Solutions

Q1-5 (Digital logic fundamentals 1)

Part A

A B C	Factorial	Div3
0 0 0	1	1
0 0 1	1	0
0 1 0	1	0
0 1 1	0	1
1 0 0	0	0
1 0 1	0	0
1 1 0	0	1
1 1 1	0	0

Part B

$$\begin{aligned}
Factorial(A,B,C) &= A'B'C' + A'B'C + A'BC' \\
&= A'(B'C' + B'C + BC') && \text{(Distributive law)} \\
&= A'(B'(C' + C) + BC') && \text{(Distributive law)} \\
&= A'(B' + BC') && \text{(Complement + identity laws)} \\
&= A'((B' + B)(B' + C')) && \text{(Distributive law)} \\
&= A'(B' + C') && \text{(Complement + identity laws)} \\
&= A'(BC)' && \text{(De Morgan's law)}
\end{aligned}$$

Alternative way to simplify it:

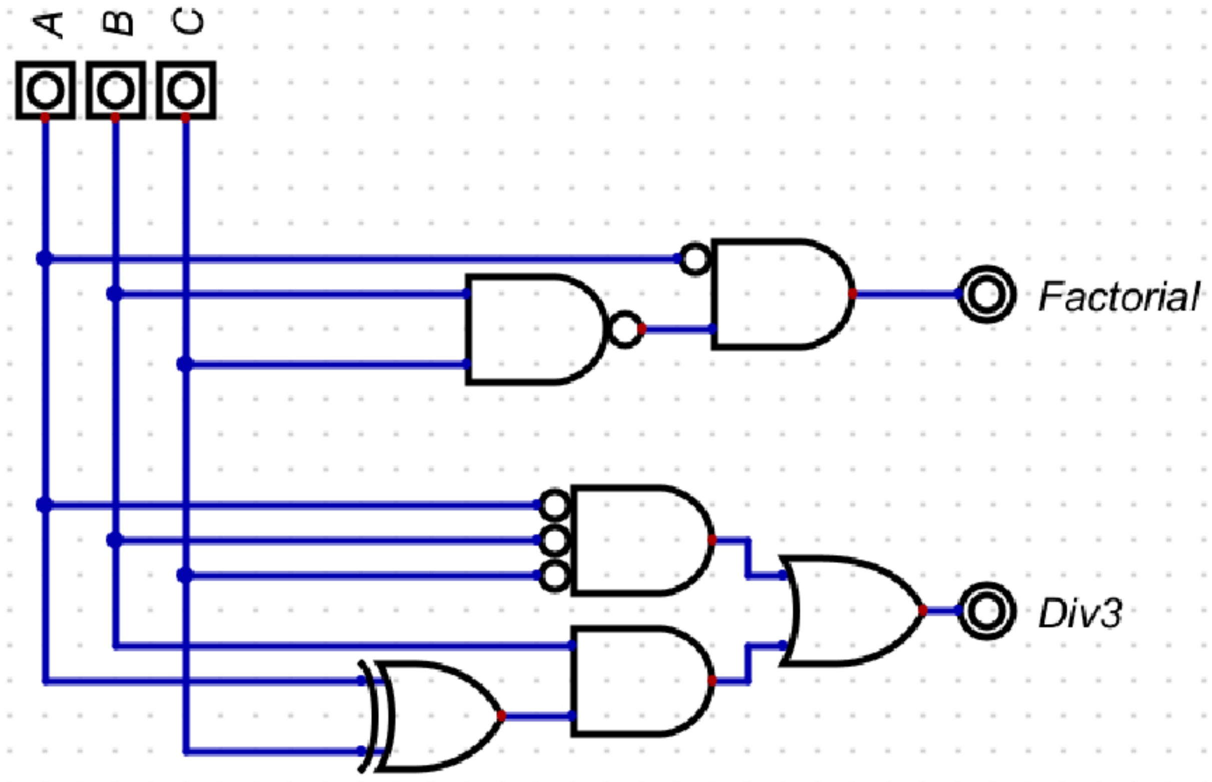
$$\begin{aligned}
Factorial(A,B,C) &= A'B'C' + A'B'C + A'BC' \\
&= A'B'C' + A'B'C' + A'B'C + A'BC' && \text{(A + A = A)} \\
&= A'(B'C' + B'C) + A'(B'C' + BC') && \text{(Distributive law)} \\
&= A'(B'(C' + C)) + A'(C'(B' + B)) && \text{(Distributive law)} \\
&= A'B' + A'C' && \text{(Complement + identity laws)} \\
&= A'(B' + C') && \text{(Distributive law)} \\
&= A'(BC)' && \text{(De Morgan's law)}
\end{aligned}$$

If you were sharp, you might have been able to observe that B’C’ + B’C + BC’ is the SOP for NAND by inspection as well.

Part C

$$\begin{aligned} Div3(A,B,C) &= A'B'C' + A'BC + ABC' \\ &= A'B'C' + B(A'C + AC') && \text{(Distributive law)} \\ &= A'B'C' + B(A \oplus C) && \text{(Formula for XOR)} \end{aligned}$$

Part D



Any circuit that matches the boolean equations from Parts C/D is marked correct.

