EECS 470 *Midterm Exam - Solutions*

Fall 2025

Name: _____SOLUTIONS_____ Uniqname: _____

Sign the honor code:

I have neither given nor received aid on this exam nor observed anyone else doing so.

_____

#	Points
1	**/ 15**
2	**/ 18**
3	**/ 13**
4	**/ 17**
5	**/ 16**
6	**/ 10**
7	**/11**
Total	**/100**

NOTES:

- Closed book and one 8.5x11 page of notes
- Calculators are allowed, but no smart watches, cell phones, earbuds, etc.
- Don't spend too much time on any one problem.
- You have about 120 minutes for this exam.
- There are **16** pages including this one.
- **Be sure to show your work and explain what you've done where appropriate.**

1) Short Answer - Fill in the correct bubbles or blank [15 Points]:

1.1) In R10k, which of the following structures can change states from the dispatch of a new instruction? (Select all that apply)

◯ Map Table ◯ Arch. Map Table ◯ Freelist ◯ Reservation Station

1.2) In Tomasulo's Algorithm, which hazards are circumvented by design? (Select all that apply)

◯ RAW ◯ WAW ◯ WAR

1.3) In Scoreboarding, an instruction stalls in the Dispatch stage for _____ hazards (select all that apply).

◯ RAW
◯ RAR
◯ WAW
◯ WAR
◯ Structural
◯ None of the above

1.4) In a P6-style architecture, when an instruction completes execution, to which of the following is the resulting **value** broadcasted over the CDB? (Select all that apply)

◯ Reservation station
◯ Register file
◯ Map table
◯ Reorder buffer
◯ Functional units
◯ Only the tag is broadcasted, not the value

1.5) Let us assume the EX stage from the 5 stage in-order pipeline was split into two stages, EX1 and EX2, to break up a critical path that was discovered. How many cycles, if any, would instruction 3 stall in the below program?

```
1:    lw x1, 5(x3)     // x1 = mem[x3 + 5]
2:    li x2, 2         // x2 = 2
3:    addi x1, x1, 7   // x1 = x1 + 7
```

◯ 0 Cycles ◯ 1 Cycle ◯ 2 Cycles ◯ 3 Cycles

Either choice, depending on your assumption:

```
Assuming only internal forwarding from RF:
I1 = F D E E M W
i2 =   F D E E M W
i3 =     F * * D E E M W
```

```
Assuming all forwarding paths of P3:
 I1 = F D E E M W
 i2 =   F D E E M W
 i3 =     F * D E E M W
```

1.6) If asked to evaluate a suite of benchmarks and report on the average memory bandwidth as measured in MB/s, which technique would you use?

◯ Geometric Mean ◯ Harmonic Mean ◯ Arithmetic Mean

1.7) Which of the following instructions can not cause a WAW hazard. (Select all that apply)

◯ Branch
◯ Add
◯ Load
◯ Store
◯ NOP
◯ Multiply

As we intended the question, we were looking for WAW hazards in the register dataflow portion of the machine. Here are a few caveats, which meant we didn't penalize for these incorrect answers because our question was too vague:

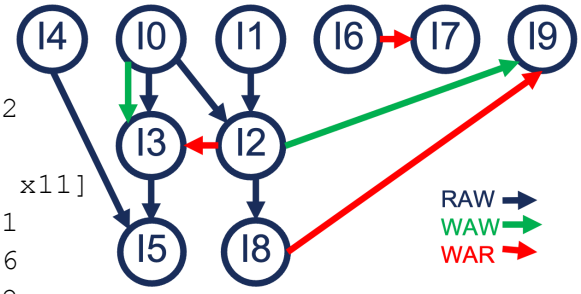
Branch = In some ISA's, like ARM v7, the PC is a general purpose register (R15). So you could have WAW in that ISA on a Branch.

