Computer Systems II Homework 2

3220102060 Li Yingqi

November 10, 2023

Question 1 Pipeline Design Concept

(a)

- A. throughput

Pipelining enables the processor to execute multiple instructions simultaneously, which can improve the throughput of the processor.

- A. True

In a single-cycle processor, all instructions take the same amount of time to execute, which violates the idea of "make the common case fast".

- A. True

For a stage to produce a valid result, we need to provide it with sufficient time to complete the operation. So if there are more stages, each stage is more likely to be shorter and the latency of the pipeline is reduced, which means that the clock speed can be increased.

- A. True

The deeper the pipeline, the more likely it is to encounter hazards. In such cases the processor must stall or flush the pipeline, which will degrade the performance.

- B. 800 ps

Assume a data read is initiated at the EX stage and the data is available at the end of the EX stage. The pipeline may look like table 1.

Instructions	T_{clk}								
	1	2	3	4	5	6	7	8	9
`ld x1, 100(x4)`	IF	ID	EX	MEM	WB				
`add x4, x1, x0`		IF	ID[1]	ID[2]	EX	MEM	WB		
`ld x2, 200(x4)`			IF[1]	IF	ID[3]	EX	MEM	WB	
`ld x3, 400(x4)`					IF	ID[4]	EX	MEM	WB

[1] stalls since `x1` is not ready
[2] `x1` is forwarded from MEM stage of the first `ld` instruction
[3] `x4` is forwarded from EX stage of the `add` instruction
[4] `x4` is forwarded from MEM stage of the `add` instruction

Table 1: Pipeline of the given instructions

(b)

The longest critical path is Data Access which takes 500 ps, so the minimum clock period is 500 ps. Compared to 800 ps in the single-cycle implementation, the speedup is $800/500 = 1.6$.

(c)

Combining ID and EX stages yields a 150 ps stage. The longest critical path is still Data Access which takes 500 ps, so the speedup is $800/500 = 1.6$.

Question 2 Data Hazard and Forwarding

(a)

```
add x15, x12, x11    # (1)
nop                  # add at EX stage
nop                  # add at MEM stage
ld x13, 4(x15)       # (2)
ld x12, 0(x2)        # (3)
nop                  # ld(2) at MEM stage
or x13, x15, x13     # (4)
nop                  # or at EX stage
nop                  # or at MEM stage
sd x13, 0(x15)       # (5)
```

Listing 1: Assembly code with `nop`s inserted

Without forwarding, we cannot reduce the number of `nop`s by reordering the instructions. The best we can do is to swap `ld`(3) with `nop`s after it.

The space time diagram is shown in table 2.

Instructions	T_{clk}													
	1	2	3	4	5	6	7	8	9	10	11	12	13	14
`add x15, x12, x11`	IF	ID	EX	MEM	WB									
`nop`		—	—	—	—	—								
`nop`			—	—	—	—	—							
`ld x13, 4(x15)`				IF	ID	EX	MEM	WB						
`ld x12, 0(x2)`					IF	ID	EX	MEM	WB					
`nop`						—	—	—	—	—				
`or x13, x15, x13`							IF	ID	EX	MEM	WB			
`nop`								—	—	—	—	—		
`nop`									—	—	—	—	—	
`sd x13, 0(x15)`										IF	ID	EX	MEM	WB

Table 2: Space time diagram of the given instructions

(b)

For a CPU with full forwarding, "MEM to 1st only" will stall for one cycle. Therefore the percentage of stalls is $0.2/1.2 = 16.7\%$.

There is no "MEM to 1st and MEM to 2nd" because if MEM stage needs to forward to the 1st instruction, it must stall, and after the stall the 2nd instruction will not be affected.

(c)

If we only forward from EX/MEM register, the stalls needed for each category are shown in fig. 1. The average stalls per instruction is

$$0.05 \times 1 + 0.2 \times 2 + 0.05 \times 1 + 0.1 \times 1 + 0.1 \times 1 = 0.65$$

and therefore the CPI is $1 + 0.65 = 1.65$.

If we only forward from MEM/WB register, the stalls needed for each category are shown in fig. 2. The average stalls per instruction is

$$0.05 \times 1 + 0.2 \times 1 + 0.05 \times 0 + 0.1 \times 0 + 0.1 \times 1 = 0.35$$

and therefore the CPI is $1 + 0.35 = 1.35$.

