Computer Architecture HW 4

111062117, Hsiang-Sheng Huang

May 9, 2025

1

a

sd, ld, beq.

b

add, and, beq.

c

Must-be-0 signals	**Consequences if set to 1**
Branch	PC may jump to unwanted code segment.
MemWrite	Unneccessary memory read gives pereformance overhead.

2

a

After decoding, $015A07B3_{\text{hex}} = 0000\ 0001\ 0101\ 1010\ 0000\ 0111\ 1011\ 0011_2$.

funct7	**rs2**	**rs1**	**funct3**	**rd**	**opcode**
0000000	10101	10100	000	01111	0110011
-	x21	x20	-	x15	-

So the instruction is: ADD x15, x20, x21.

Branch	MemRead	MemtoReg	ALUOp	MemWrite	ALUSrc	RegWrite
0	0	0	10	0	0	1

b

The input values of the ALU are **Reg[20]** and **Reg[21]**.

3

a

- **add:** 20 (PC Read) + 200 (I-Mem) + 120 (Register File) + 10 (Mux) + 150 (ALU) + 10 (Mux) + 10 (Register Setup) = 520 (ps)
- **ld:** 20 (PC Read) + 200 (I-Mem) + 120 (Register File) + 150 (ALU) + 200 (D-Mem) + 10 (Mux) + 10 (Register Setup) = 710 (ps)
- **sd:** 20 (PC Read) + 200 (I-Mem) + 120 (Register File) + 150 (ALU) + 200 (D-Mem) = 690 (ps)
- **beq:** 20 (PC Read) + 200 (I-Mem) + 120 (Register File) + 10 (Mux) + 150 (ALU) + 5 (Single Gate) + 10 (PC Mux) + 10 (PC Setup) = 525 (ps)

Therefore, **add** has the least execution time at 520 ps.

b

710 ps. This is because **ld** has the longest latency at 710 ps.

4

a

- **Single-cycle:** 1 cycle per instruction. Therefore, $\text{CPI}_{\text{single}} = 1$.
- **Multi-cycle:** $\text{CPI}_{\text{multi}} = 0.5 \times 4 + 0.2 \times 6 + 0.14 \times 5 + 0.12 \times 4 + 0.04 \times 3 = 4.5$

Let N be the number of instructions. The ratio between the total numbers of clock cycles for the multi-cycle processor and the single-cycle processor is:

$$\begin{aligned} \text{Ratio} &= \frac{\text{CPI}_{\text{multi}} \cdot N}{\text{CPI}_{\text{single}} \cdot N} \\ &= \frac{4.5 \cdot N}{1 \cdot N} \\ &= 4.5 \end{aligned}$$

b

- **Single-cycle:** Each cycle takes 60 ns. Therefore, the total time is $\text{T}_{\text{single}} = N \times 60\text{ns} = 60\text{ns} \times N$
- **Multi-cycle:** Each instruction takes an average of 4.5 cycles and each cycle takes 12 ns. Therefore, the total time is $\text{T}_{\text{multi}} = 4.5 \times N \times 12\text{ns} = 54\text{ns} \times N$

The speedup is:

$$
\begin{aligned}
\text{Speedup} &= \frac{\text{T}_{\text{single}}}{\text{T}_{\text{multi}}} \\
&= \frac{60\text{ns} \times N}{54\text{ns} \times N} \\
&= \frac{60}{54} \\
&\approx 1.11
\end{aligned}
$$

5

a

- `ld x12, 8(x3)` generates value in `x12` that is used in `or x14, x12, x13`.
- `addi x13, x4, 7` generates value in `x13` that is used in `or x14, x12, x13`.
- `or x14, x12, x13` generates value in `x14` that is used in `sd x14, 0(x5)`.

b

The five-stage pipelined processor takes **8 clock cycles** to complete this instruction sequence. Cycle-by-cycle analysis:

Cycle	1	2	3	4	5	6	7	8
ld x12, 8(x3)	IF	ID	EX	MEM	WB	-	-	-
add x13, x4, 7	-	IF	ID	EX	MEM	WB	-	-
or x14, x12, x13	-	-	IF	ID	EX	MEM	WB	-
sd x14, 0(x5)	-	-	-	IF	ID	EX	MEM	WB

No stalls are needed in this sequence because:

- The `add` instruction between `ld` and `or` prevents a load-use hazard.
- With data forwarding, x12, x13, and x14 can be used as soon as they are available.

c

No stall is needed. The only potential need for a stall would be between `ld` and `or`, but the `add` instruction (which is between them) prevents this hazard. The `ld` result is available in the `MEM` stage (at the end of cycle 4), and the `or` instruction can use it in the `EX` stage (at the beginning of cycle 5). Therefore, no stall is required.

d

Without forwarding and hazard detection, we need to insert 2 NOPs after the `add` instruction and 2 NOPs after the `or` instruction to ensure correct execution:

```
ld x12, 8(x3)
addi x13, x4, 7
NOP
NOP
```

```
or x14, x12, x13
NOP
NOP
sd x14, 0(x5)
```

This arrangement ensures that:
1. `ld` writes to x12 in cycle 5, and `or` reads it in cycle 6 (safe)
2. `addi` writes to x13 in cycle 6, and `or` reads it in cycle 7 (safe)
3. `or` writes to x14 in cycle 9, and `sd` reads it in cycle 9 (safe)

