EECS 470 *Midterm Exam*

Fall 2021

Name: _____ unique name: _____

Sign the honor code:

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

_____

NOTES:

- Open book and Open notes
- Calculators are allowed, but no PDAs, Portables, Cell phones, etc.
- Don't spend too much time on any one problem.
- You have about 120 minutes for the exam.
- There are ***10*** pages including this one.
- Common assumption:
  - Memory has 32-bits of address and is byte addressable.
- **Be sure to show work and explain what you've done when asked to do so.**
- **The last page has two "answer areas". Clearly mark which one you want graded or we will grade the first one.**

1. Fill-in-the-blank or circle the *best* answer **[12 points, -2 per wrong/blank, minimum 0]**

    a. Given an 8MB, two-way associative cache with 32-byte lines, you will need _____ bits to index the cache.

    b. Say a wire that was 1mm in length, 20nm wide and 15nm high was replaced by a wire that as 4mm long, 40nm wide and had the same height. You would expect the resistance to ***go up by 8x / go up by 2x / stay constant / go down by 2x / go down by 8x.***

    c. The *wakeup* process can be best described as addressing ***true dependencies / false dependencies / control hazards / structural hazards***.
    The *select* process can best be described as addressing ***true dependencies / false dependencies / control hazards / structural hazards***.

    d. As you decrease the size of the RoB, you expect the IPC of the processor on a given workload to go ***up / down***, while you expect the clock period to go ***up / down*** or stay the same.

    e. The clock period of a 20 MHz clock is _____ microseconds.

    f. In the computer that first used Tamasulo's algorithm, the primary source of high-latency instructions was ***load / store / floating point / branch*** instructions. In modern computers it is instead ***load / store / floating point / branch*** instructions that are the primary long-latency instructions.

2. You're benchmarking a program that has a parallel component and a serial component. When using 2 cores, the parallel component takes up 80% of execution time and the serial component takes up the other 20%. The parallel and serial components never run at the same time. The parallel component's performance scales linearly with more cores.
    a. What would you expect the speedup to be if you forced the program to run with a single core? Show your work. **[4]**

    b. What is the theoretical maximum possible speedup (**compared to 1 core**) you can achieve by simply adding more cores to your system? Show your work, clearly highlighting any number used from part a (so we can better manage partial credit). **[4]**

3. Consider the R10K scheme and its structures as taught in class. Indicate what values are read or written by an instruction at a given point during its time in the out-of-order core. You are not responsible for the crossed out cells (though things may be happening there). Clearly indicate if nothing happens in a given situation. We've done one entry for you as an example. *Assume the RS is freed at issue.* **[12 points]**

	RAT	**RS**	**ROB**
Dispatch	Read: The 2 source registers. Write: The location indexed by the destination AR is written with the PRF number allocated for this instruction.		
Execution complete			
Commit			

4. Short answer **[12 points]**
    a. Consider the pipeline you were to implement for your third assignment, but assume that the structural hazard has been removed. A given program consists of 15% loads, 10% stores, 20% branches and the rest are ALU operations. If 40% of the branches are taken and 20% of all loads are followed by an instruction dependent on them, what is the expected CPI of the processor on this program? Show your work **[4]**

    b. One idea presented in McFarling's "Combining Branch Predictors is a "bimodal/gshare" predictor. Briefly define what this predictor is. You may assume "gshare" and "bimodal" are terms you don't need to redefine. **[4]**

    c. Consider a 32-entry, direct-mapped BTB which stores 10 bits of tag. Which bits will be used as the index? Which bits will be used as the tag? Assume you have 32-bit addresses with the most significant bit being 31 and the least significant bit being 0. **[4]**