Part E

- Lookup table is easier to implement
- Lookup table is configurable after manufacturing
- Basic logic gates uses less gates and is less spatially expensive
- Basic logic gates allow a shorter critical path

Q6-10 (Digital logic fundamentals 2)

A. X1, X2 and X3
B. False
C. C1, E2, G2, F2
D. A1, B1, D2, E2, G2
E. C1, E2, G2, F2

Q11-12 (Finite state machines 1)

Sample solution:

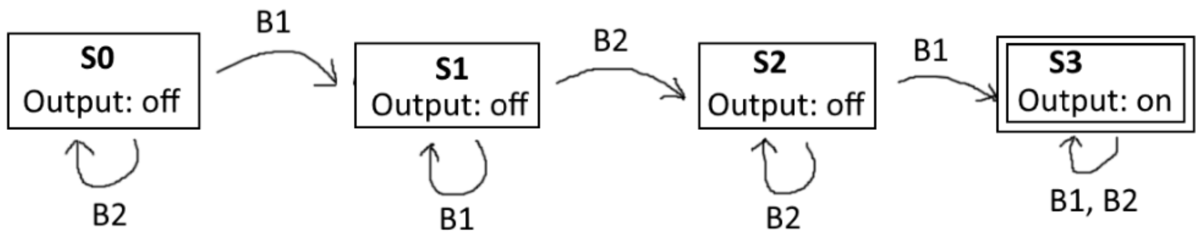
Part A

We build a separate finite state machine for each output.

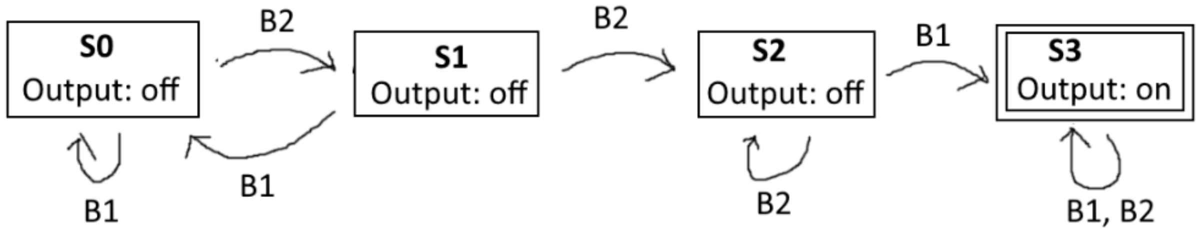
Because exactly one button is pressed each cycle, we can label each transition with a particular button (which implies the other button is not pressed).

State transition diagrams

Blue LED:



Red LED:



Note: we implicitly assumed that LEDs are never turned off once they are on, but we marked you as equally correct if you assumed otherwise for the red LED.

Encoded state transition table

Apply the following encodings for states:

State label	Encoding (2 bit number)
S0	00
S1	01
S2	10
S3	11

The encoded state transition tables are then as below.

Also label the 1st bit of the state as S0, and the 2nd bit of the state as S1 (in LITTLE endian order). The new values of these states are labelled with a prime.

The encoded state transition tables are then as below.

Blue LED:

Current state		**Inputs**		**Next state**	
S1	**S0**	**B1**	**B2**	**S1’**	**S0’**
0	0	1	0	0	1
0	0	0	1	0	0
0	1	1	0	0	1
0	1	0	1	1	0
1	0	1	0	1	1
1	0	0	1	1	0
1	1	X	X	1	1

Red LED:

Current state		**Inputs**		**Next state**	
S1	**S0**	**B1**	**B2**	**S1’**	**S0'**
0	0	1	0	0	0
0	0	0	1	0	1
0	1	1	0	0	0
0	1	0	1	1	0
1	0	1	0	1	1

1	0	0	1	1	0
1	1	X	X	1	1

Note: Cases where neither button is pressed or both buttons are pressed at once are **undefined behavior**. You are free to do anything in those cases. You can either explicitly state what happens in those cases in the above tables, or exclude them entirely.

Next state equations

Just use “sum of products” on the above tables. No need for simplification, though in below solutions some simplifications may have been applied.

For blue LED:

$$S_0' = S_1'S_0'B_1B_2' + S_1'S_0B_1B_2' + S_1S_0'B_1B_2' + S_1S_0$$
$$S_1' = S_1'S_0B_1'B_2 + S_1$$

For red LED:

$$S_0' = S_1'S_0'B_1'B_2 + S_1S_0'B_1B_2' + S_1S_0$$
$$S_1' = S_1'S_0B_1'B_2 + S_1$$

Output equations

For both LEDs, output is 1 if and only if state is 11, so the output equation is:

$$\text{Output} = S_0S_1$$

Notes for alternative correct solutions

- Valid to assume that the red LED is turned off again if the button press sequence is broken
    - We clarified this point on Ed, but several students have said they didn’t see this clarification, so we’re marking this as correct as well
- Combining the blue LED and red LED’s FSMs into a single FSM
- Using a different encoding (e.g. one hot encoding)

Part B

We must ensure all propagation of signals between state elements is complete within a clock period.

In this case, the only such propagation delay to consider is that between the state register output and input, which consists of

- Clock to Q delay for state register (for getting current state)
- Computation of next state logic
- Setup time for state register (for storing the next state)

Hence, we will compute the critical path time as the sum of the above delays, and set our clock period as low as possible while being greater than the critical path time.

Note: some people noted that we said that a button is pressed every cycle and hence wrote that human reaction time will decide cycle time. This is true, but was only awarded full marks if it was justified that circuit propagation delay was insignificant in comparison. Honestly quite unrealistic to press a button every clock cycle…

Q13-15 (Finite state machines 2 - PPUs)

