Computer System Architecture
6.823 Quiz #4
May 11th, 2016
Professors Daniel Sanchez and Joel Emer

Name: ______________________

This is a closed book, closed notes exam.
80 Minutes
15 Pages (+2 Scratch)

Notes:
- Not all questions are of equal difficulty, so look over the entire exam and budget your time carefully.
- Please carefully state any assumptions you make.
- Please write your name on every page in the quiz.
- You must not discuss a quiz's contents with other students who have not yet taken the quiz.
- Pages 16 and 17 are scratch pages. Use them if you need more space to answer one of the questions, or for rough work.

Part A	________	32 Points
Part B	________	32 Points
Part C	________	21 Points
Part D	________	15 Points
TOTAL	________	**100 Points**

Part A: VLIW Processors (32 points)

In this question, we will examine the execution of the code below on a single-issue in-order processor and a VLIW processor.

```
      ;; for (i = 0; i < N; i++)
      ;;    Y[i] = Y[i] + A*X[i];

      ;; Initial values:
      ;;  f1 := A
      ;;  r1 := &X[0] and r2 = &Y[0]
      ;;  r3 := &X[N] (first address after vector X)
I1:  loop: ld f0, 0(r1)
I2:        fmul f2, f0, f1
I3:        ld f3, 0(r2)
I4:        fadd f4, f2, f3
I5:        st f4, 0(r2)
I6:        addi r1, r1, 4
I7:        addi r2, r2, 4
I8:        bne r1, r3, loop
```

Question 1 (5 points)

The code above runs on an in-order, single-issue processor with perfect branch prediction and full bypassing. ALU (integer) operations have a 1-cycle latency (so, thanks to bypassing, consecutive dependent ALU operations execute without stalling), loads have a 2-cycle latency, and floating-point operations have a 3-cycle latency. How many cycles will the processor stall per loop iteration?

Question 2 (5 points)

Assume the in-order processor has appropriate support for software pipelining (e.g., a rotating register file). If you applied software pipelining to the original loop, what is the minimum number of iterations that you would need to overlap to remove all stalls in steady-state operation?

Question 3 (8 points)

Write VLIW code for the original instruction sequence, assuming the 3-operation VLIW format shown below. The VLIW architecture has the same fixed delays as the in-order processor (1/2/3 cycles for ALU/memory/floating-point operations, respectively), and has no stall logic. You may reorder and modify the code. For full credit, your implementation should use the minimum number of VLIW instructions.

Inst.	ALU/Branch Unit	Memory Unit	Floating Point Unit
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			

Question 4 (10 points)

Apply loop unrolling to the VLIW code to eliminate all stalls. Unroll the fewest number of iterations required to cover any latencies. Whatever degree of unrolling you choose, assume it divides the number of loop iterations exactly.

Inst.	ALU/Branch Unit	Memory Unit	Floating Point Unit
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			

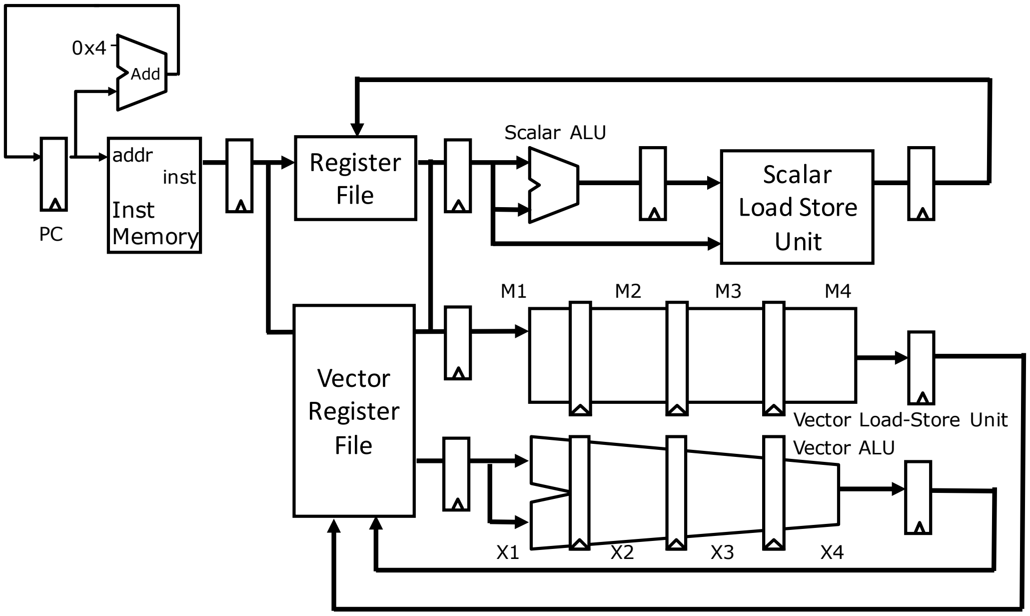
Question 5 (4 points)

Assume the VLIW processor has appropriate support for software pipelining (e.g., a rotating register file). What is the maximum throughput, in cycles per iteration, that the VLIW processor could achieve by applying software pipelining to the original loop?

