University of California at Berkeley
College of Engineering
Department of Electrical Engineering and Computer Sciences

EECS152/252A J. Wawrzynek
Spring 2022 4/7/22

Midterm Exam 2

Name: _____

Student ID number: _____

You have until 11AM to take the exam.

This is a *closed-book, closed-notes* exam, except for one handwritten sheet. Also no calculators, phones, pads, or laptops allowed.

Each question is marked with its number of points (one point per expected minute of time). Start by answering the easier questions then move on to the more difficult ones. You can use the backs of the pages to work out your answers. Neatly copy your answer to the allocated places. **Neatness counts.** We will deduct points if we need to work hard to understand your answer.

Write the student ID numbers of the person to your left and to your right on this page. If you are sitting on an aisle, just indicate "aisle" for left or right.

Left student ID number: _____

Right student ID number: _____

Before you turn in your exam, write your student ID number on all pages.

Friday 31st March, 2023 20:03

1. Comparing Architectures [8 pts]

(a) Below we describe certain types of programs and process/compiler technologies. For each, circle which type of architecture would be *most appropriate*.

i. Programs with many branches, where the branch behavior is not known until runtime.

Out-of-order superscalar VLIW Vector processor

ii. Linear algebra algorithms (such as matrix multiplication).

Out-of-order superscalar VLIW Vector processor

(b) Suppose that you have code that exhibits a high level of thread-level parallelism (TLP). You realize that you could run this code on either a superscalar OoO core with SMT, or on a GPU.

i. Suppose that your code includes deeply nested if-statements. Would you choose the GPU or the superscalar OoO core with SMT? Explain why, in no more than three sentences.

ii. Suppose that your code includes a very large number of scattered memory accesses. Would you choose the GPU or the superscalar OoO core with SMT? Explain why, in no more than three sentences.

iii. Suppose that your code exhibits a very high degree of data-level parallelism (in addition to TLP). Would you choose the GPU or the superscalar OoO core with SMT? Explain why, in no more than three sentences.

2. Out-of-Order Execution and Branch Prediction [21 pts]

Consider the following code, which performs an element-wise "max" operation on two vectors.

```
// In C
for (int i = 0; i < N; i++)
  if (a[i] >= b[i])
    c[i] = a[i];
  else
    c[i] = b[i];

// In assembly
//   x1 is the pointer to \a"
//   x2 is the pointer to \b"
//   x3 is the pointer to \c"
//   x4 is the length of the vectors (N)

loop:  ld x5, 0(x1) // a[i]
       ld x6, 0(x2) // b[i]

       blt x5, x6, label1

       sd x5, 0(x3) // c[i] = a[i]
       j label2

label1: sd x6, 0(x3) // c[i] = b[i]

label2: add x1, x1, 4
        add x2, x2, 4
        add x3, x3, 4
        add x4, x4, -1

        bnez x4, loop
 end:
```

Suppose that you have an out-of-order, superscalar core with the following characteristics:

- *Up to two* instructions can be fetched, decoded, dispatched, and issued every cycle.
- *Up to two* instructions can be committed every cycle.
- The ROB has *four* entries.
- Adds and branches take 1 cycle to execute.
- Loads and stores take 3 cycles to execute.
- All functional units (including memory units) are fully pipelined.
- All instructions must spend 1 cycle in the write-back stage before their result can be used by a dependent instruction.
- Instructions can commit one cycle after writeback, and ROB entries can be reused one cycle after commit.

- The CPU can execute all instructions speculatively in case of a branch. Speculative stores go into a store buffer to avoid corrupting memory.

Below is the register rename table at the beginning of execution:

Architectural Register	**Physical Register**
x1	P1
x2	P2
x3	P3
x4	P4
x5	P5
x6	P6

The free list is a FIFO with the initial state below. The leftmost element is the head of the free list, and the rightmost element is the tail.

P8	P10	P12	P14	P16	P18	P20

(a) **Register Renaming** [9 pts]

For this section, assume that the CPU correctly predicts all branches (as well as branch targets).

In the table below, fill in the cycle number for when each instruction enters the ROB, issues, writes back, and commits. Also, fill in the new register names for each instruction, where applicable. Use only physical register names for the src entries.

Assume that the ROB is initially empty. Part of the table has been filled in for you.