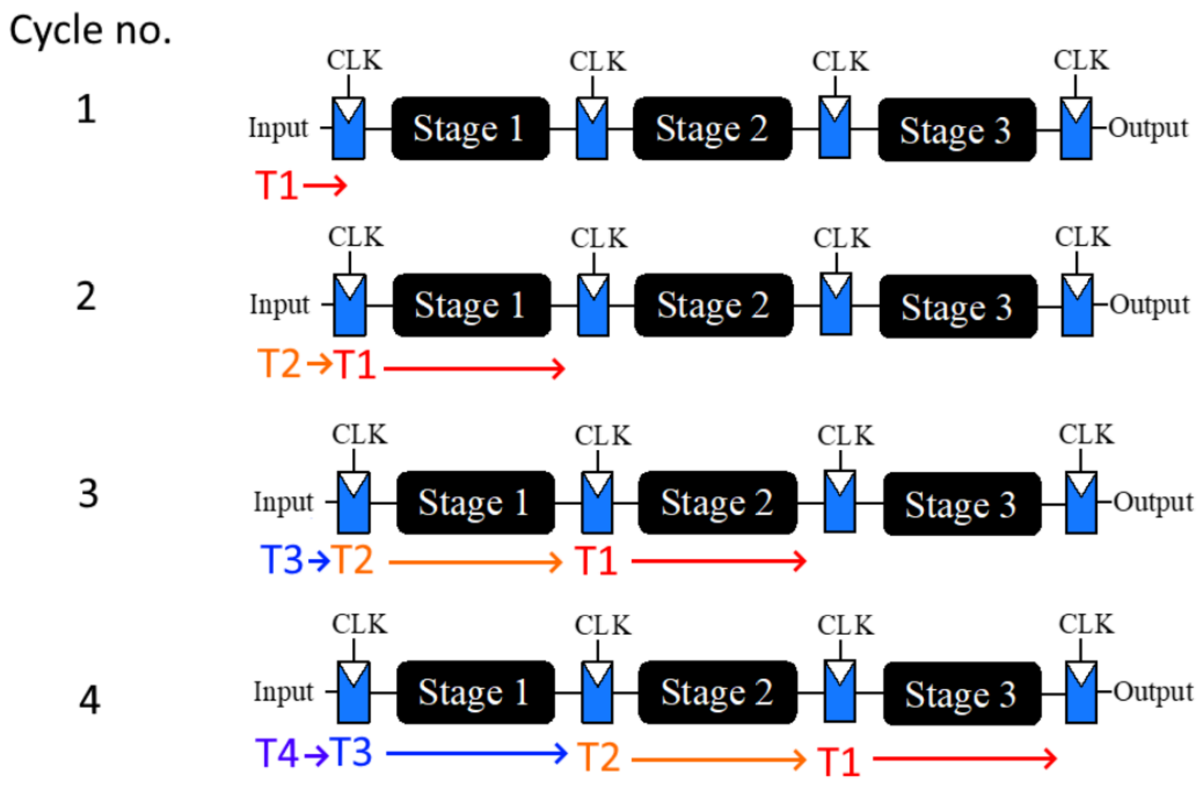
Part A

We first need to determine the minimum clock period we can clock this PPU at, which is given by the longest propagation delay between state elements/registers; that is the time taken by Stage 3 in this case, which is:

$$t = t_{pcq} + t_{stage-3} + t_{setup} = \mathbf{0.2 + 3.7 + 0.1 = 4\ ns}$$

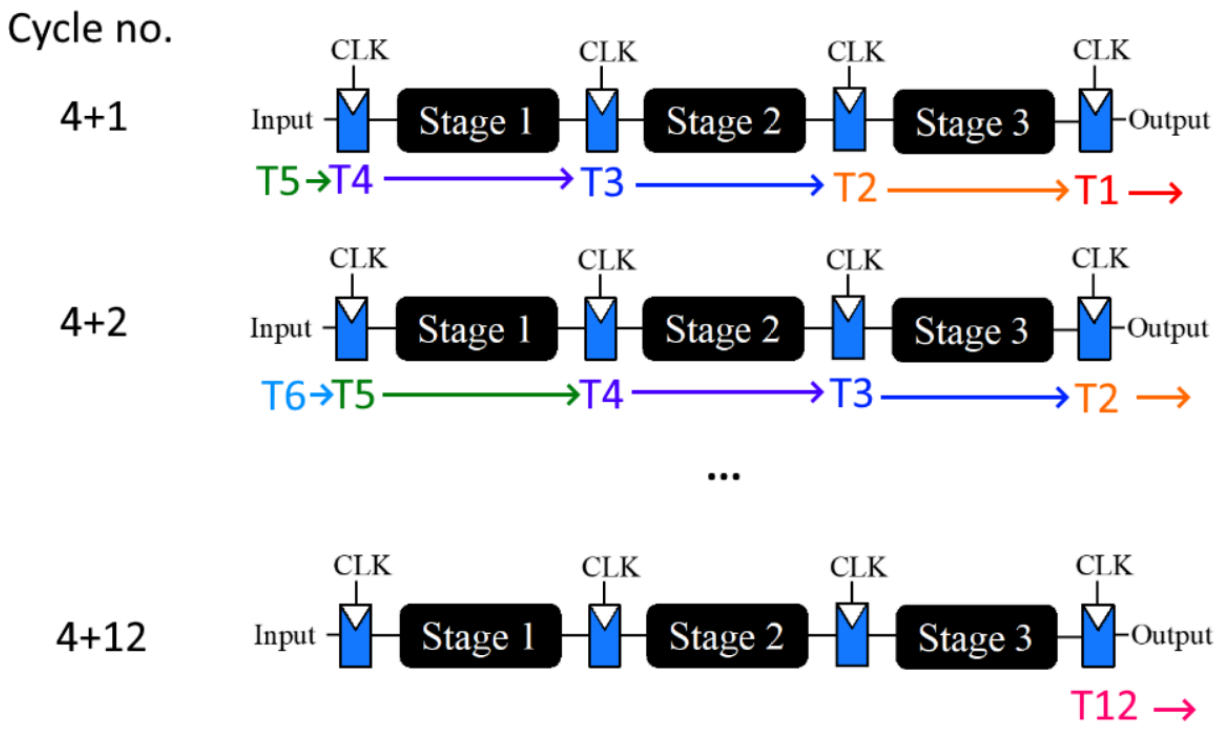
So this is the minimum clock period.

Then, we consider the number of cycles for processing 12 tokens. In each cycle, one token appears on the input of the PPU, and we consider the processing complete after all the tokens have been outputted (at the end of the cycle where the last token is outputted). We first have four cycles as the pipeline is filled up:



In this diagram, the tokens are labeled T1, T2, and so on. Note that because of the register sitting in front of stage 1, the first token cannot actually start Stage 1 until one clock cycle has passed.

