CS 250P
Fall 2019
Midterm
11/13/2019
Time Limit: 3:30pm - 4:50pm

Name (Print):

- **Don't forget to write your name on this exam.**
- **This is an open book, open notes, open electronics exam. But no online or in-class communication, and no Internet search.**
- **Ask us if something is confusing in the questions.**
- **Organize your work**, in a reasonably neat and coherent way, in the space provided. Work scattered all over the page without a clear ordering will receive very little credit.
- **Mysterious or unsupported answers will not receive full credit**. A correct answer, unsupported by explanation will receive no credit; an incorrect answer supported by substantially correct explanations might still receive partial credit.
- If you need more space, use the back of the pages; clearly indicate when you have done this.

Problem	Points	Score
1	5	
2	5	
3	5	
4	15	
5	15	
6	10	
7	10	
Total:	65	

1. Basics of CPU instruction set
   (a) (5 points) Write a simple assembly program that computes factorial of N, i.e., the product of all positive integers less than or equal to N. You can use any instruciton set you know (e.g., x86 or RISC), you don't have to be precise (the sketch of the code will work)

2. Basics of digital design
   (a) (5 points) Design a logical circuit that adds two two-bit numbers. Make sure that an additional output signals the overflow.

3. Pipelines and stalls

   (a) (5 points) Consider a 32-bit in-order pipeline with full bypassing that has the following stages:

```
Fetch   Decode/Regread   IntALU    Writeback
                         IntALU    Datamem     Datamem     Writeback
                         FPALU1    FPALU2      Writeback
```

   After decode, Int-adds go through the stages labeled "IntALU" and "Writeback", loads/stores go through the stages labeled "IntALU", "Datamem", "Datamem", and "Writeback", while FP-adds go through the stages labeled "FPALU1", "FPALU2", and "Writeback".

   How many stall cycles are introduced between the following pairs of successive instructions (remember, the processor implements full bypassing)?

   (i) Int-add, providing the address for a load/store,

   (ii) Load, providing the data for an FP-add

   (iii) Load, providing the data for a store,

   (iv) FP-add, followed by dependent FP-add.

4. Branch delay slot

   Consider a 7-stage in-order processor, where the instruction is fetched in the first stage, and the branch outcome is known after 3 stages. Estimate the CPI (cycles per instruction) of the processor under the following scenarios (assume that all stalls in the processor are branch-related and branches account for 20% of all executed instructions, assume that all branches are taken 60% of the time and not-taken 40% of the time).

   (a) (5 points) On every branch, fetch is stalled until the branch outcome is known.

   (b) (5 points) Every branch is predicted not-taken and the mis-fetched instructions are squashed if the branch is taken.

   (c) (5 points) The processor has two delay slots and the two instructions following the branch are always fetched and executed, and You are able to move three instructions before the branch into the delay slot.

5. Loop unrolling

   Consider a basic in-order pipeline with bypassing (one instruction in each pipeline stage in any cycle). The pipeline has been extended to handle FP add and FP mult. Assume the following delays between dependent instructions:

   - Load feeding any instruction: 3 stall cycles
   - FP ALU feeding any instruction (except stores): 4 stall cycles
   - FP MULT feeding any instruction (except stores): 7 stall cycles
   - FP ALU feeding store: 3 stall cycles
   - Int add feeding any instruction: 0 stall cycles
   - A conditional branch has 1 delay slot (an instruction is fetched in the cycle after the branch without knowing the outcome of the branch and is executed to completion)

   Below is the source code and default assembly code for a loop.

   Source Code:

```
for (i=1000; i>0; i--) {
    x[i] = y[i] + z[i];
}
```

   Assembly Code:

```
Loop:
    L.D F1, 0(R2) // Get y[i]
    L.D F2, 0(R3) // Get z[i]
    ADD.D F3, F2, F1 // Add the two numbers
    S.D F3, 0(R4) // Store the result into x[i]
    DADDUI R2, R2, #-8 // Decrement R2
    DADDUI R3, R3, #-8 // Decrement R3
    DADDUI R4, R4, #-8 // Decrement R4
    BNE R2, R1, Loop // Check if we've reached the end of the loop
    NOP
```

   (a) (5 points) Show the schedule (what instruction issues in what cycle) for the default code.

(b) (5 points) How should the compiler order instructions to minimize stalls (without unrolling) (note that the execution of a NOP instruction is effectively a stall)? Show the schedule. How many cycles can you save per iteration, compared to the default schedule?

(c) (5 points) How many times must the loop be unrolled to eliminate stall cycles? Show the schedule for the unrolled code.

6. Branch prediction

   The following "for" loop executes inside an infinite while loop. The inner "for" loop executesfive iterations and has a branch inside.

```
while (1) {

    for (i = 0; i < 10; i++) {

        if ((i & 1) == 0) {
            ...
        }
    }
    ...
}
```

   (a) (5 points) Assume that your CPU is using a 2-bit saturating counter predictor. Provide the pattern of branch prediction decisions by the predictor (assume that the external while loop ran for 100 iterations already, and the counters are in the steady state). What is the prediction success rate? I.e., out of 20 branches executed in one iteration of the inner "for" loop, what fraction is predicted correctly?

(b) (5 points) Imagine you implement a local history predictor (you're free to choose history size). How does this change the accuracy of the branch prediction?

7. Out-of-order execution

(a) (10 points) Consider an out-of-order processor similar to the one described in class. The architecture has 32 logical registers and 42 physical registers. On power up, the following program starts executing (to simplify the problem, some of the initialization code is not shown and you can ignore that code).

```
line1: L.D LR1 0(LR2)
       ADD.D LR1, LR1, LR1
       ST.D LR1, 0(LR2)
       ADD.D LR2, LR2, 8
       BNE LR2, LR3, line1
```

The processor has a width of 4, i.e., every pipeline stage can move 4 instructions through in every cycle. Show the renamed code for the first 20 instructions of this program. How soon will the 20th instruction get committed?

Assumptions:

Assume that branch prediction is perfect for a simple program like this. With the help of a trace cache, even fetch is perfect. Assume that caches are perfect as well. Assume that the dependent of an ADD.D instruction can leave the issue queue in the cycle right after the ADD.D. Assume that the dependent of an L.D cannot leave in the next cycle, but the cycle after that. Assume a ROB, an issue queue, and an LSQ with 20 entries each. When the thread starts executing, its logical register LR1 is mapped to physical register PR1, LR2 is mapped to PR2, and so on. An instruction goes through 5 pipeline stages before it gets placed in the issue queue and an additional 5 pipeline stages (6 for a LD/ST) after it leaves the issue queue (in other words, an instruction will take a minimum of 11 cycles to go through the pipeline). When determining if a L.D can issue, you need not check to see if previous store addresses have been resolved (just to make the problem simpler).