Time				OP	Dest	Src1	Src2
Enter ROB	Issue	WB	Commit				
0	1	4	5	`ld x5, 0(x1)`	P8	P1	-
0	1	4	5	`ld x6, 0(x2)`	P10	P2	-
1	5	6	7	`blt x5, x6, label1`	-	P8	P10
1	5	8	9	`sd x5, 0(x3)`	-	P8	P3
6	7	8	9	`j label2`	-	-	-

Also, fill in the values in the free list when the last instruction in the table above commits:

P12	P14	P16	P18	P20	P5	P6

Q2a Grading Rubric
Total of 9 pts

- Timing and register name table is not completely correct [-1]
- Does not delay the time when "j label2" enters the ROB [-1]
- Uses old register names for dependent instructions [-1]
- Incorrect number of destination registers for one instruction [-1]
- Incorrect number of destination registers for two or more instructions [-2]

- Incorrect number of source registers for one instruction [-1]
- Incorrect number of source registers for two or more instructions [-2]
- Does not pop elements off the free list correctly [-1]
- Does not push elements into the free list correctly [-1]

(b) **Speculation** [5 pts]

i. Suppose that in the instruction stream in **??**, the "`blt`" instruction was mispredicted. After the following instructions were flushed, what would be the state of the free list?

P12	P14	P16	P18	P20	P5	P6

ii. Suppose that we removed the speculative store buffer. How could we still maintain precise exceptions in that case? How would this affect program performance? Explain your answer, but in no more than three sentences.

One way we could maintain precise exceptions would be to execute stores only when they are at the head of the ROB and are known not to cause an exception. This might hurt performance by stopping stores (and potentially dependent instructions) from being executed speculatively or out-of-order.

(We also accepted answers which suggested loading and recording the previous data at a memory address prior to a store).

Q2b Grading Rubric
Total of 5 pts

- Free list is not exactly the same as in 2a [-1]
- Does not describe a plausible method to maintain precise exceptions in this scenario (or does not use enough detail) [-2]
- Does not mention that preventing speculative/out-of-order execution might hurt performance. (Or someother reasonable discussion of the performance impact of their solution). [-2]
- Unreasonably expensive solution, such as switching to complete in-order execution [-1]

(c) **Branch Prediction** [7 pts]

i. Suppose you have only two entries available in your branch-target buffer (BTB). If you wanted to maximize performance, which of these following three branches would you NOT store in your BTB? Circle only one.

```
blt x5, x6, label1
```
CORRECT

```
j label2
```

```
bnez x4, loop
```

ii. Now, suppose that the two vectors are actually very small. Vector "a" is {0, 100, 1}. Vector "b" is {9, 0, 9}.
Suppose that our branch predictor is a PC-indexed branch history table (BHT). Each entry in the BHT is a *two-bit saturating counter*. None of the branches in the loop above conflict in the BHT. Assume that all BHT entries begin execution at 10 (weakly taken). What will be the branch prediction accuracy of the "`blt x5, x6, label1`" branch?

i	State	Predicted	Actual
0	10	Taken	Taken
1	11	Taken	Not taken
2	10	Taken	Taken

Prediction accuracy: 2/3

Q2c Grading Rubric

Total of 7 pts

- Incorrect answer for multiple choice question [-2]
- Incorrect prediction accuracy [-2]
- Incorrect prediction accuracy without reasonable state transitions shown [-3]

3. VLIW [15 pts]

Suppose that we have a VLIW architecture with the following *fully-pipelined* functional units:

- One ALU unit which also performs branches. *1 cycle delay*
- Two FPU units which perform floating-point operations. *3 cycle delay*
- One load/store unit. *2 cycle delay*

Now, consider the code below, which computes the sum and sum-of-squares of a vector. You can assume that *N is very large*.

```
// In C
for (int i = 0; i < N; i++) {
  sum += arr[i];
  sumOfSquares += arr[i]*arr[i];
}

// In assembly
//   f0 is \sum"
//   f1 is \sumOfSquares"
//   x1 points to \arr"
//   x2 points to the end of \arr"

loop: fld f2, 0(x1)
      fadd f0, f0, f2 // sum += arr[i];
      fmul f3, f2, f2
      fadd f1, f1, f3 // sumOfSquares += arr[i]*arr[i];
      addi x1, x1, #4
      bne x1, x2, loop
```