Then, following that, each cycle one token is outputted until all 12 tokens have been outputted:



In total, it will take 4 + 12 = 16 cycles till all tokens have been outputted. Given the clock period of 4 ns, this means a time of 16 cycles * 4 ns per cycle = 64 ns.

Note: this question has room for debate as to how many cycles it actually takes to fill the pipeline, e.g. does the input have to wait a cycle before entering the input register or can it be assumed to start inside that first register.

We will be more clear about these issues in future exams.

Part B

The logic is the same except now after cycle 4 we have a billion tokens to output each cycle, adding up to 1,000,000,004 cycles. Multiply by 4 ns per cycle to get a time of 4.000000016 seconds (4 seconds + 16 ns).

Part C

We simply split the billion tokens across the four PPUs so each PPU processes 250 million tokens, and the number of cycles that takes is calculated as before, resulting in 250,000,004 cycles. Multiply by 4 ns per cycle to get a time of 1.000000016 seconds (1 second + 16 ns).

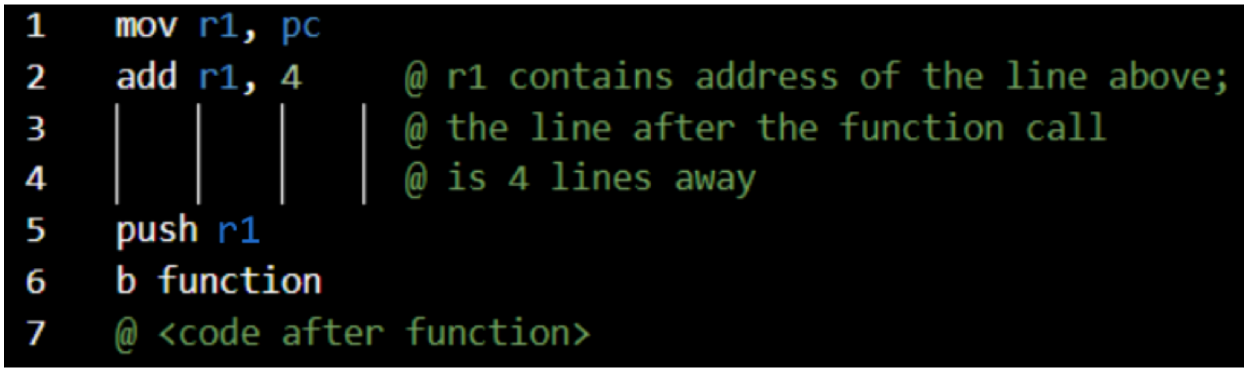
Q16-17 (Assembly and ISA 1 - fibonacci)

Part A

The purpose of the link register is to store the return address before jumping to a function, so then by moving that value back into PC we return to where the function was called and continue. Without a link register, we still need to store the return address somewhere, and the only choice is memory. For nested functions we will need to store multiple return addresses, and since we want to use the most recent value of PC stored first (i.e. data structure should be FIFO), we can store the PC on the stack.

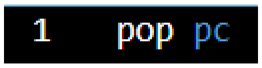
The return address should correspond to the instruction AFTER the jump; it should be incremented before pushing accordingly.

You would jump to a function with the following:



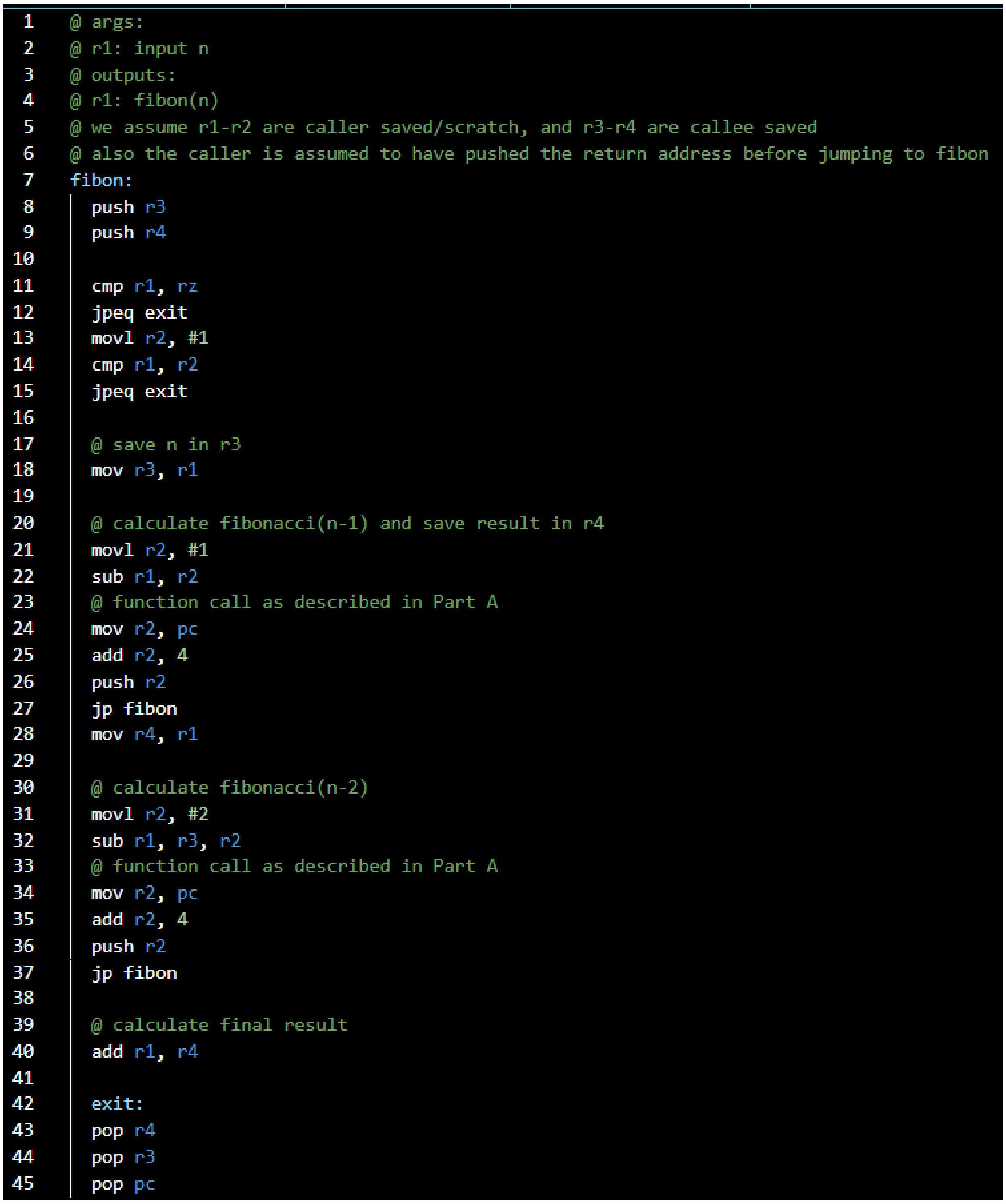
```
mov r1, pc
add r1, 4      @ r1 contains address of the line above;
               @ the line after the function call
               @ is 4 lines away
push r1
b function
@ <code after function>
```

