Computer Architecture

Fall, 2025
Week 12
2025.11.17

組別：_____ **簽名：**_____

[group 1]

1. True or False

a. In a pipelined processor, "Forwarding" (or bypassing) can resolve all types of data hazards, completely eliminating the need for "Stalls."
b. In a pipelined design, the number of instructions that must be flushed to handle a branch hazard is fixed, regardless of which pipeline stage the branch decision is completed in.
c. "Dynamic Multiple Issue" allows instructions to execute out of order, but to ensure program correctness, the results must be committed in order.
d. In a "Static Multiple Issue" architecture, the CPU examines the instruction stream at runtime and groups instructions to be issued together.
e. Deepening the pipeline (Deeper pipeline) is the only method to increase Instruction-Level Parallelism (ILP).

Ans:

a. False.
   Forwarding can resolve most data hazards (e.g., R-Type-use), but it cannot resolve the "Load-Use Hazard." When a lw instruction fetches data from memory, the data is not available until the end of the MEM stage. A subsequent instruction that needs this data in its ID stage cannot get it in time, even with forwarding. This situation requires one "Stall" cycle (or a "Bubble") to be inserted.
b. False.
   The branch penalty—the number of instructions to be flushed—depends directly on the pipeline stage where the branch decision is made. If the decision is made in the MEM stage, 3 instructions (in the IF, ID, and EX stages) must be flushed. If the design is optimized to move branch decision hardware to the ID stage, then only 1 instruction (the one just fetched into the IF stage) needs to be flushed, significantly reducing the performance penalty.
c. True.
   The CPU allows instructions that have no dependencies (e.g., sub $s4, $s4, $t3) to execute before an earlier instruction that is stalled waiting for data (e.g., addu $t1, $t0, $t2 waiting on an lw). However, to maintain logical correctness and handle exceptions precisely, the results must be "committed" (written to registers) in the original program order (in-order).
d. False.
   In "Static Multiple Issue," the scheduling is done by the Compiler at compile-time. The compiler groups instructions into "issue packets" (as in VLIW) and manages

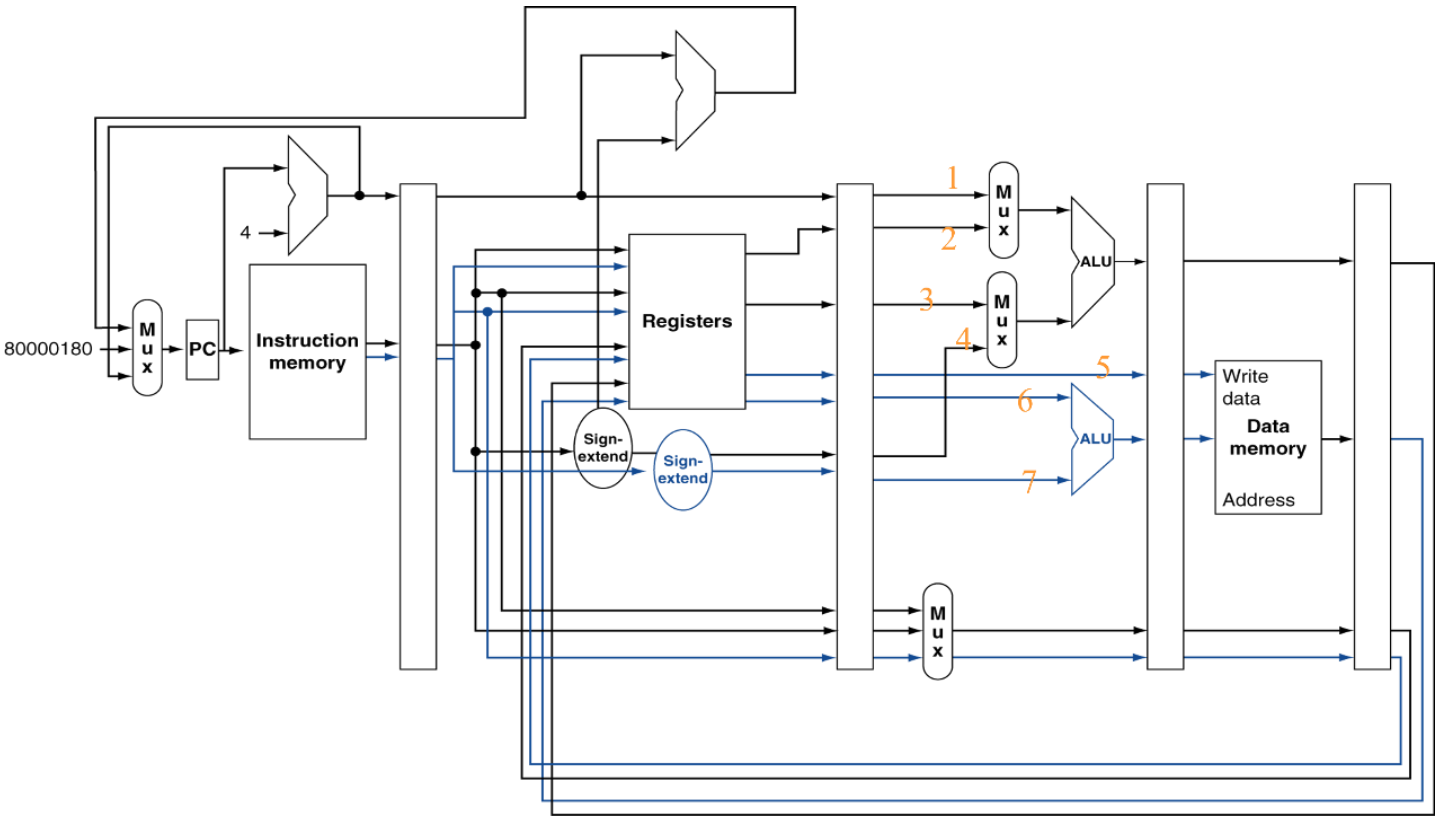
dependencies. Conversely, in "Dynamic Multiple Issue," it is the CPU (hardware) that decides which instructions to issue simultaneously at runtime.

