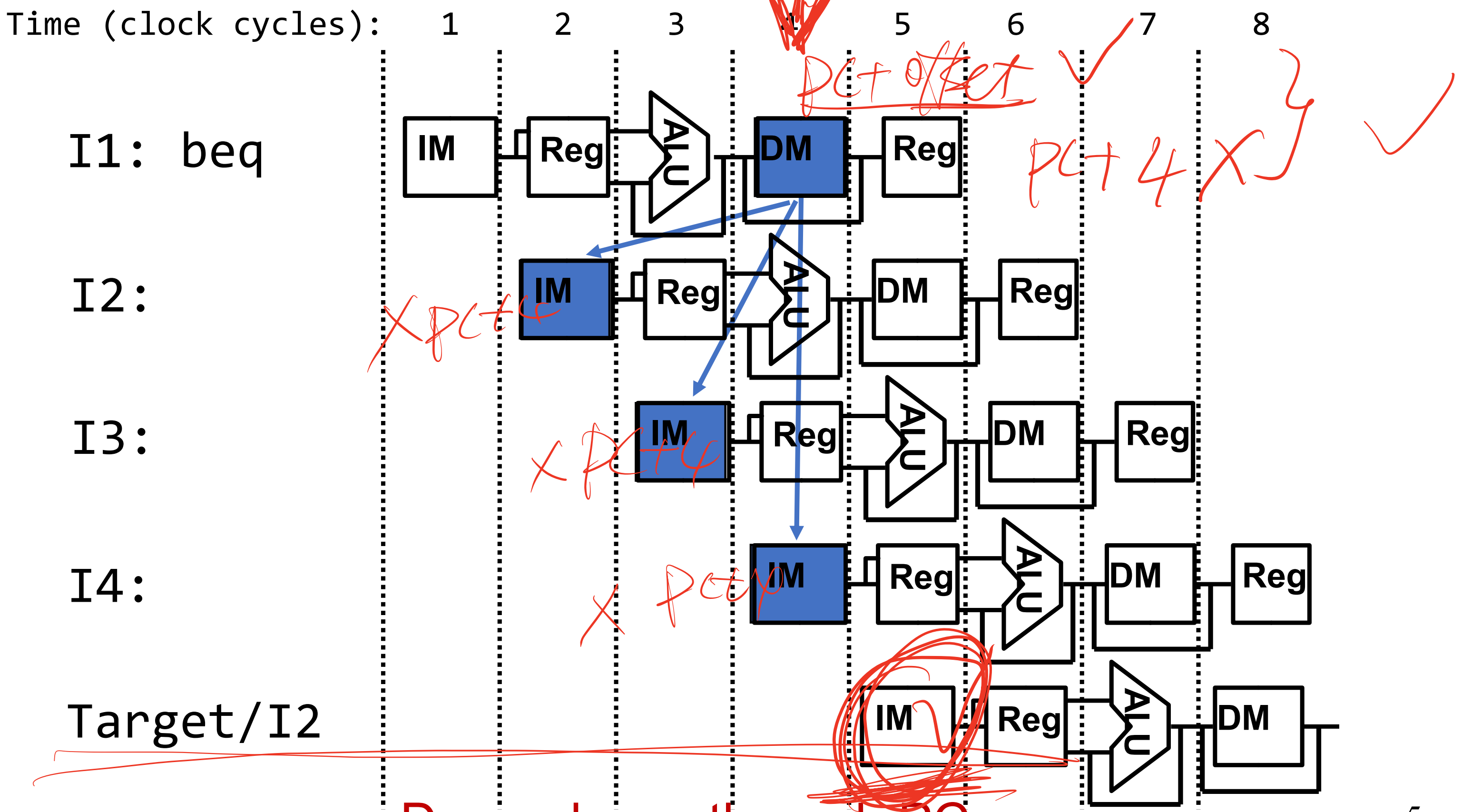


Control Hazards: Branch Instructions

BEQ is resolved in MEM. The correct instruction is fetched when it is in WB

“A branch is resolved” means the processor is certain about which instruction to execute next

Need to know branch target address and condition specified in the branch instr.