5. The ISA used by your employer, QuickCPU Inc., lacks predicated instructions. You've been asked to consider adding predication to the ADD and ADDI instruction. The QuickCPU ISA has 32-bit instructions, uses 32 general purpose registers (GPRs) and uses 12-bit signed immediate values. The ADD instruction currently uses 3 register arguments (two source, one destination) while the ADDI instruction uses 2 register arguments and one immediate argument. **[10 points]**
    a. What *fraction* of the total ISA encodings possible are used by these two instructions? Briefly show your work.**[4]**

    b. If we had all of the ADD and ADDI instructions predicated by a single register (say r25) how would the above answers change? That is, there would be no non-predicated version of the instructions. **[3]**

    c. Same as part b, but we now can use any of the GPRs as a predicate. **[3]**

6. Consider the following RISC-V code. The hex numbers on the left are the address of the line in question.

```
                li r2, 1              // r2 = 1
                li r3, 0              // r3 = 0
                li r4, 9              // r4 = 9
0x100 loopi:    addi r2, 1            // r2 = r2 + 1
                ...                   // set r5 =(r2 mod 4)
0x2c4           bne r5, r0, addr3     // Branch 1
                addi r4, 1
     addr3:     addi r5, 1
                ...                   // set r6 to 1 if (r2 < 12); else to 0
0x4b8           bne r6, r0, loopi     // Branch 2
                wfi
```

What are the exact mispredicton rates for the following predictors? Predictors are initialized to not-taken or strongly not-taken as appropriate. *You must show your work to get credit!* **[12 points]**

a. A bimodal predictor with 4 entries each with a 2-bit saturating counter. **[5]**

Misprediction rate	
Branch 1:	Branch 2:

b. A local history predictor with a 4-entry BHT followed by an 8-entry PHT where each entry in the PHT is 1 bit. **[7]**

Misprediction rate	
Branch 1:	Branch 2:

7. You have been tasked with implementing a SystemVerilog module named **`expiry_reg`** that was started, but not completed, by a coworker. It is part of a larger design of a countdown FIFO called `expiry_fifo`. This is a FIFO (First In First Out) data structure where data will be invalidated after existing in the queue for a fixed number of clock cycles. The Countdown FIFO has a maximum size of `pCAPACITY`, but if elements are pushed when it is full, the oldest element will be lost forever.

   The `expiry_fifo` is built by connecting multiple instances of `expiry_reg`. `expiry_reg` is a shift register that will invalidate its data after `pLIFETIME` cycles. When the `shift` signal is high, the `expiry_reg` will shift in new information from its input at the next clock cycle.

   On the next page is the SystemVerilog code for `expiry_reg`, however, the `always_comb` block is incomplete. We have also provided the implementation of `expiry_fifo` as a handout in case you find referring to it to be helpful.

   You can assume the following typedefs throughout this problem, but the explicit size/width of DATA and COUNT should not affect your answer:

   sys_defs.svh:

```
typedef logic[31:0] DATA;
typedef logic[31:0] COUNT;
```

   [15 points]

Continued on the next page.

Problem 7 continued.

Fill in the `always_comb` for the `expiry_reg`. You may use typedefs and parameter values wherever necessary. You may not add internal/external signals to the design. Do not use assign statements for this problem.

```
module expiry_reg #(parameter pLIFETIME=100)
(
    input logic clock, reset,
    input DATA data_in,
    input COUNT counter_in,
    input logic valid_in, shift,
    output DATA data_out,
    output COUNT counter_out,
    output logic valid_out
);

    COUNT counter_next;
    DATA data_next;
    logic valid_next;

    always_comb begin
      /* Your code goes here */









    end

    always_ff @(posedge clock) begin
        if (reset) begin
            data_out        <= 64'b0;
            valid_out       <= 0;
            counter_out     <= 0;
        end else begin
            data_out        <= data_next;
            valid_out       <= valid_next;
            counter_out     <= counter_next;
        end
    end
endmodule
```

8. Consider the following state of a machine implementing what we've called the R10K algorithm with a retirement RAT. **[19 points]**

RAT

Arch Reg #	Phy. Reg #
0	0
1	12
2	4
3	8
4	2

ROB

Buffer Number	PC	Executed?	Dest. PRN	Dest ARN	
0	20	N	0	0	← HEAD
1	24	N	6	1	
2	28	Y	7	1	
3	32	Y	--	--	
4	36	N	8	3	
5	40	Y	12	1	
6					← TAIL
7					
8					