And return from a function using:



```
pop pc
```

Part B



```
@ args:
@ r1: input n
@ outputs:
@ r1: fibon(n)
@ we assume r1-r2 are caller saved/scratch, and r3-r4 are callee saved
@ also the caller is assumed to have pushed the return address before jumping to fibon
fibon:
  push r3
  push r4

  cmp r1, rz
  jpeq exit
  movl r2, #1
  cmp r1, r2
  jpeq exit

  @ save n in r3
  mov r3, r1

  @ calculate fibonacci(n-1) and save result in r4
  movl r2, #1
  sub r1, r2
  @ function call as described in Part A
  mov r2, pc
  add r2, 4
  push r2
  jp fibon
  mov r4, r1

  @ calculate fibonacci(n-2)
  movl r2, #2
  sub r1, r3, r2
  @ function call as described in Part A
  mov r2, pc
  add r2, 4
  push r2
  jp fibon

  @ calculate final result
  add r1, r4

  exit:
  pop r4
  pop r3
  pop pc
```

Some alternative solutions also marked as correct.

Q18-Q32 (Assembly and ISA 2)

1. Uarch
2. ISA
3. Uarch
4. Uarch
5. ISA
6. Uarch
7. ISA
8. ISA
9. Uarch
10. Uarch
11. ISA
12. Uarch
13. Uarch
14. Uarch
15. ISA

Q33 (Assembly and ISA 3)

The correct answers are:

- Complex instructions instead of reduced instructions ,
- Sophisticated memory addressing modes ,
- Wide registers ,
- Conditional execution for all instructions ,
- A large programmer-visible register file

Q34 (I/O 1 - Amdahl's Law)

Sample solution

We need to be careful here, as the two upgrades target only a part of the program, and the overall speedup may not correspond to the disk speedup/CPU speedup.

Amdahl's law gives that if part of the program, of proportion p of the total program, is sped up by a factor of s, the speedup of the total program, denoted by S, is:

$$S = \frac{1}{(1-p) + p/s}$$

For the disk upgrade, 50% of the program (disk usage) is made 1.2 times faster. This makes the overall program's speedup as below, according to Amdahl's law:

$$\begin{aligned} S &= \frac{1}{(1-p)+p/s} \\ &= \frac{1}{0.5+0.5/1.2} \\ &\approxeq 1.091 \end{aligned}$$

This corresponds to a 9.1% increase for a cost of \$4000, and hence the cost is \$440 per 1% increase of performance.

For the CPU upgrade, 50% of the program (CPU usage) is made 2 times faster. His makes the overall program's speedup as below:

$$\begin{aligned} S &= \frac{1}{(1-p)+p/s} \\ &= \frac{1}{0.5+0.5/2} \\ &\approxeq 1.333 \end{aligned}$$

This corresponds to a 33.3% increase for a cost of \$6000, and hence the cost is \$180 per 1% increase of performance.

Clearly the CPU upgrade is better.

Q35-37 (I/O 2)

Part A

Programmed interrupt driven I/O

- Slow, so you don't want to waste time waiting for it (polling is bad)
- Amount of data being transferred is quite small so DMA is unnecessary.

Part B

DMA interrupt driven I/O

- Slow, so you don't want to waste time waiting for it (polling is bad)
- Amount of data transferred is large; manually transferring it would cost a lot of CPU time, which DMA avoids

Part C

Programmed polling I/O

- All a thermostat does is keep displaying the current temperature, so it's reasonable to just poll for it

Q39-42 (Microarchitecture 2)

Part A

Part B

Issues that pose threats to correctness in this case are (1) data dependencies and (2) control hazards.

