Solutions

Digital Logic Fundamentals 1

Part A

Sample solution

A	B	C	O	E
0	0	0	0	1
0	0	1	1	0
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	0	1
1	1	1	1	0

Rubric

Total marks: 5

Mark breakdown:

- Truth table is set up with correct rows and columns, and input combinations (1 mark)
- E column is correct (2 marks)
    - Minus 1 mark if input of 000 gives output of 0 for E
- O column is correct (2 marks)

Parts B and C

Sample solution

They should be able to simplify O to “C” and E to “NOT C”

COMP students may also use the notation $\wedge$ for AND, $\vee$ for OR and $\neg$ for NOT, which is the standard in COMP1600 and MATH1005.

Rubric

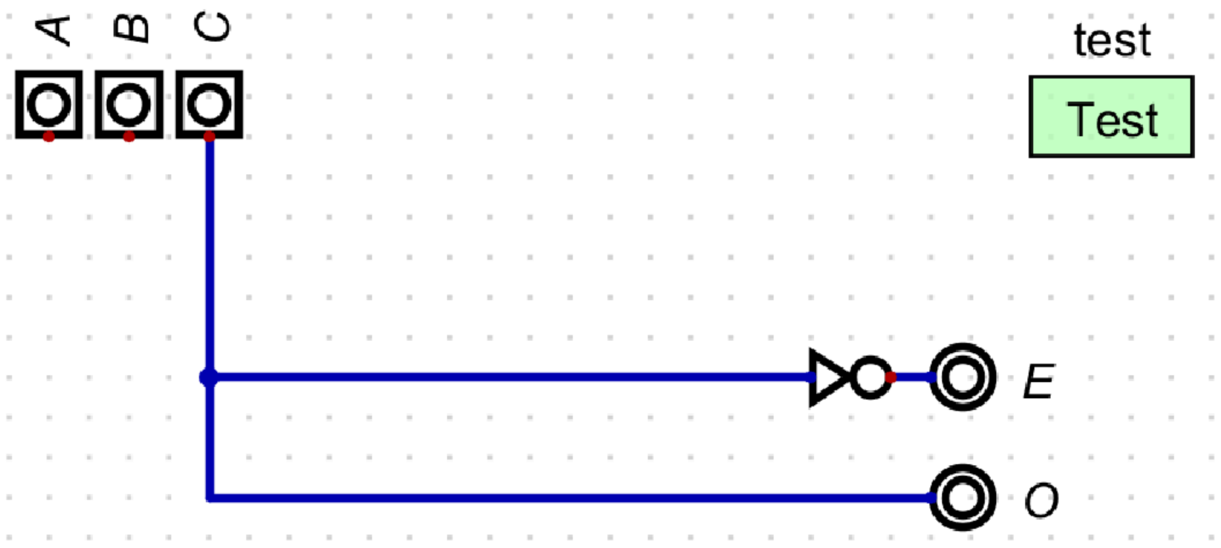
Total marks: 10 for each part

Mark breakdown:

- Correct sum of products (6 marks)
- Correct simplification with working out/showing steps (4 marks)
    - If they do some simplification but could have gone further, award 2-3 marks (depending on how far they got)
    - No need to state exactly which law/rule is being used at each point in the simplification, as long as it is reasonably clear
- Result is to be considered correct as long as it fits the truth table they gave in Part A (to avoid double punishing for earlier mistakes)

Part D

Sample solution



Rubric

Total marks: 10

Any schematic is to be marked correct as long as it:

- Fits the Boolean equations given in Parts B/C (may be simplified)
- Uses only AND, OR, XOR, NAND, NOR, NXOR, NOT
    - Three-input or negated input versions of these gates are allowed too
- It is acceptable to do O and E as seperate circuits, or as a single circuit where the outputs are merged into one 2-bit output.

Part E

(Note: different question from normal exam:)

Explain the difference between a demultiplexer and a decoder. Which one of the two has a higher logic complexity? Assume when comparing them that they have the same number of select bits and the demultiplexer input has 3 bits.

