Computer Architecture, Section 379: Homework #4

Yousef Alaa Awad

November 3, 2025

1

Given: Answer the following questions based on the following table. All different datapaths support the following four instruction types with listed delay of each component. Assume no hazard is detected in the piplined datapath. Please explain your answer.

Instruction Class	Instruction Fetch	Register Read	ALU Operation	Data Access	Register Write	Total Time
Load Word (lw)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store Word (sw)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (add, sub, and, or, slt)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (beq)	200 ps	100 ps	200 ps			500 ps

A) What should the clock cycle time be for the single-cycle and pipelined data-paths? Please explain your answer.

For a single cycle data path the clock cycle time would be the highest time taken instruction time. In this case it would be the load word instruction, and therefore each cycle would take 800ps.

For a pipelined data-path, the clock cycle time would be the highest time taken per portion of an instruction. In this case, according to the table, it would be 200 ps.

B) What is the latency (defined as the delay from when the instruction enters the datapath until it finishes) of an R-format instruction in the single-cycle and pipelined datapaths?

The latency of a single cycle datapath R-format instruction would simply be the previous single cycle time that we found in Part A. Therefore, it is 800 ps.

The latency of a pipelined datapath R-format instruction would be the total stages that are nessecary multiplied by the time per stage. In the case of R-format instructions, it requires 4 stages (Fetch, Decode, Operate, and Write-Back) all of which taking 200 ps of time (as taken from Part A). Therefore, for a pipelined datapath, an R-format instruction would take 800 ps, coincidentally taking the same amount of time as a single-cycle datapath.

C) What is the latency (defined as the delay from when the instruction enters the datapath until it finishes) of load word (lw) instruction in the single-cycle and pipelined datapaths?

For load word in a single-cycle datapath, it would, again as in Part B, be simply the total cycle time, or, therefore, 800 ps.

For load word in the pipelined datapath, it would simply, as in Part B, be the amount of stages times the time per stage. In the case of load word, it contains 5 stages, all taking 200 ps per, making it have a total latency time of 1000 ps.

D) What is the averaged throughput (defined as the number of instructions executed in 1 nanosecond in this case) of the single-cycle and pipelined datapaths? Assume the frequency of four different instruction types are the same.

Now, before we begin, 1 nanosecond is 1000 picoseconds!!

For a single-cycle datapath each cycle takes 800 ps, as found in Part A. Therefore, the instructions per nanoseconds is simply $\frac{1 \text{ nanosecond}}{800 \text{ picosecond per instruction}}$ which is $\frac{1000 \text{ picoseconds}}{800 \text{ picosecond per instruction}}$ and comes out to 1.25 instructions per nanosecond.

For a pipelined datapath, each cycle takes 200 ps, which is the same as each stage time. Now, in a pipelined datapath, each stage is the execution of a different instruction, meaning that, after 5 instructions enter the datapath, you are completing an instruction every 200 ps/cycle. This, therefore, means that for every 1 nanosecond, there will be 5 instructions per nanosecond.

2

Given: The following codes run on the five-stage pipelined datapth. For each piece of code, answer the following questions as well.

```
sub t1, t2, t3
add t1, t4, t5
or t4, t2, t6
```
(a) Code 1

```
and t2, t5, t1
sub t3, t2, t0
nor t7, t1, t5
```
(b) Code 2

```
and t2, t5, t1
sub t3, t2, t0
nor t7, t2, t1
```
(c) Code 3

```
lw t1, 22(t0)
and t2, t1, t3
```
(d) Code 4

```
lw t1, 22(t0)
sub t6, t0, t2
xor t3, t1, t5
```
(e) Code 5

A) Without forwarding, insert the necessary number of 'nops' for the code to execute correctly.

Now, assuming that there is a 5-stage pipelined datapath, we need too, for all codes, find out if any instructions (up to 4 in the past) require any registers/memory located in the current instructions being executed. We do this so that we can stall the CPU via bubbles as the pipeline **does not** have any forwarding (of which, nowadays, isn't done anymore but I digress).

Code 1:

```
sub t1, t2, t3
add t1, t4, t5
or t4, t2, t6
```
Code 1

Now in code 1 (shown above) literally none of the following instructions require a result to be finalized before execution. Especially so since this is a single-core cpu.

Cycles	1	2	3	4	5	6	7
sub	FETCH	DECODE	EXEC	MEM	WB		
add		FETCH	DECODE	EXEC	MEM	WB	
or			FETCH	DECODE	EXEC	MEM	WB

Code 2:

```
and t2, t5, t1
sub t3, t2, t0
nor t7, t1, t5
```

Code 2

Now in code 2 (shown above) sub utilizes the register t2 from the previous instruction and, and since sub requires t2 to be written too, then the sub instruction requires to wait an additional 3 cycles before it can do a register read (or 3 nops). That is the only one that we need to insert NOPs for, and so it will look like the following code!