The data dependencies in this program are:

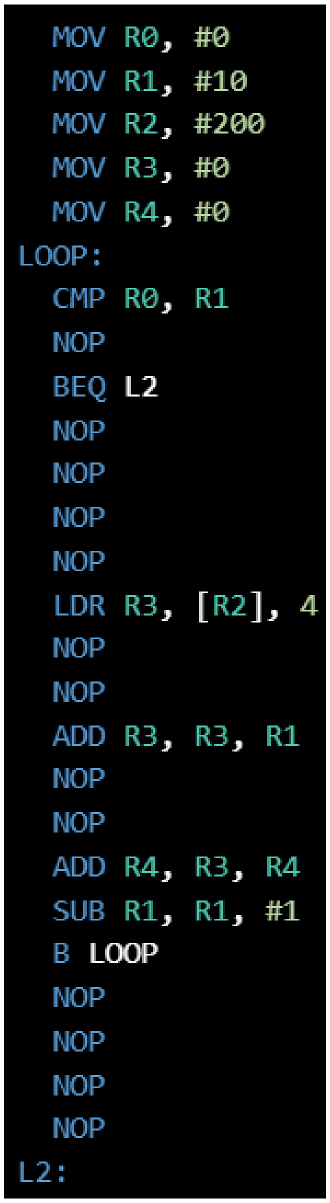
- Between i6 and i7 (BEQ depends on flags generated by CMP)
- Between i8 and i9 (i8 writes r3, i9 reads r3)
- Between i9 and i10 (i9 writes r3, i10 reads r3)

The first dependency is a flags dependency; i6 updates flags at the end of EX (which in this CPU appear to be directly in the control unit). I7 may read flags in different places depending on the CPU architecture; the version in the textbook reads it during EX, which actually means this requires no NOPs. This dependency was quite an edge case and marked lightly in the original exam, and we will be more clear about it in future years.

The second and third dependencies are register file data dependencies; the second instruction writes the data to RF in WB stage and the first reads them in the DE/RF-read stage. We assume that if these two happen simultaneously, the write happens before the read. So we need a gap of 2 stages between these instructions, i.e. 2 NOPs.

As for control hazards, there are two branch instructions: i6 and i12. There are 4 instructions that are fetched after the branch is fetched and before it completes writeback. This means we need 4 NOPs after each branch. (Note: 3 NOPs, from assuming PC writeback occurs before PC read for instruction fetch if occurring in the same cycle, was also accepted).

The final program after adding NOPs:


```
  MOV R0, #0
  MOV R1, #10
  MOV R2, #200
  MOV R3, #0
  MOV R4, #0
LOOP:
  CMP R0, R1
  NOP
  BEQ L2
  NOP
  NOP
  NOP
  NOP
  LDR R3, [R2], 4
  NOP
  NOP
  ADD R3, R3, R1
  NOP
  NOP
  ADD R4, R3, R4
  SUB R1, R1, #1
  B LOOP
  NOP
  NOP
  NOP
  NOP
L2:
```

IPC, instructions per cycle, is calculated as:

- There are 4 cycles spent to fill up the pipeline initially, after which 1 instruction from the program executes every cycle
- Before loop: 5 instructions in 5 cycles
- Loop body: for each of the 10 iterations that the loop body runs, we execute 7 useful instructions and 13 NOPs in 7 + 13 = 20 cycles
- Loop exit: We execute 2 instructions and 5 NOPs during the loop exit in 2 + 5 = 7 cycles

This gives an IPC of:

$$\begin{aligned}\text{IPC} &= \frac{\text{no instructions}}{\text{no cycles}} \\ &= \frac{5 + 10 * 7 + 2}{4 + 5 + 10 * 20 + 7} \\ &\approx 0.356 \text{ instructions per cycle}\end{aligned}$$

Note: counting the NOPs as instructions was also marked as correct.

Part C

Pair of instructions that no longer stall + pair that still stalls:

i9 and i10 no longer stall, as when i10 is in RF-read, i9 is in EX and finishes computing its value at the end of EX (as it is an ALU instruction), which it can forward to i10 before it reaches EX. i8 and i9 still result in a stall however because i8 is a ldr and so its value isn't ready until after MEM, by when i9 will have finished EX and it is too late to forward the result to it unless it is stalled.

New IPC:

- 4 cycles to fill pipeline
- Preamble runs with no stalls (5 instructions, 5 cycles)
- Loop body: for each of the 10 iterations we run 7 instructions in 7 cycles + 5 cycles of stalling (1 cycle due to load-use hazard, 4 cycles due to unconditional branch)
- Loop exit: we run 2 instructions in 2 cycles + 2 cycles of penalty for mispredicted conditional branch

Giving an IPC of:

$$\begin{aligned}\text{IPC} &= \frac{\text{no instructions}}{\text{no cycles}} \\ &= \frac{5 + 10 * 7 + 2}{4 + 5 + 10 * (7 + 5) + 2 + 2} \\ &\approx 0.579 \text{ instructions per cycle}\end{aligned}$$

Part D

There is only one conditional branch in the program, which is untaken for each iteration that the loop continues and taken when the loop exits. This results in a branch direction pattern of 10 untaken branches followed by 1 taken one.

If the code executes once, then the Smith predictor, starting with counter 11 (strongly taken), mispredicts the first two branches (which are untaken). By this point the counter has changed to 01 (weakly untaken), and it predicts the next eight branches correctly as untaken. When the loop exits, the counter is 00 (strongly untaken) so it mispredicts the taken branch. Ultimately, accuracy is 8/11 = 72%.

If the code executes four times, after the first execution the Smith counter is 01 (weakly untaken) as a result of the taken branch that occurred at the end, which means it predicts the next (untaken) branch correctly as well as the subsequent 9 untaken branches. It only mispredicts the taken branch when the loop exits. Ultimately, accuracy increases to 38/44 = 86.3%.

Q43-45 (Microarchitecture 3)

Part A

- {i1 to i3, true, by way of R1}
- {i3 to i5, true, by way of R1}
- {i1 to i3, output, by way of R1}
- {i2 to i4, true, by way of R2}
- {i4 to i6, true, by way of R2}
- {i2 to i4, output, by way of R2}

Note: recall for the STR instructions, the first operand is not destination register but rather the value being stored, so i5 corresponds to a read of R1 (not a write!) and similar for i6.

Also, note that because i3 updates R1 in between i1 to i5, i1 to i5 is NOT a true dependency. Similarly for i2 to i6.

Part B

Two independent chains, which are:

- {i1, i3, i5}
- {i2, i4, i6}

Part C

Two.

We cannot reorder or simultaneously issue any instructions in a dependent chain, which means only at most two instructions in this program (one from each of the chains) can run at a time.