(a) **Scheduling** [3 pts]

Schedule the loop above in the table below. You can just write the opcode and destination registers in the table. You can ignore the fixed integer offsets/constants in the "fld" and "addi" instructions. In other words, when you write the fld and addi instructions in the table, just write "fld f2" and "addi x1". We have filled out one element of the table below to get you started. *Minimize* the number of cycles taken. You can re-order instructions, but do not perform software pipelining or loop unrolling.

label	ALU	FPU	FPU	MEM
loop	addi x1			fld f2
		fmul f3	fadd f0	
	bne loop		fadd f1	

Q3a Grading Rubric
Total of 3 pts. Full credit assigned if schedule is correct and optimal

- Includes invalid scheduling of instructions [-1]
- Schedule is sub-optimal (does not minimize the number of cycles taken) [-1]

(b) **Software Pipelining** [10 pts]
Schedule the operations using software pipelining alone (without loop unrolling). Include the prologue and epilogue. Minimize the number of cycles taken by your program.

label	ALU	FPU	FPU	MEM
	`addi x1`			`fld f2`
	`addi x1`	`fadd f0`	`fmul f3`	`fld f2`
`loop:`	`addi x1`	`fadd f0`	`fmul f3`	`fld f2`
	`bne loop`	`fadd f1`		
		`fadd f0`	`fmul f3`	
		`fadd f1`		
		`fadd f1`		

Q3b Grading Rubric
Total of 10 pts. Full credit assigned if schedule is correct and optimal

- Correctness:
  - Includes incorrect prologue [-1]
  - Includes incorrect epilogue [-1]
  - Includes incorrect loop body [-1.5]
  - Miscellaneous errors [-0.5]
- Schedule optimality:
  - Schedule does not minimize the number of cycles taken [-1]

(c) **Short Answer** [2 pts]
Suppose that you added an infinite number of FPUs and ALUs to this VLIW machine. Would you be able to achieve one cycle per iteration then, without any loop unrolling? (You would still be permitted to re-order and pipeline your code). If so, why? If not, why not? Do not answer with more than three sentences.

There are two possible answers, depending on the scope of optimization techniques allowed as part of software pipelining.

- While not discussed in class, modulo scheduling is a technique that can be applied to software pipelining loops, with which we can set up the prologue and epilogue to achieve one cycle per iteration of the loop without unrolling. The answer is **yes** if you schedule instructions with RAW dependencies from different iterations of the loop, as shown in the Figure below.

label	ALU 1 and 2	FPU1	FPU 2 and 3	MEM
	addi x1			fld f2
	addi x1			fld f2
	addi x1		fmul f3; fadd f0	fld f2
	addi x1		fmul f3; fadd f0	fld f2
	addi x1		fmul f3; fadd f0	fld f2
loop	addi x1; bne	fadd f1	fmul f3; fadd f0	fld f2
		fadd f1	fmul f3; fadd f0	
		fadd f1	fmul f3; fadd f0	
		fadd f1		
		fadd f1		
		fadd f1		

- Since modulo scheduling was not covered during lectures and discussions, it is fair to limit the scope of SW pipelining to not allow prologue and epilogue set up as shown in the Figure. In this case, the answer is **no**, because otherwise the only way to resolve RAW data dependency between the `fmul f3` and `fadd f1` in each iteration is to use loop unrolling to achieve one cycle per iteration.

Q3c Grading Rubric
Total of 2 pts. 1 pt for correct answer, and 1 pt for valid reasoning.

4. MULTITHREADING [11 pts]

(a) **Performance** [4 pts]

Consider the code below, which adds a constant value to every element of a vector:

```
// In C
for (int i = 0; i < N; i++)
  arr[i] += x;

// In assembly
//   x1 points to \arr"
//   x2 points to the end of \arr"
//   f0 contains \x"

loop: fld f1, 0(x1)
      addi x1, x1, #4
      fadd f1, f0, f1 // arr[i] += x;
      fsd f1, -4(x1)
      bne x1, x2, loop
```

Suppose that:

- All memory operations take 10 cycles.
- Floating point additions take 3 cycles.
- Integer addititions and branches take 1 cycle. There is perfect branch prediction.

i. How many threads are required to avoid all stalls with fixed round-robin scheduling? Don't reorder the instructions.