Cycles	1	2	3	4	5	6	7
and	FETCH	DECODE	EXEC	MEM	WB		
sub		FETCH	-	-	DECODE	EXEC	cont*
nor					FETCH	DECODE	cont*

**sub/nor continues as normal and is ommitted for space reasons*

Code 3:

```
and t2, t5, t1
sub t3, t2, t0
nor t7, t2, t1
```

Code 3

Now in code 3 (shown above) sub utilizes the register t2 from the previous instruction and, and since sub requires t2 to be written too, then the sub instruction requires to wait an additional 3 cycles before it can do a register read (or 3 nops), and thankfully by the time that nor requires t2 as well, nothing about it has to change. That is the only one that we need to insert NOPs for, and so it will look like the following code!

Cycles	1	2	3	4	5	6	7
and	FETCH	DECODE	EXEC	MEM	WB		
sub		FETCH	-	-	DECODE	EXEC	MEM
nor					FETCH	DECODE	cont*

**nor continues as normal and is ommitted for space reasons*

Code 4:

```
lw t1, 22(t0)
and t2, t1, t3
```

Code 4

Now in code 4 (shown above) and utilizes the register t1 from the previous instruction lw, and since lw requires t1 to be written too after loading from memory, then the and instruction requires to wait an additional 2 cycles before it can do a register read (or 2 nops). That is the only one that we need to insert NOPs for, and so it will look like the following code!

Cycles	1	2	3	4	5	6	7
lw	FETCH	DECODE	EXEC	MEM	WB		
and		FETCH	-	-	DECODE	EXEC	cont*

**and continues as normal and is ommitted for space reasons*

Code 5:

```
lw t1, 22(t0)
sub t6, t0, t2
xor t3, t1, t5
```

Code 5

Now in code 5 (shown above) xor utilizes the register t1 from the previous previous instruction lw, and since lw requires t1 to be written too after loading from memory, then the sub instruction occurs, AND THEN the xor instruction occurs and requires t1, it needs to wait an additional cycle before it can do a register read (or 1 nop). That is the only one that we need to insert NOPs for, and so it will look like the following code!

Cycles	1	2	3	4	5	6	7
lw	FETCH	DECODE	EXEC	MEM	WB		
sub		FETCH	DECODE	EXEC	MEM	WB	
xor			FETCH	-	DECODE	EXEC	cont*

**xor continues as normal and is ommitted for space reasons*

B) With the forwarding unit available (only supports forwards MEM→EX and WB→EX), show how the code will execute. Mention the data that's forwarded between the stages. Use 'nops' only when necessary.

Now, with forwarding available, it does not need to wait for an instruction to end!!! This therefore means that it can compress the NOPs needed and simply has to finish the stage that of register write/write-back! This therefore means the following for the codes (explanation is self-explanatory so it is ommited):

Code 1 - SAME

Cycles	1	2	3	4	5	6	7
sub	FETCH	DECODE	EXEC	MEM	WB		
add		FETCH	DECODE	EXEC	MEM	WB	
or			FETCH	DECODE	EXEC	MEM	WB

Code 2

Cycles	1	2	3	4	5	6	7
and	FETCH	DECODE	EXEC	MEM	WB		
sub		FETCH	DECODE	EXEC	MEM	WB	
nor			FETCH	DECODE	EXEC	MEM	WB

Note: t2 is being forwarded (MEM→EXEC)

Code 3

Cycles	1	2	3	4	5	6	7
and	FETCH	DECODE	EXEC	MEM	WB		
sub		FETCH	DECODE	EXEC	MEM	WB	
nor			FETCH	DECODE	EXEC	MEM	WB

Note: t2 is being forwarded both times (WB→EXEC and MEM→EXEC)

Code 4

Cycles	1	2	3	4	5	6	7
lw	FETCH	DECODE	EXEC	MEM	WB		
and		FETCH	-	DECODE	EXEC	MEM	WB

Note: t1 is being forwarded (WB→EXEC)

Code 5

Cycles	1	2	3	4	5	6	7
lw	FETCH	DECODE	EXEC	MEM	WB		
sub		FETCH	DECODE	EXEC	MEM	WB	
xor			FETCH	DECODE	EXEC	MEM	WB

Note: t1 is being forwarded (WB→EXEC)