Store = If you assumed WAW in the memory system, you would be correct as we didn't specify not to consider that. Also, some ISA's, like ARM v7, you can have the store instruction update the base_address to contain base_address+offset at the end of the instruction. This means in that architecture a Store can cause a WAW. In class we focused on Store not having a destination, so I hope you selected it though.

2) It all Depends [18 Points]:

Consider the following snippet from a RISC-V assembly program (destination registers are specified first and all registers are initialized to some known value):

```
I0: addi x1, x0, 5     //x1 = x0 + 5
I1: addi x2, x0, 7     //x2 = x0 + 7
I2: add  x3, x1, x2    //x3 = x1 + x2
I3: addi x1, x1, 1     //x1 = x1 + 1
I4: lw   x4, 0(x11)    //x4 = mem[0+ x11]
I5: add  x7, x4, x1    //x7 = x4 + x1
I6: add  x9, x5, x6    //x9 = x5 + x6
I7: addi x5, x0, 42    //x5 = x0 + 42
I8: sw   x3, 0(x10)    //mem[0 + x10] = x3
I9: addi x3, x0, 0     //x3 = x0 + 0
```



2.1) How many **true (RAW)** and **false (WAW, WAR)** dependencies are present in the above program? **[6 Points]**

RAW: _____6_____ WAW: _____2_____ WAR: _____3_____

2.2) If our pipeline's integer ALUs take a single cycle, and the **pipelined** load/store functional units takes 3 cycles (includes address calculation), what is the **minimum number of cycles** needed to execute the above program on an out-of-order machine with infinite issue width, ALUs and load/store queues assuming perfect renaming? Why? **[4 Points]**

Clarification during the exam: infinite ALUs, load-store units

5 cycles – the longest true dep. chain is (I0 || I1)→I2→I8 which is 1+1+3 = 5 cycles

The problem makes no assumptions about the architecture of the system – we instead consider an "ideal" system which is limited only by the data-flow dependencies imposed by RAW hazards (minimum number of cycles for execution on ANY hypothetical OoO architecture).

Note, the problem assumes an infinite system, so there are no in-order pipeline stalls.

2.3) Suppose the load/store unit were not pipelined (it cannot start a new memory op until the previous one finishes). How, if at all, would this change your answer to part (2.2), and why? **[4 Points]**

Clarification during the exam: We have a SINGLE non-pipelined load-store unit, still infinite ALUs

Now we need 6 cycles, because the sw (I8) must wait for the lw (I4) to finish execution before it goes to the LSU, making it a minimum of 3 + 3 = 6 cycles. This is essentially a new HAZARD edge in the graph from I4 -> I8.

2.4) If this program were to run on the spec-compliant 5-stage VeriSimpleV pipeline you implemented in Project 3, how many **total values** would be forwarded from a producer to a consumer throughout program execution (do not include internal forwarding through the register file)? **[4 Points]**

3 values will be forwarded

- I0 to I2 (x1)
- I1 to I2 (x2)
- I4 to I5 (x4)

I3 to I5 is NOT a forwarding condition because of the load-use hazard caused by I4 – I3's dest will be forwarded to I5 through the register file!

3) Power and Performance [13 Points]:

3.1) Your company is trying to run the oogabooga program, which has 80% parallel code and takes 10s to execute on 1 core. Assume parallelism performance scales linearly and oogabooga can spawn a sufficient number of threads. Oogabooga can either be run on your office desktop with 6 cores or the company server with 124 cores. The desktop would use 80W to run oogabooga, but the server would use 360W.