e. False.
There are two main ways to increase ILP. The first is a "Deeper pipeline," which allows for a shorter clock cycle. The second is "Multiple issue," which allows multiple instructions to be started per clock cycle (CPI < 1).

[group 3]

2. Question:

In the following design of MIPS with Static Dual Issue, what do wire 1~7 each represent? (Note: separate 1~4 and 5~7)



Ans:

1. PC+4
2. rs
3. rt
4. immediate
5. rt of store
6. rs of store and load
7. immediate

[group 4]

3. Question:

In a standard MIPS Static Dual Issue processor, issuing two Load/Store instructions in the same packet is prohibited. What specific type of hazard would occur if this rule were violated?

Ans:

structural hazard (Because both instructions would attempt to access the single Data Memory port simultaneously during the MEM stage.)

[group 8]

4. Question:

True or False? Why?

A. In MIPS, interrupt managed by a System Control Coprocessor (CP0).

B. Handling branch does not need any hardware when it needs to flush instructions if it predicts wrong.

C. If we make a branch decision at ID, we can reduce the number of instructions that need to be flushed to a single instruction.

D. Exceptions arise within the CPU.

Ans:

A. False, exceptions managed by a System Control Coprocessor (CP0).

B. False, it needs hardware to flush instructions.

C. True.

D. True.

[group 9]

5. Question:

True or False

a. Predicting a branch as always taken requires no additional hardware for flushing incorrect instructions.
b. When the branch decision is made at the MEM stage, instructions in earlier stages such as IF/ID and ID/EX may need to be flushed.
c. Moving the branch address calculation to the ID stage helps reduce the delay caused by taken branches.
d. The control signal IF.flush is used to replace incorrect instructions in the pipeline with NOPs.
e. Compiler rescheduling and delay branch techniques are software-level methods for handling branch hazards.

Ans:

a. False; Predicting branches (always taken or not taken) requires additional hardware for flushing incorrect instructions when the prediction is wrong.

b. True; If the branch decision occurs at the MEM stage, earlier pipeline stages (IF/ID, ID/EX) must be flushed when the prediction is incorrect.

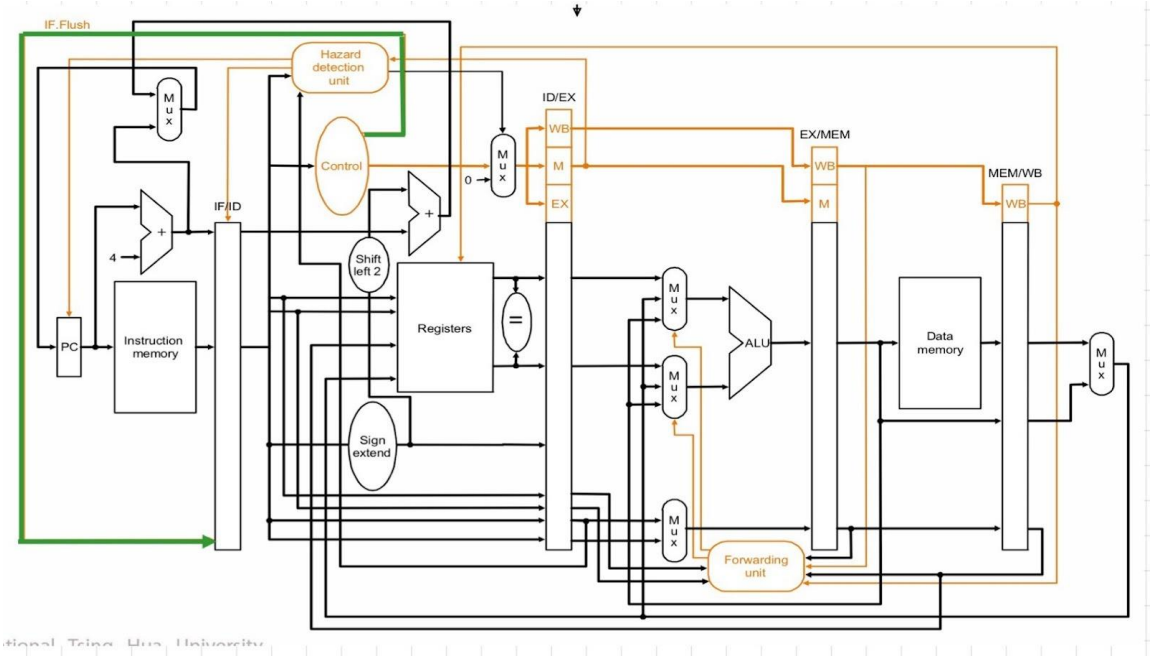
c. True; Moving branch address calculation to the ID stage lets the processor determine the branch outcome sooner, reducing delay from taken branches.

