Computer System Architecture
6.823 Final Examination

Name:____________________

This is an open book, open notes exam.
180 Minutes
21 Pages

Notes:

- Not all questions are of equal difficulty, so look over the entire exam and budget your time carefully.
- Please write your name on every page in the final (you get 5 points for doing this).

	Name:	________	5 Points
Part A	(Q1-Q5):	_______	10 Points
Part B	(Q6-Q11):	_______	12 Points
Part C	(Q12-Q14):	_______	12 Points
Part D	(Q15-Q23):	_______	34 Points
Part E	(Q24-Q26):	_______	18 Points
Part F	(Q27-Q28):	_______	16 Points
Part G	(Q29-Q32):	_______	25 Points
Part H	(Q33):	_______	10 Points
Part I	(Q34-Q38):	_______	28 Points
Part J	(Q39-Q40):	_______	10 Points

Total: ________ 180 Points

Name________________________________

Part A: Caches (10 points)

Circle whether each of the following statements is true or false. Assume in **ALL** questions that the cache capacity is fixed.

Question 1 (2 points)

Increasing cache line size reduces capacity misses.

T **F**

Question 2 (2 points)

Doubling the cache line size can at most increase conflict miss rates by a factor of 2.

T **F**

Question 3 (2 points)

Doubling the cache associativity can at most reduce conflict miss rates by half.

T **F**

Question 4 (2 points)

Compulsory misses can be reduced without changes to software.

T **F**

Question 5 (2 points)

Changing a cache from using an MRU (most recently used) replacement policy to an LRU replacement policy never increases miss rates.

T **F**

Part B: Pipeline Hacking (12 points)

Ben decides to develop a 4-stage DLX machine (called the DLX4), a variant of the standard 5-stage pipeline machine (DLX5). He wants to merge the X (execute) and M (memory) stages of the DLX5 into one unified X/M stage. To reduce the latency of this X/M stage, he places the ALU and memory in parallel so that the DLX4 can only use register-indirect addressing. Assume that both machines are fully bypassed and have branch delay slots. Determine whether each of the following statements is true or false.

Question 6 (2 points)

DLX4 has better throughput than DLX5 since one instruction takes only four stages for DLX4.

T **F**

Question 7 (2 points)

DLX4 does not have a load-delay slot (the hazard between a load instruction and an immediately following instruction that uses the result of the load) since the X and M stages are merged.

T **F**

Question 8 (2 points)

In general, when compiling the same code, the compiler for DLX4 is more likely to run out of registers than the compiler for DLX5.

T **F**

Question 9 (2 points)

DLX4 has an additional data hazard which doesn't exist for DLX5.

T **F**

Question 10 (2 points)

DLX4 requires less bypassing hardware.

T **F**

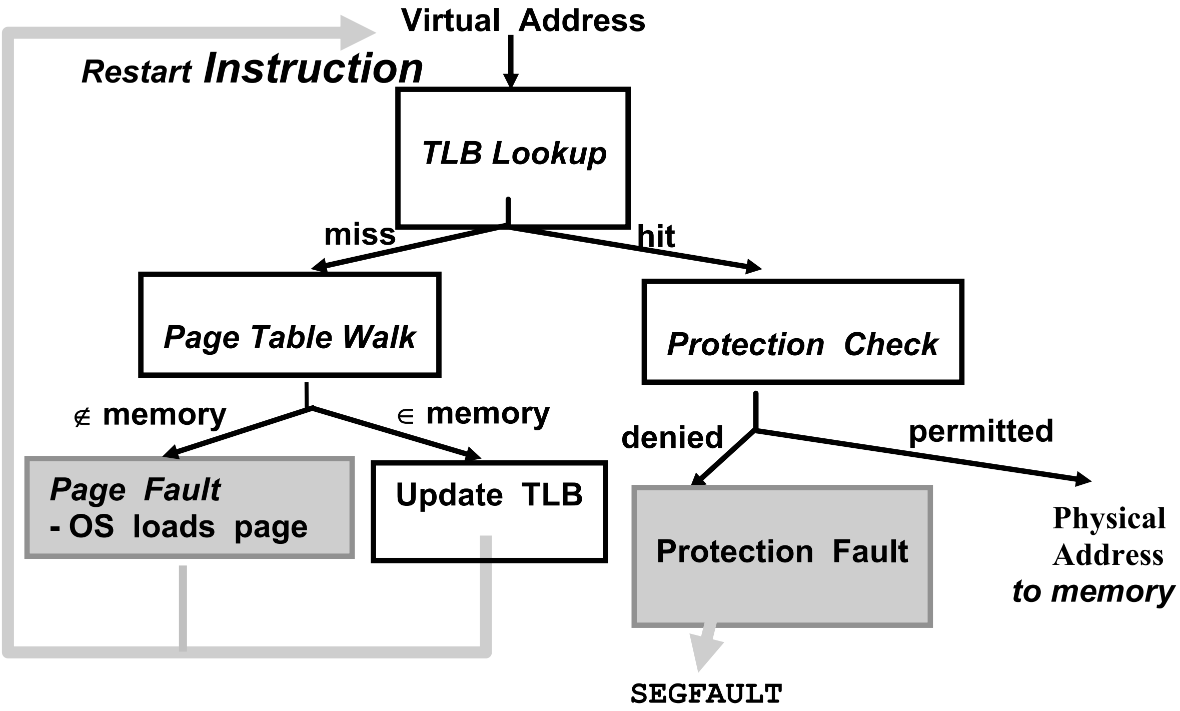
Question 11 (2 points)