Which is more efficient if the metric we consider is Ws (Watt-seconds). **[5 Points]**

Compute time on system:

6_coreT = $10 * (0.2 + \frac{0.8}{6}) = 3.33333$

124_coreT = $10 * (0.2 + \frac{0.8}{124}) = 2.06451$

Efficiency:

6_coreWs = $80 * 3.33333 = 266.6664$

124_coreWs = $360 * 2.06451 = 743.2236$

6 core is better

3.2) The company contracted Fix-it Felix to decrease the server's power usage. Felix only knows voltage scaling, so he decreased the operating core voltage from 1.3V to 0.9V. Evidently, this has a performance impact. Assume frequency scales linearly with voltage and performance scales linearly with frequency. What is the new energy (Ws) usage for the server? Would your answer for (3.1) change? **[5 Points]**

Take DVFS: $P = \frac{1}{2}*C*V^2*A*f = M*V^3$ where $M = \frac{1}{2}*C*A*n$; $f = V*n$

$$\frac{P_{new}}{P_{old}} = \frac{0.9^3}{1.3^3} = 0.33182$$

$$\frac{f_{old}}{f_{new}} = \frac{1.3}{0.9} = 1.44444$$

$$Ws_{new} = \left(\frac{f_{old}}{f_{new}} \times 124_{coreT}\right)\left(\frac{P_{new}}{P_{old}} \times 360W\right) = 2.98206 \times 119.4552 = 356.222573712$$

Or by recognizing that $\frac{f_{old}}{f_{new}} \times \frac{P_{new}}{P_{old}} = \frac{0.9^2}{1.3^2}$ then:

$$Ws_{new} = \left(\frac{0.9}{1.3}\right)^2 \times 743.2236 = 356.222573712$$

No, your answer should stay the same (still 6-core)

3.3) Meanwhile, a new startup Danube is attempting to determine the sizing of an LLM accelerator + GPU system. The machine they are using consists of 3 major phases. In phase one, the input prompt is tokenized, which is going to run on the standard in-order processor. The second phase is a KV cache step, which can be parallelized by a factor of 10 using Danube's accelerator. The third phase is the computational phase, and can be trivially parallelized on the GPU. If their initial measurements show that the three phases constitute 10%, 30%, and 60% respectively for the tokenize, KV cache, and computational phases. What speedup must the GPU achieve if they wish to improve the entire system performance by 5x? **[3 Points]**

Ahmdahl's Law:

Speedup = 1 / (serial% + parallel%_A / speedup_A + parallel%_B / Speedup_B)

So, we get:

5 = 1 / (.1 + .3/10 + .6/X)

⅕ = .13 + .6/X

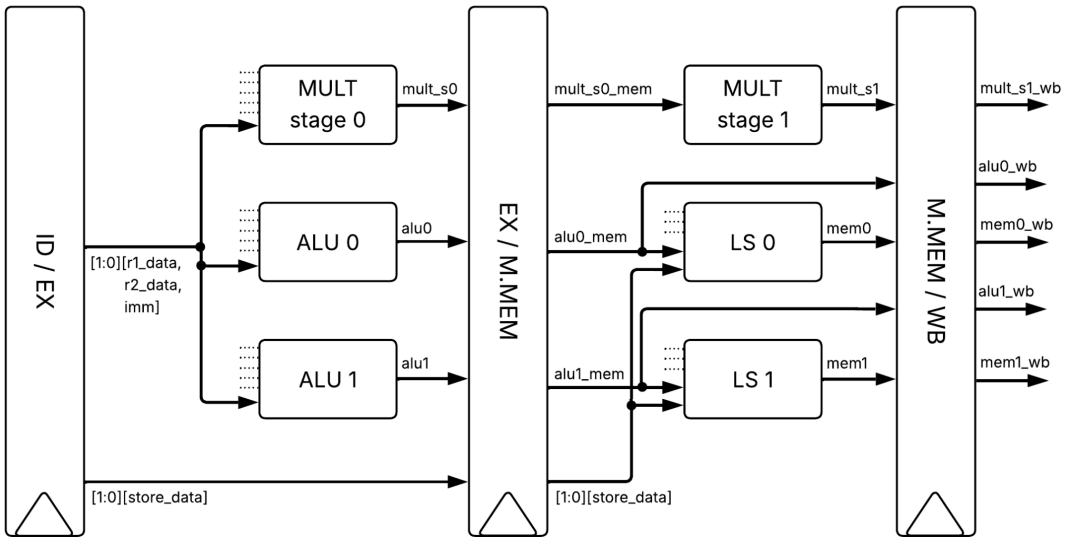
.07 = .6/X

X = .6/.07

X = 8.571

4) In Order Pipeline [17 Points]:

An up-and-coming chip manufacturer, Schmadvanced Schmicro Schmevices, has recruited you to finish their 5-stage processor! They heard the word *superscalar* and ran with it, adding-on another ALU to the EX stage, and another LS to the MEM stage! To make logic simpler, instructions going to LS 0 must calculate their address in ALU 0, and similarly for LS 1 with ALU 1. After testing their design, they found the Multiplier was holding them back from defeating their rival startup—Intel But Better Inc.—so they split the MULT unit into 2 stages, the first being in EX, and second in MEM (now M.MEM). Their frequency is blistering fast and they're so close, but there's no forwarding! A diagram of the relevant stages and connections is provided below.



4.1) Schmadvanced Schmicro Schmevices is asking you to complete the forwarding paths for them, as they're stuck on which units should really be connected. Using the output signals of the pipeline stages, and the Functional Units (FU) as inputs (assume r1 and r2 of an FU are both supported with the same path), what are all the **forwarding** paths that should exist? One of the rows is correctly filled for you. **[9 Points]**

Output \ Input	MULT s0	ALU 0	ALU 1	MULT s1	LS 0	LS 1
mult_s0_mem	○	○	○	○	○	○
alu0_mem	●	●	●	○	○	○
alu1_mem	○	○	○	○	○	○
mult_s1_wb	○	○	○	○	○	○
alu0_wb	○	○	○	○	○	○
alu1_wb	○	○	○	○	○	○
mem0_wb	○	○	○	○	○	○
mem1_wb	○	○	○	○	○	○

Output \ Input	MULT s0	ALU 0	ALU 1	MULT s1	LS 0	LS 1
mult_s0_mem	○	○	○	🟡	○	○
alu0_mem	●	●	●	○	🟡	🔵
alu1_mem	🔴	🔴	🔴	○	🔵	🟡
mult_s1_wb	🔴	🔴	🔴	○	🔴	🔴
alu0_wb	🔴	🔴	🔴	○	🔴	🔴
alu1_wb	🔴	🔴	🔴	○	🔴	🔴
mem0_wb	🔴	🔴	🔴	○	🔴	🔴
mem1_wb	🔴	🔴	🔴	○	🔴	🔴

● was given
🔴 should be marked
🟡 should not be marked, but is a path that exists, so leniency is applied
🔵 error in provided row; these should exist (as standard non-forwarding paths) but could've misled the students and should not count towards the grade whether marked or not.

Every FU and LS must acquire data from other FUs and LSs, and maintain proper forwarding abilities in the 5-stage pipeline. As a result, results from ALUs must be present in both EX/MEM.M and in MEM.M/WB, and both must forward to all units.
No forwarding should exist from mult_s0_mem or to MULT s1, as those are explicitly meant for the 2-stage multiplier. No assumptions can be made that the 2-stage has the ability to process smaller numbers in less stages, nor that the value from mult_s0_mem has any merit to program flow (i.e. its value is useful to other instructions) except for finishing the multiply instruction.

4.2) What new data hazards does the Schmadvanced Schmicro Schmevices processor have to check for? If none, justify why not. **[4 Points]**

New hazards that need to be checked for (Since there were multiple, mentioning any one of these warranted most of the points):

-RAW from the new mult stage. Same logic as the MEM ops, where a NOP needs to be used in-between consecutive RAW instructions.

-If a store is at least one of two memory instructions in the EX stage which also shares the target address with the other memory instruction, then in-orderness must be preserved and a stall (NOP) in-place of the younger memory operation must be used.

-There is a WAW dependency that is created between two consecutive instructions if they share destination registers.