Dependence through PC

Dealing with Control Hazard

- Good news: Control hazards occur less frequently than data hazards
- Bad news: There is nothing as effective against control hazards as forwarding is for data hazards

- Possible approaches
  - Stall (impacts CPI)
  - Always predict not taken (static prediction) NT
  - Move decision point as early in the pipeline as possible, thereby reducing the number of stall cycles
  - Predict with better strategy and hope for the best !

Control hazards: Branch Resolved in MEM

Processor waits until branch is resolved, in MEM (remember the AND gate?)

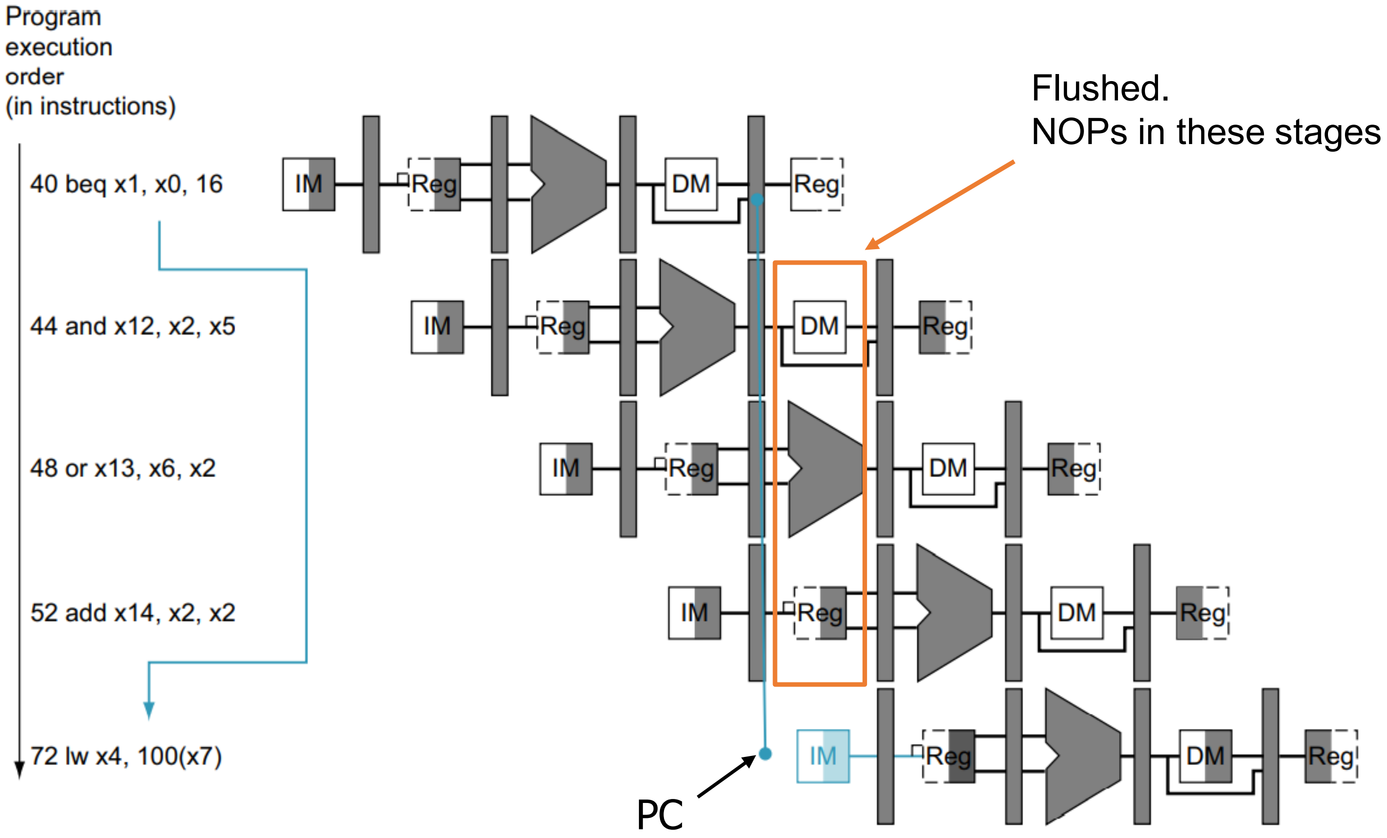
In Cycle 4, I2, I3, and I4 are in the pipeline. They are flushed in Cycle 5.
If the branch is not taken, they are fetched again.