Category	Stalls
EX to 1st only	0
MEM to 1st only	2
EX to 2nd only	1
MEM to 2nd only	1
EX to 1st and EX to 2nd	1

Figure 1: Only forward from EX/MEM

Category	Stalls
EX to 1st only	1
MEM to 1st only	1
EX to 2nd only	0
MEM to 2nd only	0
EX to 1st and EX to 2nd	1

Figure 2: Only forward from MEM/WB

Question 3 Control Hazard and Branch Prediction

The translated RISC-V assembly code may look like listing 2.

```
.start:
    lui t1, 0x800       # *p = A
    addi t0, zero, 0    # a
    addi a5, t1, 24     # A + 6
    addi t2, zero, 5    # 5
.forloop:
    beq t1, a5, .end
    lw a0, 0(a5)
    bge a0, t2, .forelse
    addi, t0, t0, 1
    jal zero, 0x8       # .forend
.forelse:
    addi, t0, t0, 2
.forend:
    addi, t1, t1, 4
    jal zero, -0x1c     # .forloop
.end:
```

Listing 2: Translated RISC-V assembly code

(a)

A total of 12 branches are executed (6 `beq` for `for`-loop check, 6 `bge` for `if-else` check).

For *always-taken* branch prediction, the number of mispredictions is 3. For *always-not-taken* branch prediction, the number of mispredictions is 9. Therefore the *always-taken* branch prediction is more accurate in this case.

(b)

Since the branch history table has 8 entries, we use `pc[4:2]` as the index, as the last two bits are always zero. According to listing 2, the `beq` instruction is at address `0x10` thus the branch history table entry is 4. The `bge` instruction is at address `0x18` thus the branch history table entry is 6. The branch history table is shown in table 3.

The number of mispredictions is 7, better than *always-not-taken* but worse than *always-taken*, since in this code the `for`-loop branch is only not-taken once. To improve the accuracy, we can initialize the branch history

	0	1	2	3	4	5	6
4	00	00 → 01	01 → 10	10 → 11	11 → 11	11 → 11	11 → 10
6	00	00 → 01	01 → 00	00 → 01	01 → 00	00 → 01	01 → 10

Table 3: Branch history table of entry 4 and 6. Red indicates misprediction, blue indicates correct prediction.

table to *weakly-taken* (10) or *strongly-taken* (11), which will reduce the number of mispredictions to 3, as shown in

	0	1	2	3	4	5	6
4	10	10 → 11	11 → 11	11 → 11	11 → 11	11 → 11	11 → 10
6	10	10 → 11	11 → 10	10 → 11	11 → 10	10 → 11	11 → 11

Table 4: Branch history initialized with *weakly-taken*

	0	1	2	3	4	5	6
4	11	11 → 11	11 → 11	11 → 11	11 → 11	11 → 11	11 → 10
6	11	11 → 11	11 → 10	10 → 11	11 → 10	10 → 11	11 → 11

Table 5: Branch history initialized with *strongly-taken*

(c)

The code shown in listing 3 is an implementation of the branch test in ID stage with forwarding. The data read is initiated at the EX stage and the data is available at the end of the EX stage.

In table 6, stalls caused by control hazards are not counted.

```
Forwarder forwarder0(
    // inputs
    .rs1_id(rs1_id),
    .rs2_id(rs2_id),
    .n_rd_id(ex_rd_id),
    .n_we_reg(ex_we_reg),
    .nn_rd_id(mem_rd_id),
    .nn_we_reg(mem_we_reg),

    // outputs
    // 2'b00 for no forwarding
    // 2'b01 for forwarding from EX
    // 2'b10 for forwarding from MEM
    .fwd_asel(fwd_asel),
    .fwd_bsel(fwd_bsel)
);

Mux4x64 bcomp_mux_a0(
    .i0(rs1_data),
    .i1(ex_alu_r),
    .i2(mem_rd_data),
    .i3(64'b0),
    .sel(fwd_asel),
    .o(bcomp_a)
);

Mux4x64 bcomp_mux_b0(
    .i0(rs2_data),
    .i1(ex_alu_r),
    .i2(mem_rd_data),
    .i3(64'b0),
    .sel(fwd_bsel),
    .o(bcomp_b)
);

BranchComp bcomp0(
    .a(bcomp_a),
    .b(bcomp_b),
    .bralu_op(bralu_op),
    .br_taken(br_taken)
);
```

Listing 3: Branch test in ID stage with forwarding

Instructions	stall cycles	`rs1` forward stage	`rs2` forward stage

`addi x1, x0, 1`
`addi x2, x0, 2`
`beq x1, x2, 0x10`
	0	MEM	EX

`addi x1, x0, 1`
`ld x2, 8(x1)`
`addi x3, x2, 1`
`beq x2, x3, 0x1c`
	0	MEM	EX

`ld x2, 8(x0)`
`ld x3, 24(x2)`
`beq x2, x3, 0x20`
	1		MEM

`addi x1, x0, 1`
`addi x2, x0, 1`
`beq x0, x1, 0x10`
	0		MEM

Table 6: Execution cycles of instruction sequences under forwarding implementation with branch test in ID stage