Optimization hazard:

-A store that does not have data ready can compute said data in parallel to its address in the EX stage, if the data source is an ALU operation and it creates a consecutive RAW dependency. Otherwise, you would have to stall.

Mentioning RAW stalling because of the superscalar width did not warrant points as this question asked for **new** hazards, as the design was changed with forwarding paths.

4.3) The Physical Design team at Schmadvanced Schmicro Schmevices has asked you to make the Functional Unit forwarding-input logic simpler as they currently take up too much area. You still need all the paths to not incur unnecessary structural stalls. How could you redesign the forwarding path inputs so every path does not directly connect to every functional unit?**[4 Points]**

For this question, anything relating to selecting forwarding paths and redirecting warranted points, but there are some important details that had to be considered:

-Just doing selection in front of every FU does not change the design. The FU is already doing this inherently, and this does not change the footprint of the design.

-Selection in front of all of the FUs (or LSs) in their respective stages with *one* unit is the main idea, as the design now benefits from there being only 2 instructions executing per cycle (so only 2x(r1,r2) selections needed at most) and redistributes to 3 FUs. This means we saved on the footprint of an additional unnecessary selector, with a small cost of redistributing to 2 or 3 FUs. Same goes for LSs, but since the saving there is negligible it's disregarded.

-Forwarding to a pipeline stage (i.e. ID/EX) or register-based buffer results is a valid approach, but requires explicit mentioning that the path forwarded from is no longer another pipeline stage register (i.e. EX/MEM.M) and is actually an FU direct output (i.e. alu0). This is the only way you would not incur a cycle gap for forwarded data, which creates its own hazards.

5) Straight Outta Registers [16 Points]

In Tomasulo-inspired architectures, we eliminate false dependencies via **register renaming** in order to improve our ability to execute instructions in parallel. In 2018, Irie et al. proposed STRAIGHT, a novel architecture that eliminates false dependencies without register renaming (in order to improve performance and power consumption).

Instead of renaming registers, the STRAIGHT ISA specifies source operands by **control-flow distance to the producer**. Encodings and a sample program for the STRAIGHT ISA are shown below.

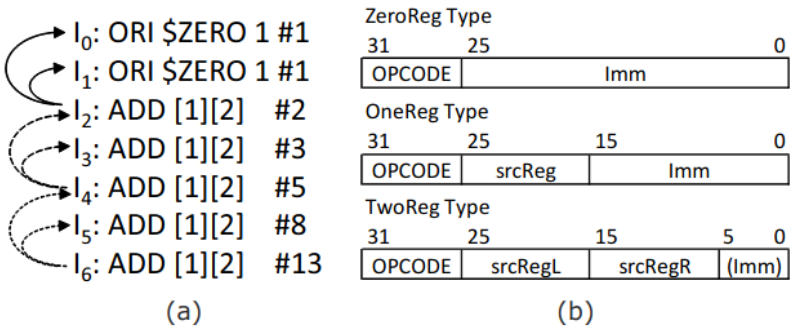


Fig. 1. Example of STRAIGHT code and bit-field formats

The example program computes a short Fibonacci sequence – each add instruction takes as operands the results of the previous two instructions, specified by [1] and [2], respectively. Source operands are encoded into the instruction in the *srcReg*, *srcRegL*, or *srcRegR* fields, depending on the instruction type.

In order to index the physical register file, a register pointer (RP) is introduced in the hardware. Incremented for each instruction, the RP value provides an instruction's destination register number. Source register numbers are obtained by subtracting the distance from the RP.

5.1) Briefly explain why the STRAIGHT inter-instruction distance technique eliminates WAR and WAW hazards. **[4 Points]**

Destination register is determined by fetch order, so by definition no two instructions share the same destination register. That is, every "write event" is distinct. This eliminates both WAW and WAR hazards.

5.2) Given the instruction encodings above, what limitation is imposed on the placement of instructions producer-consumer pairs in a STRAIGHT program? **[4 Points]**

Source registers in STRAIGHT are specified using 10-bit fields, so there is an absolute maximum inter-instruction distance of 2^10 - 1 = 1023. Answering 1024 is also ok, though in the paper, they start counting at 1.