Pipeline stalls for 3 cycles for each branch.

	1	2	3	4	5	6	7	8
`I1:beq x1,x0,I9`	IF	ID	EX	MEM	WB			
`I2:~~and~~ x12,x2,x5`		IF	ID	EX	× NOP			
`I3:~~or~~ x13,x6,x2`			IF	ID	× NOP			
`I4:~~add~~ x14,x2,x2`				IF	× NOP			
`I9 or I2`					IF			

Another view

- Three instructions following BEQ are in the pipeline
- When BEQ is resolved, they are flushed



Flushed.
NOPs in these stages

Two "Types" of Stalls

Discussed in load-use & control hazard

- NOP (or bubble) is inserted between two instructions in the pipeline (as done for load-use situations)
  - Keep the instructions in earlier stages of the pipeline (later in the code) from progressing down the pipeline for a cycle
  - Insert NOP by zeroing control bits in the pipeline register at the appropriate stage
  - Let the instructions in later stages of the pipeline (earlier in the code) progress normally down the pipeline
- Flushes (or instruction squashing) are that one or more instructions in the pipeline are replaced with NOPs
  - Zero the control bits for the instruction to be flushed
  - NOP, instead of instruction from I-Mem, is stored in IF/ID

Performance of Branch: waiting

0.2 overhead

load-use

The average CPI without control hazards is 1.2.

Suppose 25% of the instructions executed are branches.

The processor waits 3 cycles for each branch.

$0.25 \times 3 = 0.75$

What is the average CPI if control hazards are included?

$1.2 + 0.75 = 1.95$

How much faster would the processor be if there are no control hazards?

Round your answer to the nearest tenth.

$\frac{1.95}{1.2} =$

Performance Impact of Branch

CPI overhead (stall cycles) due to branch:

Frequency of branch × Number of stall cycles per branch

0.25 × 3

Reduce the penalty for each branch

Compute the branch target address and evaluate the branch condition early

Moving Decision point from MEM
3→1 to ID

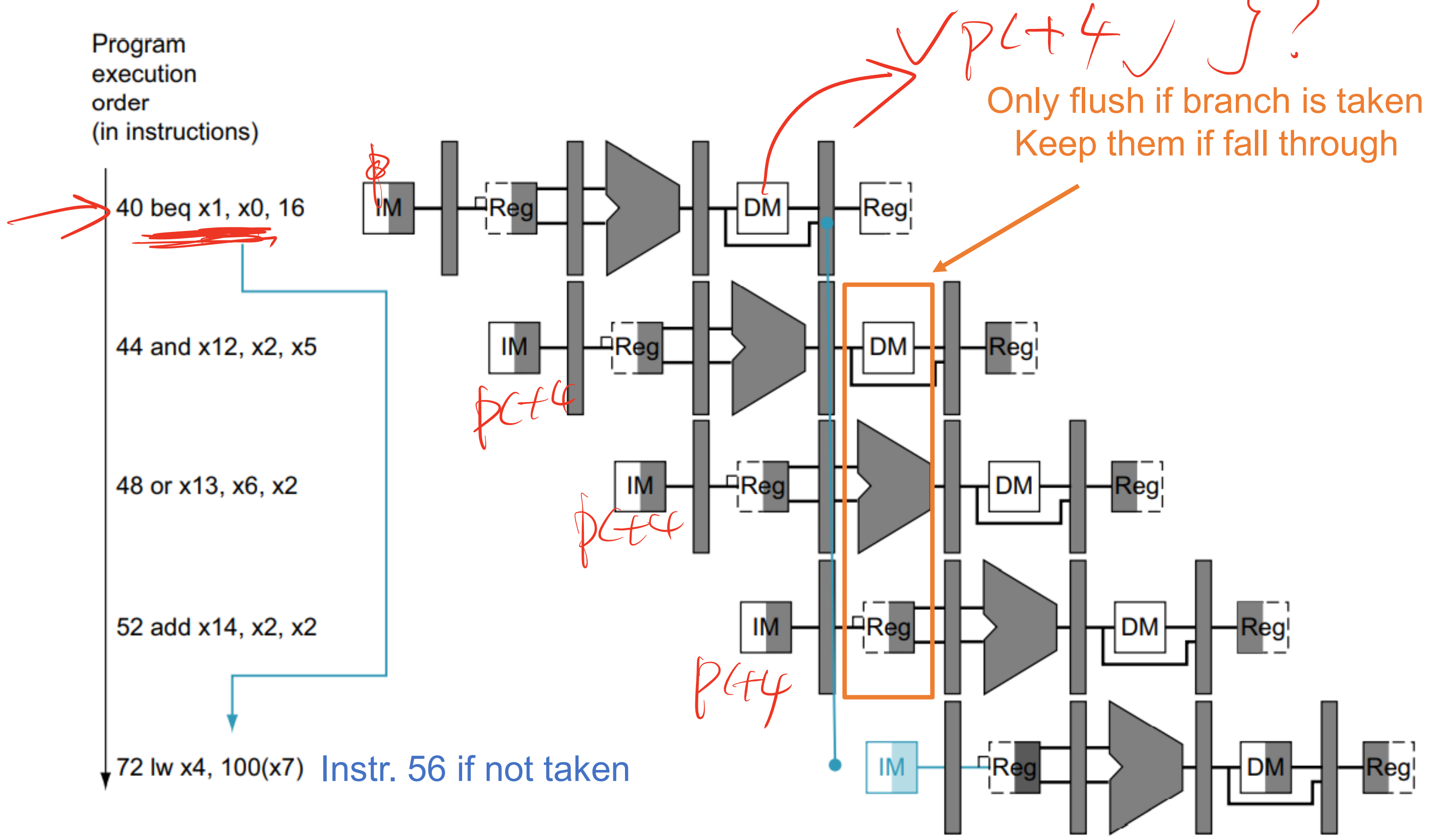
Reduce the number of times of paying penalty

Branch prediction

(Always) Predict branch is not taken

- Continue to fetch instructions as if there was no branch
- Keep the instructions in the pipeline if the branch is not taken

pc+offset



Only flush if branch is taken
Keep them if fall through

Instr. 56 if not taken

Static Branch Prediction: Predict Not Taken

- Predict not taken
  - Continue to fetch from the sequential instruction stream
  - If prediction is correct, keep the instructions and no penalty
  - If prediction is wrong, 3 instructions are flushed (→ 3 stall cycles)

Not flushed if branch is not taken

If Taken

PC+offset

	1	2	3	4	5	6	7	8
I1:beq x1,x0,I9	IF	ID	EX	MEM	WB			
I2:and x12,x2,x5		PC+4 IF	ID	EX	MEM			
I3:or x13,x6,x2			PC+4 IF	ID	EX			
I4:add x14,x2,x2				PC+4 IF	ID			
I5:					IF			

Question: Predict Not Taken Performance

Assume:

branch is resolved in MEM and processor always predicts not taken

25% of the instructions executed are branches.

30% of the branches are taken.

What is the CPI overhead due to control hazards from branches?

$0.25 \times [30\% \times 3] = 0.225$

0.21

Reduce the number of stall cycles per branch

3 cycles penalty seems to be high. Can we reduce that?