Part B: Vector Processors (32 points)

We will analyze the performance of a vector processor with the following features:

- Single-issue, in-order execution
- Scalar instructions execute on a 5-stage, fully-bypassed pipeline
- 32 vector registers, **32 elements per vector register**
- **Four** vector lanes, with one ALU and one load-store unit per lane. Both take four cycles, are fully pipelined, and can process vector elements from independent instructions at the same time (the vector register file has enough ports per lane to feed both functional units)
- No support for vector chaining



The processor can issue a single (scalar or vector) instruction per cycle. Once it issues, a vector instruction uses the lanes' ALUs or load-store units for as many consecutive cycles as needed to produce all of its results. A vector instruction stalls if either its functional unit is unavailable, or if it depends on the result of a prior instruction, in which case it stalls until the prior instruction finishes writing back **all** of its elements. The vector register file has enough ports to keep the vector ALUs and load-store units fully utilized. The processor implements the MIPS ISA plus the following vector instructions:

Instruction	**Meaning**
`setvlr Rs`	Set vector length register (VLR) to the value in Rs
`lv Vt, Rs`	Load vector register Vt starting at address in Rs
`sv Vt, Rs`	Store vector register Vt starting at address in Rs
`add.vv Vd, Vs, Vt`	Add elements in Vs, Vt, and store result in Vd
`mul.vv Vd, Vs, Vt`	Multiply elements in Vs, Vt, and store result in Vd
`add.vs Vd, Vs, Rt`	Add Rt to each element in Vs, and store result in Vd
`mul.vs Vd, Vs, Rt`	Multiply each element in Vs by Rt, and store result in Vd

Question 1 (8 points)

Vectorize the scalar code below, assuming the X and Y arrays do not overlap. Assume that N is a multiple of the maximum vector length. Assume integers are 4-bytes.

```
      ;; for (i = 0; i < N; i++)
      ;;    Y[i] = Y[i] + A*X[i];

      ;; Initial values:
      ;;  r10 := A
      ;;  r1 := &X[0] and r2 = &Y[0]
      ;;  r3 := &X[N] (first address after vector X)
I1:  loop: ld r11, 0(r1)
I2:        mul r12, r10, r11
I3:        ld r13, 0(r2)
I4:        add r14, r12, r13
I5:        st r14, 0(r2)
I6:        addi r1, r1, 4
I7:        addi r2, r2, 4
I8:        bne r1, r3, loop
```

Provide equivalent vector code. For full credit, your code should execute as quickly as possible.

```
           addi r20, r0, 32      ;; set r20 to 32
           setvlr r20            ;; use all 32 vector elements
      loop:
```

Question 2 (6 points)

What is the throughput in cycles per iteration in steady state?

Question 3 (6 points)

Suppose we add chaining support to the processor. With chaining, a vector instruction that depends on a previous instruction can start execution if the first set of elements it processes is either already written to the vector register file or is available in the writeback stage (we add the requisite bypass paths). What is the throughput in cycles per iteration in steady state?

Question 4 (12 points)

State whether each of the loops below can be vectorized on our vector processor. If the code would require the vector processor to have additional features to be vectorizable, specify these features. If the code cannot be vectorized, state your reasoning.

All loops below operate on integer arrays `A[N]`, `B[N]`, and `C[N]`. These arrays do not overlap.

```
a) for (int i = 0; i < N; i++)
       A[i] = A[i+1] + B[i];
```

```
b) for (int i = 0; i < N; i++)
       A[i] = A[i-1] / B[i];
```

c)
```
for (int i = 0; i < N; i++)
  if (A[i] > 0)
    C[i] = A[i] + B[i];
```

d)
```
for (int i = 0; i < N; i++)
  C[i] = A[i] + A[B[i]];
```

Part C: Trace Scheduling and Predication (21 points)