d. True; The control signal IF.Flush zeros out the instruction in IF/ID, effectively making it a NOP (no operation). (“Add a control signal, IF.Flush, to zero instruction field of IF/ID => making the instruction an NOP.”)

e. True; Compiler rescheduling and delay branch are software-level strategies that rearrange instructions to minimize branch penalties.

[group 10]

6. Question:



Using the structure above (with forwarding, hazard detection, and predict branch always not taken), how many bubbles (NOPs) will be generated after the process below?

```
        addi $11, $0, 0
        addi $10, $0, 2
loop:
        addi $11, $11, 1
          sub $2, $1, $3
          and $4, $2, $5
          or    $4, $4, $2
          add $9, $4, $2
          lw $2, 20($9)
          and $5, $2, $4
          or $4, $2, $4
        bne $11, $10, loop
exit:
          add $6, $6, 1
```

Ans:

3

The loop will run 2 iterations. In both loop, bubble will be generated after lw (since Rd of lw == Rs of the next instruction). Also, in the first loop, there is one bubble generated after bne jump to the next loop. As a result, there are 3 bubbles generated in total after the process.

[group 11]

7. Question:

Which of the following techniques are primarily handled by the compiler, by hardware, or require both?

A. Multiple issue
B. VLIW (Very Long Instruction Word)
C. Out-of-order execution

Ans:

A. Both

B. Primarily handled by the compiler(software-based)

C. Primarily handled by the hardware

[group 13]

8. Question:

Consider the following loop running on a static two-issue MIPS pipeline that has

Slot A: ALU or branch instruction
Slot B: Load or Store instruction

```
Loop:   lw    $t0, 0($t1)
        addi  $t1, $t0, 5
        sw    $t1, 0($s1)
        addi  $s1, $s1, 4
        bne   $s1, $s2, Loop
```

Reorder the instructions to minimize pipeline stalls on this two-issue processor

- You may issue one ALU/branch and one load/store per cycle.
- Assume branch prediction is correct
- Keep loop behavior identical

Show your schedule of first iteration in the following table.

Clock Cycle	ALU / branch	Load / store
1		
2		
3		
4		
5		

Ans:

Clock Cycle	ALU / branch	Load / store
1	nop	lw $t0, 0($s1)
2	nop	nop
3	addi $t1, $t0, 5	nop
4	addi $s1, $s1, 4	sw $t1, 0($s1)
5	bne $s1, $s2, Loop	nop

[group 2]

9. What is the difference between static multiple issue and dynamic multiple issue?

Ans:

由軟體(compiler)處理稱為 static multiple issue

由硬體(processor)處理稱為 dynamic multiple issue，又稱 superscalar

[group 12]

10. Question:

What is the difference between exceptions and interrupts?

Ans:

An exception arises within the CPU (e.g., overflow, undefined opcode, syscall) and is synchronous, caused by the current instruction.
An interrupt comes from an external device (e.g., keyboard, I/O controller) and is asynchronous, occurring independently of the current instruction.

[group 5]

11. Question:

Why should dynamic scheduling be done? Why not just let the compiler schedule code?

Ans:
Dynamic scheduling is necessary because the compiler (static scheduling) cannot access run-time information like variable memory latencies (cache misses) or true branch outcomes. Dynamic hardware detects dependencies on the fly and uses Out-of-Order Execution (OOE) to tolerate latency and maximize Instruction-Level Parallelism (ILP) by executing independent instructions while others stall. This dramatically improves performance over a purely statically scheduled pipeline while ensuring program order is maintained for commitment to guarantee correctness.