Branch Resolved in { MEM ☆☆☆
ID ☆☆

Why 3 cycles? Because the branch is resolved in MEM

Resolving branch in earlier pipeline stages reduces stall cycles

The earlier, the better!

The earliest is ID stage, where we know it is a branch and we read registers

	1	2	3	4	5	6	7	8
`I1:beq x1,x0,I9`	IF	ID	EX	MEM	WB			
`I2:and x12,x2,x5`		IF	ID	EX				
`I3:or  x13,x6,x2`			IF	ID				
`I4:add x14,x2,x2`				IF				
`I9 or I2`					IF			

Looping structures

- We learned two structures for loop
- Which one works better with predict not taken?

```
Loop: beq x1,x2,Out   1   ✓
      1st loop instr
      2nd loop instr
          .
          .
          .
      n-th loop instr
      beq  x0, x0, Loop   2   ?
Out:  fall through instr
```

```
Loop: 1st loop instr
      2nd loop instr
          .
          .
          .
      n-th loop instr
      bne x1,x2,Loop   3   ?
      fall through instr
```

Looping structures on "Predict not taken"

- Predict not taken works well for branching at "top of the loop"
  - But such loops have "jumps" at the bottom of the loop to return to the top of the loop and incur stalls

- Predict not taken does NOT work well for branching at "bottom of the loop"
  - The branch is taken most times
  - Fall through instruction is always flushed

Can we do better?

Yes. learn it in Advanced Comp Arch.

Need better prediction!

- A simple static prediction scheme will hurt performance
  - The behavior of branches are different

- In deeper pipelines, branch penalty is more significant
  - Branch decisions cannot be made early, e.g., in the second stage

- In superscalar processors, processors execute many instructions a cycle. Branches appear more frequently (per cycle)
  - Also, a stall cycle is an opportunity to execute multiple instructions

Why dynamic branch prediction?

Need better prediction!

NT → T → T → NT

- Advance in technology made it possible to predict branch behavior dynamically with hardware
  - We can put more transistors in the processor !
  - Use runtime information for better prediction !

- We want to customize predication for each branch
  - Predict not taken for some
  - Predict taken for some

Summary of Control Hazard

- Reduce the performance impact of control hazard
  - Find out the outcome and compute the target address early
- Static branch prediction
- Resolve branches in ID stage
  - More hardware
  - Forwarding to ID stage
- Dynamic branch prediction

Study the “Check yourself” question at the end of Section 4.9

In later courses

- We will answer the following questions:
  - Does predicting taken always result in stall cycles?
  - Do we have better branch predictors?
  - How are branch predictors implemented?
  - I heard branch prediction is exploited in security attacks. Do we have secure predictors?

Summary of Pipeline

latency ↑

- All modern processors use pipelining for performance

Throughput ↑ XXXX

- Pipeline clock rate limited by slowest pipeline stage – so designing a balanced pipeline is important
- Must detect and resolve hazards
  - Structural hazards – designing the pipeline correctly
  - Data hazards
    - Stall (impacts CPI)
    - Forward (requires hardware support) ✓

RAW (Forward)
Load-use (Forward +1 CC)
MEM-MEM (Forward)

  - Control hazards – put the branch decision hardware in as early a stage in the pipeline as possible
    - Stall (impacts CPI)
    - Static and dynamic prediction (requires hardware support)
- Pipelining complicates exception handling

X X X

Single-cycle-processor

Explanation of 1-bit predictor example

A 1-bit predictor will be incorrect twice when not taken

Assume predict_bit = 0 to start (indicating branch not taken)

1. First time, the predictor mispredicts the branch since the branch is taken back to the top of the loop; invert prediction bit (predict_bit = 1)

2. As long as branch is taken (looping), prediction is correct

3. When exiting the loop, the predictor again mispredicts the branch since this time the branch is not taken falling out of the loop; invert prediction bit (predict_bit = 0)