Answers related to the size of the physical register file are microarchitectural limitations, not limitations inherent to the STRAIGHT ISA.

5.3) In a STRAIGHT microarchitecture, how large must the physical register file be in order to guarantee that there will be no RP aliasing (which could lead to unexpected overwriting of values)? **[4 Points]**

MAX_RP = (maximum distance between instructions + ROB size)

Old register values are only safe to overwrite after no later instruction will ever read them, i.e. after all consumers (possibly hundreds of dynamic instructions later) have retired.

So, to guarantee safety, PRF must be able to accommodate all instructions that could still be in flight, plus all values that might still be live due to the maximum distance between producer and consumer, i.e. max control flow distance + rob size

This is a similar principle to R10k, where we need PRF_SZ >= ROB_SZ + #Arch Regs

5.4) Identify one additional piece of information that must be restored on a branch misprediction, exception, etc., as compared to a traditional R10k architecture.**[4 Points]**

The register pointer (RP). Other valid pieces of information are also acceptable.

6) SystemVerilog Design [10 Points]:
You are given a register file module (with 2 read ports and 2 write ports) containing 16 registers, *without internal forwarding*. Implement a module that uses this register file and forwards writes to reads that are accessing the written register in the same clock cycle. *You may assume that the system will never set both write indexes to the same register.*

The provided module without internal forwarding is named **`regfile`** and has the exact same inputs and outputs (with the same names), as **`forwarded_regfile`** below.

```
`define NUM_REGS 16

typedef logic [31:0] reg_t;

module forwarded_regfile(
    // Clock and reset
    input  clock, reset;
    // Enable inputs
    input  [1:0] w_en, r_en;
    // Write and read indexes
    input  [1:0][$clog2(NUM_REGS)-1:0] w_idx, r_idx;
    // Write and read data
    input  [1:0] reg_t w_data;
    output [1:0] reg_t r_data;
);
===== Solution below ======
    logic [1:0] reg_t rf_r_data; // r_data from regfile

    regfile rf(
        .clock(clock), .reset(reset),
        .w_en(w_en), .r_en(r_en),
        .w_idx(w_idx), .r_idx(r_idx),
        .w_data(w_data), .r_data(rf_r_data)
    );

    always_comb begin
        r_data = rf_r_data;
        for (int i = 0; i < 2; i++) begin
            for (int j = 0; j < 2; j++) begin
                if (r_en[i] && w_en[j] && r_idx[i] == w_idx[j]) begin
                    r_data[i] = w_data[j];
                end
            end
        end
    end
===== Solution above ======
endmodule
```

Common errors:

- Using sequential logic in forwarding. Even though the provided regfile module is a sequential module, the outer forwarding module should only be setting the r_data output combinationally using a mux.
- All the signals (w_en, r_en, w_idx, r_idx, w_data, r_data) are treated as though they have scalar width (width 1).
- Only 2 combinations of w_idx[i] to r_idx[j] out of the 4 are checked.
- w_en is not checked when comparing indexes. This can cause a wrong value from w_data to be forwarded to the r_data if an invalid w_idx is compared to a r_idx. Checking r_en is optional.
- Trying to 'overwrite' the outputted r_data from the module in the forwarding case which causes r_data to have multiple drivers.
- Module instantiations in always blocks. A module cannot be called like a software function. It needs to be instantiated once externally and its outputs are to be used after manipulating its inputs in always blocks or assign statements.
- assign statements in always blocks (minor).
- Implement a regfile instead of using the provided module (not much deducted for this).

7) R10K and P6 Walkthrough [11 Points]

On the next pages, you will find a set of charts showing a snapshot of a MIPS R10k-like microarchitecture after four cycles executing a sequence of instructions. You must advance this machine 2 additional clock cycles (to the end of cycle #6). Use the state tables to record the contents of each hardware structure at the end of the sixth clock cycle.