In this question, we will study the tradeoffs between trace scheduling and predication. We will use the MIPS code sequence shown below.

```
     ;; if (x != 0) c += b;
     ;; else c += 10;
  I0: ld r1, 0(r11)        ;; r1 := x
  I1: ld r2, 0(r12)        ;; r2 := c
  I2: bne r1, I5           ;; x != 0 ?
  I3: addi r2, r2, 10      ;; c += 10
  I4: j end
  I5: ld r3, 0(r13)        ;; r3 := b
  I6: add r2, r2, r3       ;; c += b
 end: …
```

Question 1 (5 points)

A trace scheduling compiler optimizes the above code sequence, leveraging that the branch I2 is rarely taken. The optimized trace is shown below. Write the compensation code for this trace.

```
 I1: ld r2, 0(r12)
 I0: ld r1, 0(r11)
 I3: addi r2, r2, 10
 I2: bne r1, compensation
end: …
```

Trace A

```
compensation:








    j end
```

Compensation A

Question 2 (6 points)

Instead of performing trace scheduling, the compiler decides to emit predicated code for the given code sequence. We extend the MIPS ISA with a single predicate register, `p`, new instructions to set `p`, and extend all opcodes to support predicated variants of all ALU and memory instructions, as shown below:

Instruction	**Meaning**
`setpeq rs`	Set predicate register `p` to 1 if `rs` == 0, or set `p` to 0 otherwise
`setpne rs`	Set predicate register `p` to 1 if `rs` != 0, or `p` to 0 otherwise
`(p) instruction`	If predicate register `p` is 1, then execute the instruction.
`(!p) instruction`	If predicate register `p` is 0, then execute the instruction.

Rewrite the original code sequence to leverage predication. For full credit, use the minimum number of instructions.

Question 3 (10 points)

Consider the standard 5-stage MIPS pipeline: fully bypassed, with a one-cycle load-use stall, where branches are predicted not taken, and branches are resolved in the ALU stage.

We extend this pipeline to support predication by maintaining the predicate register `p` in the ALU stage. `setpeq` and `setpne` update `p` eagerly in the ALU stage, and predicated instructions that should not be executed are turned to noops as they traverse the ALU stage.

Please answer the following questions:

a) (2 points) How many cycles does the trace-scheduled code in Question 1 take to execute in this processor, *if branch I2 is not taken*?

b) (3 points) How many cycles does your trace-scheduled code in Question 1 take to execute in this processor, *if branch I2 is taken*?

c) (3 points) How many cycles does your predicated code in Question 2 take to execute in this processor?

d) (2 points) How frequently does branch I2 need to be taken for the predicated code to be faster than the trace-scheduled code?

Part D: Transactional Memory (15 points)

You are designing a hardware transactional memory (HTM) system that uses pessimistic concurrency control (i.e., on each load/store, the HTM checks for conflicting accesses to the same address made by other transactions). Comment on whether the following conflict resolution policies suffer from either livelock (i.e., the system may reach a state where *no single transaction* makes forward progress) or starvation (i.e., the system may reach a state where *at least one transaction* does not make forward progress). State your reasoning.

1. **Requester wins**: Upon a conflict, the transaction whose request initiated the conflict check is granted access to the data, and any conflicting transactions are aborted. After aborting, transactions immediately restart execution.

2. **Timestamp-based, retain timestamp on abort**: Each transaction is assigned a unique timestamp when it first begins execution. Timestamps are monotonically increasing. Upon a conflict, if the requesting transaction's timestamp is lower than the timestamps of all other conflicting transactions, the requester is granted access to the data, and other conflicting transactions are aborted. Otherwise, the requesting transaction is aborted.

   After aborting, transactions immediately restart execution. Aborted transactions retain their original timestamp when they restart execution.

3. **Timestamp-based, discard timestamp on abort**: Like the previous policy, except that aborted transactions discard their previous timestamp and acquire a new one when they restart execution.

4. **Random-number-based, retain random number on abort**: Each transaction is assigned a unique random number when it first begins execution. Upon a conflict, if the requesting transaction's random number is lower than the random numbers of all other conflicting transactions, the requester is granted access to the data, and other conflicting transactions are aborted. Otherwise, the requesting transaction is aborted.

   After aborting, transactions immediately restart execution. Aborted transactions retain their original random number when they restart execution.

5. **Random-number-based, discard random number on abort**: Like the previous policy, except that aborted transactions discard their previous random number and acquire a new one when they restart execution.

Scratch Space

Use these extra pages if you run out of space or for your own personal notes. We will not grade this unless you tell us explicitly in the earlier pages.

Scratch Space