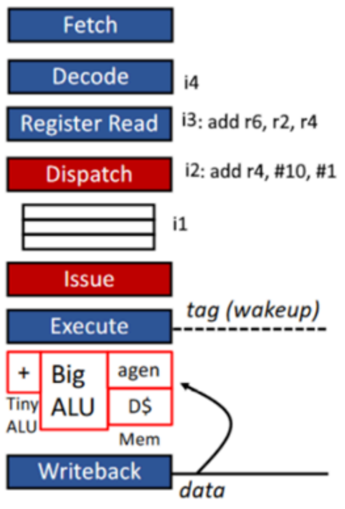
(Note: if you thought we were talking in general, not just about instructions from the above program, we also accepted that since question wording is unclear)

Q46-49 (Microarchitecture 4)

Part A

The overall situation of the processor at the end of each cycle number of interest is as follows:

Cycle 5



Scoreboard

	v
R0	1
R1	1
R2	0
R3	1
R4	0
R5	1
R6	0
R7	1

Register File

	value
R0	#10
R1	#16
R2	#10
R3	#10
R4	#10
R5	#10
R6	#10
R7	#10

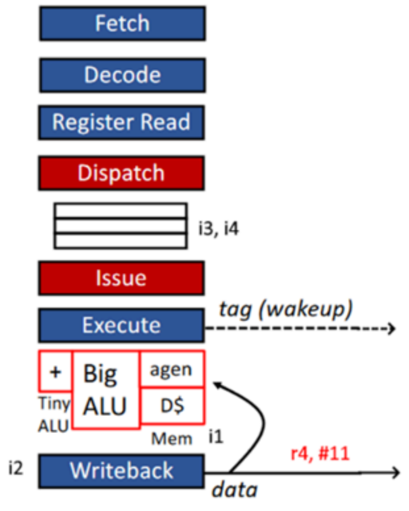
Issue Queue (IQ)

v	dst tag	rs1 rdy	rs1 tag/value	rs2 rdy	rs2 tag/value
1	r2	1	#16	1	#0
0					
0					

Pipeline cycle-by-cycle diagram format:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: LDR R2, [R1,#0]	FE	DE	RR	DI	IS												
i2: ADD R4, R3, #1		FE	DE	RR	DI												
i3: ADD R6, R2, R4			FE	DE	RR												
i4: SUB R7, R6, #3				FE	DE												

Cycle 8



Scoreboard

	v
R0	1
R1	1
R2	0
R3	1
R4	1
R5	1
R6	0
R7	0

Register File

	value
R0	#10
R1	#16
R2	#10
R3	#10
R4	#11
R5	#10
R6	#10
R7	#10

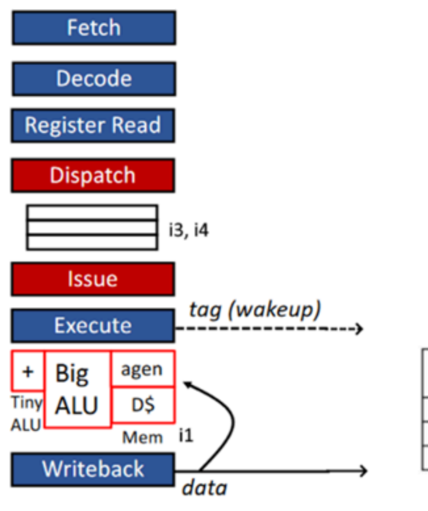
Issue Queue (IQ)

v	dst tag	rs1 rdy	rs1 tag/value	rs2 rdy	rs2 tag/value
1	r6	0	r2	1	#11
1	r7	0	r6	1	#3
0					

Pipeline cycle-by-cycle diagram format:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: LDR R2, [R1,#0]	FE	DE	RR	DI	IS	$EX_{@}$	$EX_{D\$}$	MEM									
i2: ADD R4, R3, #1		FE	DE	RR	DI	IS	EX	WB									
i3: ADD R6, R2, R4			FE	DE	RR	DI	IS	IS									
i4: SUB R7, R6, #3				FE	DE	RR	DI	IS									

Cycle 9



Scoreboard

	v
R0	1
R1	1
R2	0
R3	1
R4	1
R5	1
R6	0
R7	0

Register File

	value
R0	#10
R1	#16
R2	#10
R3	#10
R4	#11
R5	#10
R6	#10
R7	#10

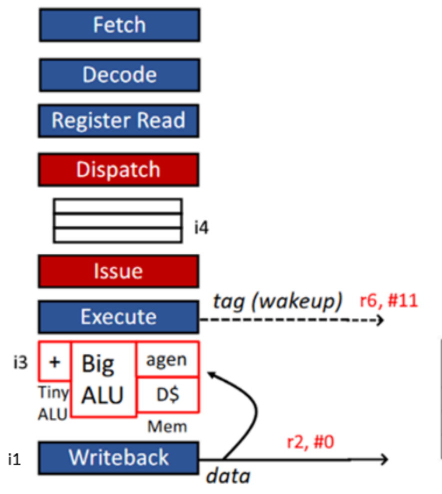
Issue Queue (IQ)

v	dst tag	rs1 rdy	rs1 tag/value	rs2 rdy	rs2 tag/value
1	r6	0	r2	1	#11
1	r7	0	r6	1	#3
0					

Pipeline cycle-by-cycle diagram format:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: LDR R2, [R1,#0]	FE	DE	RR	DI	IS	$EX_{@}$	$EX_{D\$}$	MEM	MEM								
i2: ADD R4, R3, #1		FE	DE	RR	DI	IS	EX	WB									
i3: ADD R6, R2, R4			FE	DE	RR	DI	IS	IS	IS								
i4: SUB R7, R6, #3				FE	DE	RR	DI	IS	IS								