Sample solution

- Demultiplexers and decoders both have a select input and one output for each possible select input, but demultiplexer has an additional input which decoder doesn't. Additionally, if that input is "a", demultiplexer sends "a" to the selected output and 0 to all others while decoder sends 1 to the selected output and 0 to all others. Demultiplexer requires more logic as it is essentially a more general version of the decoder.

Rubric

Total marks: 5

Mark breakdown:
- Identifying what a decoder and a demultiplexer are (3 marks)
  - Minus 1.5 marks if they didn't read the "de" part and describe a multiplexer instead
- Stating and explaining correctly that demux has higher complexity (2 mark)

Digital Logic Fundamentals 2

Correct answers:
A: d
B: b
C: c
D: d
E: a

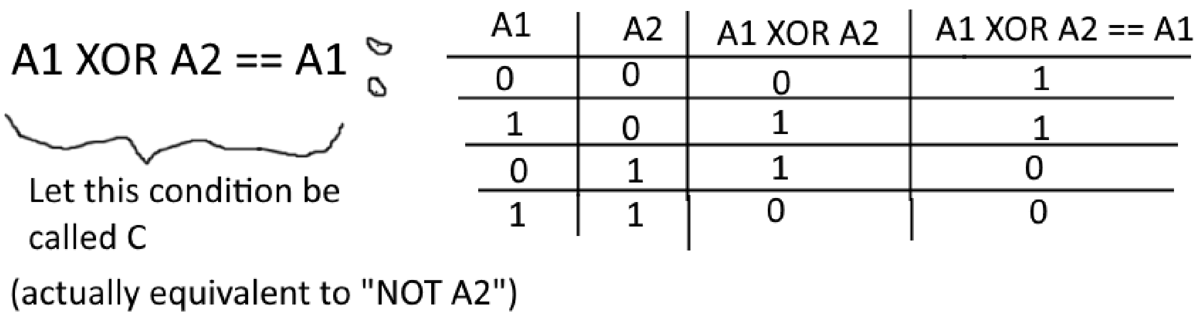
Finite State Machines 1

Part A

Sample solution

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).

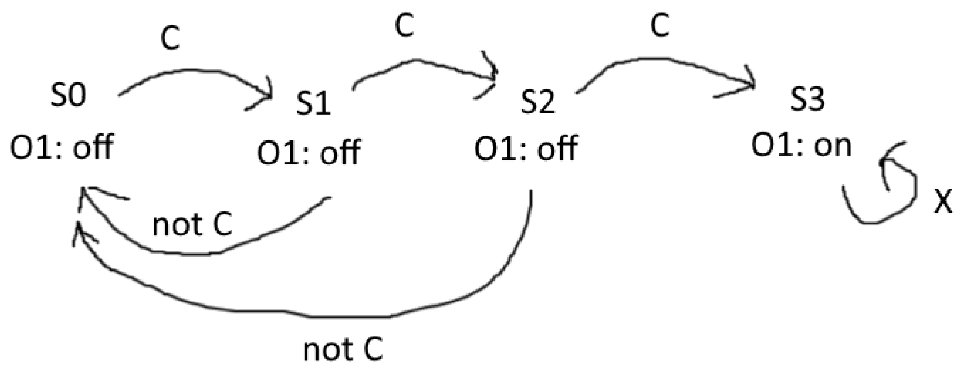
State transition diagrams



$$\underbrace{A1 \text{ XOR } A2 == A1}_{\text{Let this condition be called C}}$$

(actually equivalent to "NOT A2")

A1	A2	A1 XOR A2	A1 XOR A2 == A1
0	0	0	1
1	0	1	1
0	1	1	0
1	1	0	0



$$\begin{tikzcd}
\begin{array}{c}S0\\ O1: \text{off}\end{array} \arrow[r, bend left, "C"] & \begin{array}{c}S1\\ O1: \text{off}\end{array} \arrow[r, bend left, "C"] \arrow[l, bend left, "\text{not } C"] & \begin{array}{c}S2\\ O1: \text{off}\end{array} \arrow[r, bend left, "C"] \arrow[ll, bend left=50, "\text{not } C"] & \begin{array}{c}S3\\ O1: \text{on}\end{array} \arrow[loop right, "X"]
\end{tikzcd}$$

Note: the text wasn't very clear about whether A1 XOR A2 == A1 needs to be true for 3 CONSECUTIVE cycles and students have done both (consecutive and non-consecutive). I'll subtract 1 mark for the latter.