The fld-to-fadd dependency causes the longest stall.
Suppose that the fld happens at time T. The fadd happens at $T + 2N$, where N is the number of threads.
$2N + T - T \geq 10$
$N \geq 5$

To illustrate, consider the following sequence of threads:
fld fld fld fld fld addi addi addi addi addi fadd

ii. Suppose that we instead use data-dependent thread scheduling, which switches threads whenever a stall would occur due to a RAW hazard. (You can assume that WAW and WAR hazards do not cause stalls).
How many threads would be required to avoid all stalls? (Consider only the steady-state). Don't reorder the instructions.

Once again, we want to remove the stall caused by the fld-to-fadd dependency.

In the steady-state, we could run a sequence of 4 consecutive instructions (fsd → bne → fld → addi) without any dependencies between them.
$4N \geq 10 - 2$
$N \geq 2$
We also need to add one for the main thread, so that we require 3 threads in total.

To illustrate, consider the following sequence of threads:
fsd bne fld faddi fsd bne fld faddi fsd bne fld faddi fadd fadd fadd

Q4a Grading Rubric
Total of 4 pts

- Incorrect result for round-robin [-2]
- Incorrect result for data-dependent scheduling [-2]
- Incorrect result for data-dependent scheduling, but identifies 4-instruction sequence which can proceed without stalls. [-1]

(b) **SMT** [7 pts]

Suppose that you added simultaneous multithreading (SMT) to a superscalar out-of-order core with a unified physical register file.

i. Would adding SMT to this core be expected to increase, decrease, or have no effect on the following values?

- The maximum number of instructions which can be committed every cycle.
  No effect
- The average number of instructions which would be committed every cycle.
  Increase
- The average number of instructions which are flushed upon an exception/misprediction. (Assume that the total ROB size remains constant)
  Decrease. There will be fewer instructions per thread in the ROB which need to be flushed upon an exception/misprediction in a single thread.
- The number of rename tables.
  Increase
- The number of functional units.
  No effect / Increase. You would want to increase the number of overutilized functional units, and keep the same number of underutilized functional units.

ii. Describe one way in which SMT can reduce the performance of a single particular thread. Your answer should not be more than three sentences.

There are many potential answers here. E.g. cache pollution, competition for instruction fetches, competition over functional units, etc.

Q4b Grading Rubric
Total of 7 pts

- Incorrect answer for (i) [-1 each]
- Incorrect/implausible answer for (ii), or not enough detail [-2]

5. Vector Processors [5 pts]

(a) Suppose you wanted to calculate the sum of all the elements in a vector. (This is a reduction which yields a single scalar result). How would you perform this operation with a vector ISA which supports vector additions, but no reductions? You can answer qualitatively, and you don't need to show any code.

The key idea is to replace the reduction with a series of vector additions. In a divide-and-conquer-like scheme, one could transform the reduction of a vector length N into a addition of two vectors of each length $N/2$. Then you perform an addition of two vectors of length $N/4$, and so on. Setting aside details on how to handle cases when N is strictly not a power of two, a reduction of a vector of length N can be performed with $O(log_2 N)$ vector additions (`vadd`).

Q5a Grading Rubric
Total of 2 pts. Full points are assigned to ideas referring to the notion of breaking down the reduction into a sequence of vector additions. Must include reference to vector additions, sums, and/or accumulation.

(b) Consider a vector processor with only a single vector lane. Would you expect this vector machine to perform better, worse, or the same as a scalar in-order CPU when running vectorizable code? Explain your answer in no more than three sentences.

Yes, the single vector lane processor still holds many advantages against scalar CPUs when executing vectorizable code. The correct solutions may refer to any of the following points:

- code size reduction from vector instructions reduces ICache pressure (reduced instruction fetch)
- vector processor can save on cycles and logic dedicated to fetch and decode compared to a scalar CPU
- vector processor has much simpler control and thus more likely to benefit from reduced critical path delay
- reduced number of branches; no hardware required for dynamic data hazard checks
- vector processor likely to have deeper pipelined functional units

Q5b Grading Rubric
Total of 3 pts. +1 for correct answer and +2 for clear and valid reasoning.