RRAT

Arch Reg #	Phy. Reg #
0	1
1	3
2	4
3	5
4	2

RS

RS#	Op Type	Op1 Ready?	Op1 PRN/value	Op2 Ready?	Op2 PRN/value	Dest PRN	ROB
0	*	Y	4	Y	-5	0	0
1	+	N	7	N	0	8	4
2	+	Y	4	N	0	6	1
3							
4							

PRF

Phy Reg #	Value	Free	Valid
0	1	N	N
1	2	N	Y
2	3	N	Y
3	4	N	Y
4	-5	N	Y
5	6	N	Y
6	7	N	N
7	-15	N	Y
8	9	N	N
9	0	Y	N
10	11	Y	N
11	12	Y	N
12	6	N	Y

KEY:

- **Op1 PRN/value** is the value of the first argument if "Op1 ready?" is yes; otherwise it is the Physical Register Number that is being waited upon.
- **Op2 PRN/value** is the same as above but for the second argument.
- **Dest. PRN** is the destination Physical Register Number.
- **Dest. ARN** is the destination Architectural Register Number.
- **ROB** is the associated ROB entry for this instruction.
- **Free/Valid** indicates if the PRF entry is currently available for allocation and if the valid in it is valid. *A free entry should be marked as invalid*.

Say that the instruction in ROB #3 is a branch and it was mispredicted: the next PC should have been 64. Say that the instruction in memory location 64 is R2=R1+R3 and in 68 is R0=R0+R2. Update the machine to the state where the branch has left the RoB, and the instructions at memory 64 and 68 have dispatched but not started execution. *When selecting a PRF use the* **highest** *numbered physical register available*, otherwise when making an arbitrary decision, just be sure it is legal. ***Be sure to update the head and tail pointers!*** **[18]**

On the following page is an extra copy of this state. You may use this one or the one on the next page but be sure to cross out (with a BIG X) the one you don't want graded.

```
module expiry_fifo #(parameter pCAPACITY=32, parameter pLIFETIME=100)
(
    input logic clock,
    input logic reset,
    input DATA data_in,
    input logic push, // "data_in is valid"
    input logic pop,
    output logic is_empty,
    output logic is_full,
    output DATA data_out,
    output logic valid_out,
);

    DATA [pCAPACITY-1:0] internal_data;
    COUNT [pCAPACITY-1:0] internal_count;
    logic [pCAPACITY-1:0] internal_valid;
    logic [pCAPACITY-1:0] internal_reset;
    logic [pCAPACITY-1:0] pop_sel_out;
    logic valid_in;
    COUNT start_count;

    expiry_reg #(.pLIFETIME(pLIFETIME)) nodes [pCAPACITY-1: 0]
    (
      // inputs
        .clock(clock),
        .reset(internal_reset),
        .data_in({data_in, internal_data[pCAPACITY-1:1]}),
        .valid_in({valid_in, internal_valid[pCAPACITY-1:1]}),
        .counter_in({start_count, internal_count[pCAPACITY-1:1]}),
        .shift(push),
     // outputs
        .data_out(internal_data),
        .valid_out(internal_valid),
        .counter_out(internal_count),
    );

    pri_sel #(.SIZE(pCAPACITY)) pop_selector //Works much like project 1.
    (
        .req(internal_valid),
        .gnt(pop_sel_out)
    );

    assign valid_in = push;
    assign is_empty = ~|internal_valid;
    assign is_full = &internal_valid;
    assign internal_reset = ({pCAPACITY{pop}} & {pop_sel_out}) | {pCAPACITY{reset}};
    assign start_count = pLIFETIME + 1;

    always_comb begin
        valid_out   = 0;
        data_out    = ~0;
        for (int i = pCAPACITY-1; i >= 0; i--) begin
            if (internal_valid[i]) begin
                valid_out = 1;
                data_out = internal_data[i];
            end
        end
    end
endmodule
```

Code for expiry_fifo to be used as reference if needed or helpful.