Cycle	1	2	3	4	5	6	7	8	9	10	11	12
ld x12, 8(x3)	IF	ID	EX	MEM	WB	-	-	-	-	-	-	-
add x13, x4, 7	-	IF	ID	EX	MEM	WB	-	-	-	-	-	-
NOP	-	-	-	-	-	-	-	-	-	-	-	-
NOP	-	-	-	-	-	-	-	-	-	-	-	-
or x14, x12, x13	-	-	-	-	IF	ID	EX	MEM	WB	-	-	-
NOP	-	-	-	-	-	-	-	-	-	-	-	-
NOP	-	-	-	-	-	-	-	-	-	-	-	-
sd x14, 0(x5)	-	-	-	-	-	-	-	IF	ID	EX	MEM	WB

Therefore, a minimum of 4 NOPs are needed to resolve all data hazards.

e

The processor takes **12 clock cycles** to complete the sequence in (d). As shown in the cycle-by-cycle analysis table in (d), the first instruction begins execution in cycle 1, and the last instruction (sd) completes its WB stage in cycle 12.

6

a

Cycle	1	2	3	4	5	6	7	8	9
add x3, x1, x2	IF	ID	EX	MEM	WB	-	-	-	-
sub x4, x1, x2	-	IF	ID	EX	MEM	WB	-	-	-
ld x5, 4(x3)	-	-	IF	ID	EX	MEM	WB	-	-
STALL	-	-	-	IF	ID	-	-	-	-
sub x1, x4, x5	-	-	-	-	-	(ID)	EX	MEM	WB

We need 1 stall cycle due to the load-use hazard (`ld` generates x5 which is needed by the following `sub` instruction).
The total number of cycles is 9 for 4 instructions. Therefore, the average CPI is:

$$\begin{aligned}\text{Average CPI} &= \frac{\text{Total cycles}}{\text{Number of instructions}} \\ &= \frac{9}{4} \\ &= 2.25\end{aligned}$$

b

Yes, it is possible to reorder the code to reduce the CPI. We can reorder as follows:

```
add x3, x1, x2
ld x5, 4(x3)
sub x4, x1, x2
sub x1, x4, x5
```

With this ordering, the pipeline execution becomes:

Cycle	1	2	3	4	5	6	7	8
add x3, x1, x2	IF	ID	EX	MEM	WB	-	-	-
ld x5, 4(x3)	-	IF	ID	EX	MEM	WB	-	-
sub x4, x1, x2	-	-	IF	ID	EX	MEM	WB	-
sub x1, x4, x5	-	-	-	IF	ID	EX	MEM	WB

By reordering, we avoid the load-use hazard.
The total number of cycles is 8 for 4 instructions. Therefore, the average CPI is:

$$\begin{aligned}\text{Average CPI} &= \frac{\text{Total cycles}}{\text{Number of instructions}} \\ &= \frac{8}{4} \\ &= 2.0\end{aligned}$$

7

a

Always taken: $\frac{3}{8} = 37.5\%$.
Always not taken: $\frac{5}{8} = 62.5\%$.

b

Ground truth	T	NT	T	NT	T	NT	NT	NT
State	T	T	NT	T	NT	T	NT	NT
Decision	T	T	NT	T	NT	T	NT	NT
Correctness	✓	×	×	×	×	×	✓	✓

Out of 8 predictions, 3 were correct.

$$\text{Accuracy rate} = \frac{3}{8} = 37.5\%$$

c

State	ST	WT	WNT	SNT
Description	Strongly Taken	Weakly Taken	Weakly Not Taken	Strongly Not Taken

Ground truth	T	NT	T	NT	T	NT	NT	NT
State	ST	ST	WT	ST	WT	ST	WT	WNT
Decision	T	T	T	T	T	T	T	NT
Correctness	✓	×	✓	×	✓	×	×	✓

Out of 8 predictions, 4 were correct.

$$\text{Accuracy rate} = \frac{4}{8} = 50\%$$

8

	IF	**ID**	**EX**	**MEM**	**WB**
1	`add x12, x6, x5`	-	-	-	-
2	`sub x10, x11, x12`	`add x12, x6, x5`	-	-	-
3	`beq x11, x12, LABEL`	`sub x10, x11, x12`	`add x12, x6, x5`	-	-
4	`sd x11, 0(x12)`	`beq x11, x12, LABEL`	`sub x10, x11, x12`	`add x12, x6, x5`	-
5	`First instruction of exception handler`	`NOP`	`NOP`	`sub x10, x11, x12`	`add x12, x6, x5`

9

Yes, it can be scheduled in four cycles. The valid static schedule is:

Cycle	**ALU**	**LS-slot**
1	`add x12, x6, x5`	`ld x30, 0(x13)`
2	`sub x10, x11, x12`	`ld x31, 0(x12)`
3	`add x30, x10, x30`	`sd x11, 0(x10)`
4	`sub x5, x11, x31`	`sd x30, 0(x10)`

Justification:

- All RAW dependencies are respected via full forwarding.
- The load-use latency for `ld x30, …` is met because its consumer (`add x30, …`) is scheduled in cycle 3, one cycle after the load's MEM stage.
- The load-use latency for `ld x31, …` is met because its consumer (`sub x5, …`) is scheduled in cycle 4, one cycle after the load's MEM stage.