Encoded state transition table

Apply the following encodings for states:

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

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 as N0 and N1.

The encoded state transition tables are then as below. We write it in terms of the variable "C = A1 XOR A2 == A1" for simplicity, but one can easily write it in terms of A1 and A2 as well (it just leads to a longer table).

Current state		**Inputs**		**Next state**	
S1	**S0**	**Not C**	**C**	**N1**	**N0**
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	0	0
1	0	0	1	1	1
1	1	X	X	1	1

(May have also used a one hot encoding if savvy enough)

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.

$$N_0 = S_1' S_0' C + S_1 S_0' C + S_1 S_0 = S_1' S_0' A_2' + S_1 S_0' A_2' + S_1 S_0$$

$$N_1 = S_1' S_0 C + S_1 S_0' C + S_1 S_0 = S_1' S_0 A_2' + S_1 S_0' A_2' + S_1 S_0$$

Output equations

Output is 1 if and only if state is 11, so the output equation is:

$$\text{Output} = S_0 S_1$$

Rubric

Total marks: 15

Mark breakdown:

- State diagrams (5 marks)
- Encoded next state transition table (5 marks)
    - Without encodings: -2 marks
- Next state equation (4 marks)
- Output equation (1 mark)

    - Zero if it depends on inputs (this is a **Moore** machine)

Notes:

- Mistakes in earlier parts can easily affect answers in later parts, so in many cases you will not be able to mark this question by direct comparison to the sample solution. Deduct marks for a mistake in only one location, and any correct steps in later parts based on incorrect results should still be awarded marks.
- Common mistakes:
    - Using a 2-bit encoding in the encoded state transition table, but then changing to a one-hot encoding when writing next state equations
        - -3 marks
    - Omitting the state transition diagrams. I've been lenient on this if I can tell what your diagram would've been from the state transition table, but be warned!
    - Mistaking the condition as A1 XOR A2 rather than A1 XOR A2 == A1 (i.e. forgetting the "equals A1" part)
        - -1 mark
    - Not returning to S0 if the streak is broken
        - Not sure if the question was clear enough here, but -1 mark
    - Flipping it around, i.e. seeming to advance to next state if A1 XOR A2 != A1, or otherwise messing up what the condition to advance is
        - -2 marks

Part B

Sample solution

Sequencing overhead (setup time before writing to registers and delay when reading from registers) increases the critical path time and hence increases the required clock period, decreasing the required clock frequency.

Rubric

Total marks: 3

Mark breakdown:
- Reasoning that clock period must be increased (2 marks)
- Hence clock frequency would be DECREASED (1 mark)

Finite State Machines 2

This question is similar to Finite State Machines 2 on the main exam, except there is one less pipeline register (none in front of the output).

The clock period is at least the length of the longest path from a register output to a register input or the output. Stage 1 is the longest stage, and takes total time 0.4 + 4.7 + 0.1 = 5.2 ns (adding times for reading and writing the relevant pipeline registers), so this is the clock period. Then the number of cycles taken in each case is:

A. 15 cycles (3 to fill up pipeline, 12 to process tokens) => 15 * 5.2 = 78 ns time
B. 1 billion + 3 cycles (3 to fill up pipeline, 1 B to process tokens) => 1,000,000,003 * 5.2 ns = 5.2 s + 15.6 ns
C. 0.25 billion + 3 cycles (3 to fill up each PPU's pipeline, 250 M to process the 250 tokens assigned to each PPU) => 250,000,003 * 5.2 ns = 1.3 s + 15.6 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 in the first register. One can also argue that the final token takes less time to reach the output in the final cycle because it's just the propagation delay of Stage 3.