DLX4 suffers a performance loss because it lacks the offset-addressing mode for loads and stores.

T **F**

Part C: Virtual Memory System (12 points)

Ben is designing a 32-bit byte-addressed virtual memory system. It has a TLB (Translation Lookaside Buffer) and 2-level hierarchical PTs (Page Tables), but doesn't have a cache. Each page table occupies one page in memory. The page table walk is done completely by hardware. After every page fault and TLB update, the instruction that caused the TLB miss is restarted. The virtual address consists of 20-bit VPN (Virtual Page Number) and 12-bit offset. The following figure shows the address translation flow diagram.



Question 12 (2 points)

Ben thinks the TLB reach should be at least 128 KB. What is the minimum number of TLB entries? (Circle the correct answer)

1) 64
2) 32
3) 16
4) 8
5) Can't determine from the given information.

Question 13 (4 points)

What is the best case time taken for a user load instruction in terms of TLB lookups, memory accesses, and page faults? Assume that there is no protection fault. (Circle the correct answer)

1) one TLB lookup
2) one TLB lookup + one memory access
3) one memory access
4) one TLB lookup + two memory accesses
5) one TLB lookup + one memory access + one page fault

Question 14 (6 points)

What is the worst case time taken for a user load instruction in terms of TLB lookups, memory accesses, and page faults? Assume that there is no protection fault and that the level-1 page table is in memory. (Circle the correct answer)

1) one TLB lookup + three memory accesses + one page fault
2) two TLB lookups + three memory accesses + two page faults
3) three TLB lookups + five memory accesses + two page faults
4) four TLB lookups + six memory accesses + two page faults
5) four TLB lookups + ten memory accesses + three page faults

Part D: Fetch Pipelines (34 points)

Ben is designing a deeply-pipelined single-issue in-order RISC processor. The first half of his pipeline is as follows:

PC	PC Generation
F1	ICache Access
F2	
D1	Instruction Decode
D2	
RN	Rename/Reorder
RF	Register File Read
EX	Integer Execute

All questions assume a **DLX/MIPS** type ISA. There are no branch delay slots and currently there is **no** branch prediction hardware (instructions are fetched sequentially unless the PC is redirected by a later pipeline stage). The subroutine call uses **`JAL/JALR`** (jump and link). These instructions write the return address (PC+4) into the link register (r31). The subroutine return uses **`JR r31`**, and the stack is used to pass arguments. Assume that PC Generation takes a whole cycle and that you cannot bypass anything into the end of the PC Generation phase.

Question 15 (2 points)

Immediately after what pipeline stage does the processor know that it is executing a subroutine return instruction?

Question 17 (2 points)

Immediately after what pipeline stage does the processor know the subroutine return address?

Question 18 (3 points)

How many pipeline bubbles are required when executing a subroutine return?