Assume the following:

- Assume the machine is a 2-wide superscalar (e.g., it can issue, complete, and retire at most 2 instructions per cycle). If there are conflicts among instructions, the machine always selects the oldest instructions first.
- Ignore the fetch stage. Assume all instructions have been fetched and are ready for dispatch whenever the out-of-order core allows.
- This machine has 4 architectural registers, a 5-entry ROB, 4 reservation stations, and 8 physical registers.
- There are **2 add** functional units with a 1-cycle latency, and **1 fully-pipelined multiply** functional unit with a **2-cycle** latency (fully-pipelined means the multiply unit has 2 pipeline stages; it can issue a new multiply each cycle, however, multiplies take 2 cycles to execute). FU Structural hazards are detected/handled at the Issue stage.
- Assume there is no bypassing in the X stage, but C will bypass to S through the physical register file.
- Assume reservation stations are freed at the same place as in lecture, and can be reused as soon as they are freed.
- This is the code that is being executed:
    - (1) R1=R1*R3
    - (2) R4=R1*10
    - (3) R1=R1*R2
    - (4) R2=R2+5
    - (5) R1=R3+R2
    - (6) R1=R1*R3
    - (7) R4=R4*10
    - (8) R1=R3+R2

Be sure to update the head and tail pointers correctly to show when instructions are dispatched and retired.

Complete Cycles 5-6 for the R10K-style Machine **[11 Points]**

R10K Cycle # 4

ROB							
ht	#	Insn	T	Told	S	X	C
h	1	I1: R1 = R1 * R3	p5	p1	c2	c3+	
	2	I2: R4 = R1 * 10	p6	p4			
	3	I3: R1 = R1 * R2	p7	p5			
t	4	I4: R2 = R2 + 5	p8	p2	c3	c4	
	5	I5: R1 = R3 + R2					

Map Table	
Reg	T+
r1	p7
r2	P8
r3	P3+
r4	p6

Arch Map	
Reg	T
r1	p1
r2	p2
r3	p3
r4	p4

Reservation Stations					
#	busy	op	T	T1	T2
1					
2	Y	I2	p6	p5	
3	Y	I3	p7	p5	p2+
4					

CDB
T

Free List

R10K Cycle # 5_Sol

ROB							
ht	#	Insn	T	Told	S	X	C
h	1	I1: R1 = R1 * R3	p5	p1	c2	c3+	c5
	2	I2: R4 = R1 * 10	p4	p5	c5		
	3	I3: R1 = R3 * R2	p7	p8			
t	4	I4: R2 = R2 + 5	p3	p2	c3	c4	c5
	5	I5: R1 = R3 + R2					

Map Table	
Reg	T+
r1	p7
r2	P8+
r3	P3+
r4	P6

Arch Map	
Reg	T
r1	p1
r2	p2
r3	p3
r4	p4

Reservation Stations					
#	busy	op	T	T1	T2
1					
2	Y	I2	p6	p5+	
3	Y	I3	p7	p5+	p2+
4					

CDB
T
p5,p8

Free List

R10K Cycle # 6_Sol

ROB							
ht	#	Insn	T	Told	S	X	C
	1						
h	2	I2: R4 = R1 * 10	p4	p5	c5	c6+	
	3	I3: R1 = R3 * R2	p7	p8	c6		
	4	I4: R2 = R2 + 5	p3	p2	c3	c4	c5
t	5	I5: R1 = R3 + R2	p1	p7			

Map Table	
Reg	T+
r1	p1
r2	p8+
r3	p3+
r4	p6

Arch Map	
Reg	T
r1	p5
r2	p2
r3	p3
r4	p4

Reservation Stations					
#	busy	op	T	T1	T2
1	Y	I5	p1	p3+	p8+
2					
3	Y	I3	p7	p5+	p2+
4					

CDB
T

Free List
~~p1~~