Cycle 12



Scoreboard

	v
R0	1
R1	1
R2	1
R3	1
R4	1
R5	1
R6	0
R7	0

Register File

	value
R0	#10
R1	#16
R2	#0
R3	#10
R4	#11
R5	#10
R6	#10
R7	#10

Issue Queue (IQ)

v	dst tag	rs1 rdy	rs1 tag/value	rs2 rdy	rs2 tag/value
0	r6	1	#0	1	#11
1	r7	1	#11	1	#3
0					

Pipeline cycle-by-cycle diagram format:

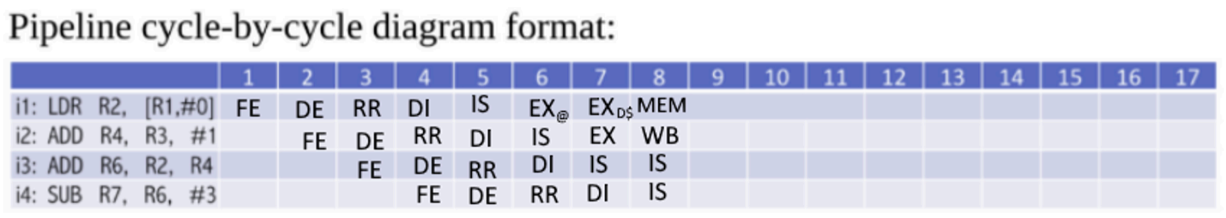
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: LDR R2, [R1,#0]	FE	DE	RR	DI	IS	$EX_{@}$	$EX_{D\$}$	MEM	MEM	MEM	MEM	WB					
i2: ADD R4, R3, #1		FE	DE	RR	DI	IS	EX	WB									
i3: ADD R6, R2, R4			FE	DE	RR	DI	IS	IS	IS	IS	IS	EX					
i4: SUB R7, R6, #3				FE	DE	RR	DI	IS	IS	IS	IS	IS					

For this question in particular:

Cycle #8:

Issue queue state:
{i3: v=1, dst tag = r6, rs1 rdy = 0, rs1 tag = r2, rs2 rdy = 1, rs2 val = 11}
{i4: v=1, dst tag = r7, rs1 rdy = 0, rs1 tag = r6, rs2 rdy = 1, rs2 val = 3}

Pipeline cycle-by-cycle diagram:



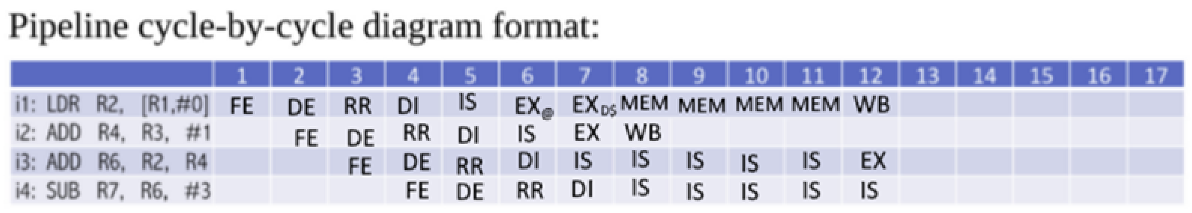
Pipeline cycle-by-cycle diagram format:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: LDR R2, [R1,#0]	FE	DE	RR	DI	IS	$EX_@$	$EX_{D\$}$	MEM									
i2: ADD R4, R3, #1		FE	DE	RR	DI	IS	EX	WB									
i3: ADD R6, R2, R4			FE	DE	RR	DI	IS	IS									
i4: SUB R7, R6, #3				FE	DE	RR	DI	IS									

Cycle #12:

Issue queue state:
{i4: v=1, dst tag = r7, rs1 rdy = 1, rs1 val = 11, rs2 rdy = 1, rs2 val = 3}

Pipeline cycle-by-cycle diagram



Pipeline cycle-by-cycle diagram format:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: LDR R2, [R1,#0]	FE	DE	RR	DI	IS	$EX_@$	$EX_{D\$}$	MEM	MEM	MEM	MEM	WB					
i2: ADD R4, R3, #1		FE	DE	RR	DI	IS	EX	WB									
i3: ADD R6, R2, R4			FE	DE	RR	DI	IS	IS	IS	IS	IS	EX					
i4: SUB R7, R6, #3				FE	DE	RR	DI	IS	IS	IS	IS	IS					

Note: invalid rows in the issue queue are omitted from the above. Also, it was confusing whether the "i1" at the beginning of the example text corresponded to a row number of the issue queue or an instruction number; both are marked as correct.

Part B

Cycle #5: 11010101
Cycle #9: 11011100

Part C

{R0 = 10, R1 = 16, R2 = 0, R3 = 10, R4 = 11, R5 = 10, R6 = 10, R7 = 10}

Part D

Tag of the register being written to (R4), value being written (#11)

Alternatively, "r4, #11"

Q50-51 (Bonus microarchitecture)

Part A (ranking microarchitectures)

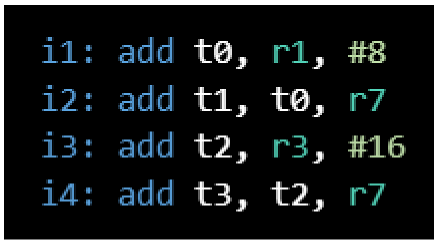
Sample solution

C, A, E, B, G, F, D

Notes:

- Hardware speculation encapsulates all of dynamic scheduling, register renaming and branch prediction, so C has everything
- C -> A -> E: progressively removing features from superscalar out-of-order CPU
- B is below C-E because in-order execution bottlenecks exploitation of ILP severely
- B -> G -> F: progressively removing features from in-order pipelined CPU
- D is below B-F because it has no exploitation of ILP at all

Part B (register renaming)



```
i1: add t0, r1, #8
i2: add t1, t0, r7
i3: add t2, r3, #16
i4: add t3, t2, r7
```