Question 19 (3 points)

Louis Reasoner suggests adding a BTB to speed up subroutine returns. Why doesn't a standard BTB work well for predicting subroutine returns?

Question 20 (6 points)

Instead of a BTB, Ben decides to add a return stack to his processor pipeline. This return stack records the return addresses of the *N* most recent subroutine calls. This return stack takes no time to access (it is always presenting a return address).
Explain how this return stack can speed up subroutine returns. Describe when and in which pipeline stages return addresses are pushed on and popped off the stack.

Question 21 (6 points)

Draw a pipeline diagram corresponding to the execution of the following code on the machine of question 20:

```
A: JAL B
.
.
B: JR r31
```

Make sure to indicate the address of the instruction that is being executed. The first two instructions are illustrated below. The crossed out stages indicate that the instruction was annulled during those cycles.

time→

A	PC	F1	F2	D1	D2	RN	RF	EX	
A+1		PC	F1	F2	D1	~~D2~~	~~RN~~	~~RF~~	~~EX~~

Question 22 (4 points)

If the return address prediction is wrong, how is this detected? How does the processor recover, and how many cycles are lost (relative to a correct prediction)? State precisely any assumptions you make.

Question 23 (8 points)

Describe a hardware structure that Ben could add, in addition to the return stack, to improve the performance of return instructions so that there is usually only a one-cycle pipeline bubble when executing subroutine returns (assume that the structure takes a full cycle to access).

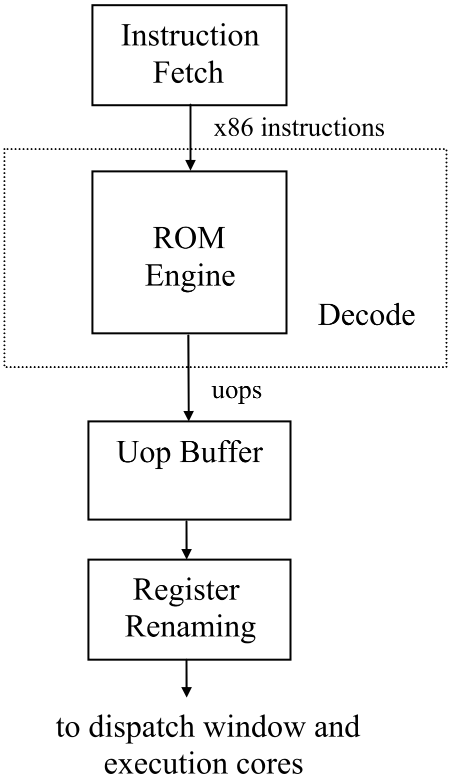
Part E: Exceptions and Register Renaming (18 points)

With the success of the Bentium processor from the midterm, Ben Bitdiddle decides to start Bentel Corporation, a company specializing in high-end x86 processors to compete with Intel. His latest project is the Bentium 4, a superscalar out-of-order processor with register renaming and speculative execution.

The Bentium 4 has 8 architectural registers (EAX, EBX, ECX, EDX, ESP, EBP, EDI, and ESI). In addition, the processor provides 8 internal registers T0-T7 not visible to the ISA that can be used to hold intermediary values used by microoperations (uops) generated by the microcode engine. The microcode engine is the decode unit and is used to generate uops for all the x86 instructions. For example, the following register-memory x86 instruction might be translated into the following RISC-like uops:

$$\begin{array}{lll} \text{ADD } R_d, R_a, \text{offset}(R_b) & \rightarrow & \text{LW} \quad \text{T0, offset}(R_b) \\ & & \text{ADD } \; R_d, R_a, \text{T0} \end{array}$$

All 16 uop-visible registers are renamed by the register allocation table (RAT) into a set of physical registers (P0-Pn) as described in Lecture 22. There is a separate shadow map structure that takes a snapshot of the RAT on a speculative branch in case of a misprediction. The block diagram for the front-end of the Bentium 4 is shown below:



Note: the decode block is actually replicated in the Bentium 4 in order to decode multiple instructions per cycle (not shown in diagram)

Question 24 (6 points)

For the Bentium 4, if an x86 instruction takes an exception before it is committed, the machine state is reset back to the precise state that existed right before the excepting instruction started executing. This instruction is then re-executed after the exception is handled. Ben proposes that the shadow map structure used for speculative branches can also be used to recover a precise state in the event of an exception. Specify an implementable strategy for taking the least number of snapshots of the RAT that would still allow the Bentium 4 to implement precise exception handling.