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

Instruction Set Architecture 1

Part A

Sample solution (edited from a student's solution)

```
// QuAC Assembly Iterative Factorial Function
// Registers:
// r1: n (input argument)
// r2: i (loop counter)
// r3: acc (accumulator for result)
// r6: sp (stack pointer)
// r8: lr (link register)

// Caller-saved registers: r1, r2
// Callee-saved registers: r3

// Factorial function
factorial:
        movl r3, 1        // Initialize accumulator to 1 (result)
        movl r2, 1        // Initialize loop counter to 1

factorial_loop:
        cmp r2, r1        // Compare i with n
        beq factorial_end   // If i == n, exit loop

        push r1            // push n onto the stack
        push r2           // Push i onto the stack

        mov r1, r3
        push lr
        bl mul            // Call mul(x, y) function with x = acc and y = i
        mov r3, r1        // move result to r3 (it is new acc)

        pop r2            // Restore i from the stack
        pop r1            // Restore n from the stack

        add r2, r2, 1     // Increment loop counter
        jmpl factorial_loop // Jump back to the loop start

factorial_end:
        movl r1, r3       // Set the result (acc) in r1 as per the calling convention
        bx lr             // Return by jumping to the link register

// End of Factorial function

// Assume mul function is already defined
```

Rubric

Total marks: 20

Marks to give for each required part of the answer:

- If and for statements are translated correctly, and basic algorithm is correct (10 marks)
- Nested function call is handled correctly:
  - There is a comment indicating callee and caller saved registers (3 marks)
  - This convention is respected and used to save registers correctly (7 marks)

If student uses ARM assembly instead of QuAC:
- If this is very minor, i.e. they just forget to change the names of some instructions like b to jp, don't bother deducting marks
- If it is major, e.g. they forget that QuAC only has 4 registers and start using r5-r9, perhaps give a few marks if the basic algorithm is still mostly okay but otherwise give 0.

Part B

Sample solution

Creating it as an instruction requires implementing the algorithm in hardware, which complicates the hardware and requires more transistors, but can allow it to be done in less cycles/time by avoiding the overhead of control flow incurred by software (e.g. the jump instructions needed to implement the FOR loop in software would have incurred overhead) plus making use of the high degree of parallelism possible in hardware.

Rubric

Total marks: 5
Breakdown:

- One valid advantage/argument for implementing the function in software (2.5 marks)
  - Simpler ISA
  - Reduced hardware complexity
  - Flexibility; programmer can modify the algorithm rather than it being fixed
- One valid disadvantage/argument for implementing it as an instruction (2.5 marks)
  - No overhead due to function call and return, and make use of parallelism in hardware; likely faster
  - Reduced code size

Instruction Set Architecture 2

Correct answers:
Uarch: 1, 4, 6, 7, 9, 10, 11, 13, 15
ISA: 2, 3, 5, 8, 12, 14

Input/Output 1

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 mark calculation, 80% of the program is made 1.1 times as fast. This makes the overall program's speedup as below, according to Amdahl's law:

$$\begin{aligned} S &= \frac{1}{(1-p) + p/s} \\ &= \frac{1}{0.2 + 0.8/1.1} \\ &\approx 1.078 \end{aligned}$$

This corresponds to a 7.8% increase for a cost of 4 hours, and hence the cost is 0.51 hours per 1% increase of performance.

For report creation, 20% of the program is made 2 times faster. This makes the overall program's speedup as below:

$$\begin{aligned} S &= \frac{1}{(1-p)+p/s} \\ &= \frac{1}{0.2/2+0.8} \\ &\approx 1.111 \end{aligned}$$

This corresponds to a 11.1% increase for a cost of 6 hours, and hence the cost is 0.54 hours per 1% increase of performance.

Hence working on the mark calculation code is better.

Rubric

Total marks: 20

Mark breakdown:

- Application of Amdahl's law to convert speedup for part of program to speedup for total program (14 marks)
- Conversion of the speedup to a percentage increase (3 marks)
- Subsequent calculation of cost per percentage increase (3 marks)

Mistakes and deductions:

- Corresponding "1.2 times faster" with "120% increase" instead of "20% increase"
    - Award 0/3 for the "Conversion of the speedup to a percentage increase" part (award 15 marks if the rest is correct)
- Not using Amdahl's law
    - If they take note of the fact that CPU usage and disk usage is split 50-50 when comparing the two, but don't consider it when considering the cost per percentage improvement, award 3/18 marks
    - Otherwise, award zero marks for whole question

Input/Output 2

On the software side, you will need to LDR/STR from the memory addresses GPIO_DIR and GPIO_OUT to set the direction bit for the LED and then the output bit for the LED.

On the hardware side, when you do a memory access, the Address Decoder works out what range of the memory map your address falls into and decides which of the units to send your requests to/from. When you LDR, it sets RDsel to 1 to select the GPIO unit to be read from. When you STR, it sets WEM and WE2 to 0 so that the value you're storing doesn't affect the actual memory nor other I/O units, while setting WE1 to 1 so that the GPIO unit receives the STR instruction and can act accordingly.

Rubric

- On software side you do a LDR/STR, looks like a memory access (5 marks)
- On hardware side, the WE signals are configured to actually send the write to the I/O unit rather than the memory unit (5 marks)

Microarchitecture 1

Correct answers: 1, 2, 3, 4, 5

Microarchitecture 2

Part A

Sample solution

4 times
- On loop 1, r1 = 2
- On loop 2, r1 = 3
- On loop 3, r1 = 5
- On loop 4, r1 = 8 (and loop breaks as r1 > r2 = 6)

Rubric

Mark black-and-white (full marks if correct, 0 if not) - except in one edge case i saw where a student read it as cmp r1, r3 not r2 but their logic was correct so i gave 2/4 marks

Part B

Sample solution

Data hazards to resolve:
- i8 and i9
- i11 and i12
- i12 and i13

- all of the above need 2 NOPs to resolve (so reader is in RR when writer is in WB)
- there’s also technically a FLAGS dependency between i6 and i7 but tbh don’t bother abt this

Control hazards to resolve:
- i7
- i10
- the above need 4 NOPs to resolve (so branch instruction has completed WB before next instruction is fetched)
  - 3 NOPs is technically not correct but it’s extremely subtle why so dont bother subtracting marks

IPC calculation:

- Number of instructions: 5 instrs pre-loop + 3 * 5 instrs during loop + 2 instrs when exiting loop + 3 instrs after exit = 25
- Number of cycles: 4 cycles to fill pipeline + 5 cycles pre-loop + 3 * (5 + 4 + 4 + 2 cycles loop) + (2 + 4) cycles when exiting loop + (3 + 2 + 2) cycles after exit = 67
- Ratio: 0.373

Rubric

Total: 20 marks

Breakdown:

- Identify all data hazards (5 marks)
- Add correct number of NOPs to fix data hazards (2 marks)
- Identify all control hazards (5 marks)
- Add correct number of NOPs to fix control hazards (2 marks)
- IPC calculation
  - Basic structure/logic of calculation is correct (3 marks)
  - Includes cycles spent filling pipeline at the beginning as stalls (1 mark)
  - Otherwise correct calculation (2 marks)

Part C

Sample solution

Example where forwarding resolves data hazard: i8 and i9
Example where it doesn’t due to load use hazard: i12 and i13

New IPC:

- Only one NOP due to load use hazard now, i12 and i13
- For untaken conditional branches, branch is predicted correctly so no penalty
- For the taken conditional branch when the loop breaks, branch misprediction carries 2 cycles worth penalty

- Unconditional branches cause 4 cycles penalty/4 NOPs

IPC calculation:

- Number of instructions: 5 instrs pre-loop + 3 * 5 instrs during loop + 2 instrs when exiting loop + 3 instrs after exit = 25
- Number of cycles: 4 cycles to fill pipeline + 5 cycles pre-loop + 3 * (5 + 4 cycles loop) + (2 + 2) cycles when exiting loop + (3 + 1) cycles after exit = 44
- IPC: 0.568

Rubric

Total marks: 20

Mark breakdown:

- Example where forwarding solves data hazard (5 marks)
- Example where it doesn't (5 marks)
- IPC calculations:
    - Considering the control hazard penalties correctly (4 marks)
    - IPC calculation itself
        - Basic structure (3 marks)
        - Consider cycles spent filling pipeline (1 mark)
        - Correct (2 marks)

Part D

Logic pretty much identical to the analogous question from main final, except the loop executes 3 times. Accuracy for 1 execution will be 25%, for 4 executions will be 62.5%.

- Correctly determining the pattern of taken/untaken branches in program (1 mark)
- Correctly considering how the Smith predictor's counter changes throughout execution (2 marks)
- Correct final answer for accuracy for
    - 1 execution of the code (1 mark)
    - 4 executions of the code (1 mark)

Common mistakes:

- Discussing 1 and 4 iterations of the loop and not the program
- Swapping "taken" and "not taken" around, despite getting the rest of the logic correct
    - Decided to award both of these two 2.5/5 marks since the student still demonstrates SOME understanding in these cases

Microarchitecture 3

A. Dependencies are:
    a. {i1 to i2, true, by way of r1}

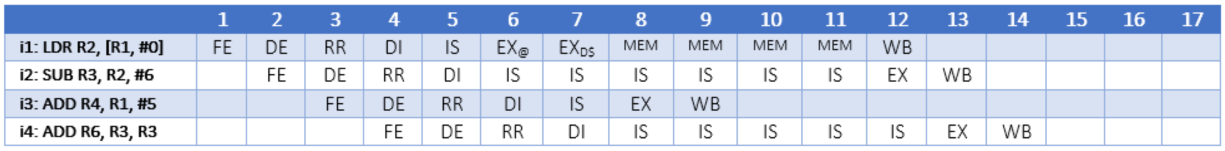
        b. {i2 to i4, true, by way of r2}
        c. {i3 to i5, true, by way of r1}
        d. {i5 to i6, true, by way of r2}
        e. {i3 to i2, anti, by way of r2}
        f. {i5 to i4, anti, by way of r2}
        g. {i3 to i1, output, by way of r1}
        h. {i5 to i2, output, by way of r2}

B. Independent chains are {i1, i2, i4} and {i3, i5, i6}
C. Two at most.

Microarchitecture 4

Part A

Pipeline cycle-by-cycle diagram:



	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: SUB R3, R2, #6		FE	DE	RR	DI	IS	IS	IS	IS	IS	IS	EX	WB				
i3: ADD R4, R1, #5			FE	DE	RR	DI	IS	EX	WB								
i4: ADD R6, R3, R3				FE	DE	RR	DI	IS	IS	IS	IS	IS	EX	WB			

Issue queue contents:

End of cycle #8:

{entry 1: v=1, dst tag = r3, rs1 rdy = 0, rs1 tag/val = r2, rs2 rdy = 1, rs2 tag/val = 6}
{entry 2: v=1, dst tag = r6, rs1 rdy = 0, rs1 tag/val = r3, rs2 rdy = 0, rs2 tag/val = r3}
{entry 3: v=0, dst tag = r4, rs1 rdy = 1, rs1 tag/val = 7, rs2 rdy = 1, rs2 tag/val = 5}

End of cycle #12:

{entry 1: v=0, dst tag = r3, rs1 rdy = 1, rs1 tag/val = 20, rs2 rdy = 1, rs2 tag/val = 6}
{entry 2: v=0, dst tag = r6, rs1 rdy = 1, rs1 tag/val = 14, rs2 rdy = 1, rs2 tag/val = 14}
{entry 3: v=0, dst tag = r4, rs1 rdy = 1, rs1 tag/val = 7, rs2 rdy = 1, rs2 tag/val = 5}

Notes:
- Some rows are uninitialised at the beginning; these can have anything one wants in them
- Between cycle 8 and cycle 12, the issue queue entries should be consistent (so entries that have since become invalid because the relevant instruction was issues should still have the contents at the time of issue)

Part B

Cycle #5 scoreboard: 11000111
Cycle #9 scoreboard: 11001101

Part C

{R0 = 8, R1 = 7, R2 = 20, R3 = 5, R4 = 12, R5 = 3, R6 = 2, R7 = 1}

Part D

R4, 12

Rubric

Copied from main exam:

Part A:

Total marks: 20

Mark breakdown:

- Pipeline cycle by cycle breakdown is correct (6 marks)
- Issue queue entries at cycle #8 (7 marks)
- Issue queue entries at cycle #12 (7 marks)
- Subtract 2 marks per one cycle off error

Part B:

Total marks: 15

Mark breakdown:

- Correct scoreboard at cycle #5 (9 marks)
- Correct scoreboard at cycle #9
    - r7 is no longer valid due to i4 passing register read (3 marks)
    - i2 completed writeback, making r4 valid again (3 marks)

Part C:

Total marks: 8

Mark breakdown:

- The results of i1 and i2 are visible in ARF and correct (4 marks)
- I3 and i4 haven't done WB yet so their results are NOT visible (4 marks)

Part D:

Total marks: 6

Bonus

Sample solution

Part 1: cannot deal with WAR/WAW hazards

Consider this example:

```
I1: ldr r2, [r6]
i2: add r1, r2, r3
i3: add r1, r4, r5
```

i3 here can theoretically be scheduled before i2 as the instructions are not actually dependent. However, if this happens on the CDC scoreboard, then i3 will write to r1 before i2 writes to it, and when i2 finally completes it overwrites the value that was in r1 written by i3. Hence, any subsequent instruction, i.e. i4, will read the value of r1 written by i2, even though the program requires that it read the value written by i3 (which comes AFTER i2 and so should decide the latest value of r1). This breaks program correctness. (5 marks)

This is fixed on ARF+ROB by renaming the registers to have different names, and delaying actually modifying the register file to when the instructions are *committed*, which happens in order. I3, upon completion, will write its result to the ROB under a different name like ROB0, but R1 will not be modified immediately, but instead when i3 will commit. As commits must happen in program order, this means i3 will update r1 after i2 does, maintaining program correctness (5 marks)

Part 2: cannot deal with mis-speculation

Consider this example:

```
i1: ldr r1, [r0]
i2: beq exit
i3: add r2, r2
Exit label:
…
```

For CDC scoreboard, it may speculate the branch i2 and guess that it is not taken, and so fetch the instruction i3 and execute it instead. Further, i3 may finish and leave the CPU before the branch resolves. However, once the branch does resolve in this case, it may have been found mis-speculated and the next instruction to execute is not i3. At this point, i3 has already updated the register file (in writeback) and left the CPU so there is no record to tell what it did or mechanism to reverse i3's actions. (5 marks)

This is resolved again via delayed commit in ARF+ROB, as i3 cannot commit its result to the register file until i2 has committed. When i2 completes it will be found mis-speculated, after which we can discard all instructions to be committed after i2 and so no harm is done to the actual register file. (5 marks)

Rubric

Embedded in the sample solution. This question was marked relatively relaxedly because it’s quite difficult, so even if a student didn’t provide an explicit example they weren’t marked down.

Notes:

- For false dependencies, the few people who answered mostly said something like “CDC will refuse to schedule instructions with false dependencies in parallel” which is… true, but the question wants you to think about WHY this is, and what would happen if you were to actually schedule instructions with false dependencies in parallel with CDC.
    - The question should’ve made it explicit you should discuss what happens if instructions with false dependencies are issued in parallel or out of order.
    - Decided to just give these ones 2.5 marks even if it’s partly our fault