Question 25 (6 points)

Ben further states that the shadow map structure does not need to take a snapshot of all the registers in the Bentium 4 to be able to recover from an exception. Is Ben correct or not? If so, state which registers do not need to be recorded and explain why they are not necessary, or explain why all the registers are necessary in the snapshot.

Question 26 (6 points)

Assume the Bentium 4 has the same register renaming scheme as the Pentium 4, as described in Lecture 22. What is the minimum number of physical registers (P) that the Bentium 4 must have to allow register renaming to work? Explain your answer.

Part F: Branch Prediction (16 points)

Ben wants to test different dynamic branch predictors for the following high-level code.

```
for (i = 0; i < N; i++) {
     if (i % 4 ==  0)    // % is modulus operator
          k = k + 1;
     if (i % 2 == 0)
          k = k + 2;
     }
```

Ben translates this high-level code into the following assembly code. Assume that N is a very large integer and there is no branch delay slot.

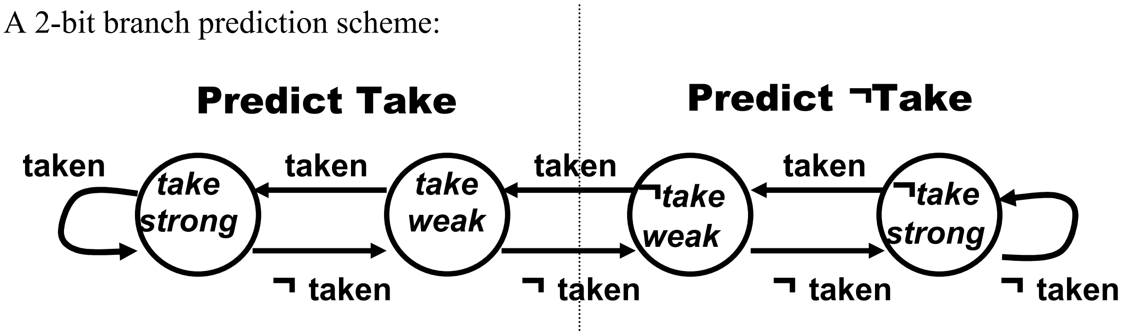
```
addi r1, r0, N // r1 holds N
add r2, r0, r0 // r2 holds i

loop:     andi r3, r2, 3      // r3 = i % 4
          bnez r3, temp1      // Branch 1
          addi r5, r5, 1      // k = k + 1

temp1:    andi r3, r2, 1      // r3 = i % 2
          bnez r3, temp2      // Branch 2
          addi r5, r5, 2      // k = k + 2

temp2:    addi r1, r1, -1     // decrement loop count
          addi r2, r2, 1      // increment i
          bnez r1, loop       // Branch 3
```

Ben has four different dynamic branch predictors as follows.

Dynamic Branch Predictor 1: 1-bit BHTs (Branch History Table)
Dynamic Branch Predictor 2: 2-bit BHTs
Dynamic Branch Predictor 3: 2-bit BHTs + 1-bit Global History bit
Dynamic Branch Predictor 4: 2-bit BHTs + 2-bit Global History bit

Ben invents a 2-bit branch prediction scheme (the figure shown below) and uses it for his dynamic branch predictors. Note that this 2-bit branch prediction scheme is different from the one in Lecture-12.

A 2-bit branch prediction scheme:



Question 27 (4 points)

Does the prediction accuracy of any of the branches depend on the initial state of the BHT for dynamic branch predictor 2? If so, state which branches.

Question 28 (12 points)

Fill out the table with the prediction accuracies of each dynamic branch predictor. If the prediction accuracy depends on the initial state, write down all the possible prediction accuracy numbers, but you don't need to specify the initial conditions.

	DBP 1	DBP 2	DBP 3	DBP 4
Branch 1				
Branch 2				
Branch 3				

Part G: Trace Scheduling (25 points)

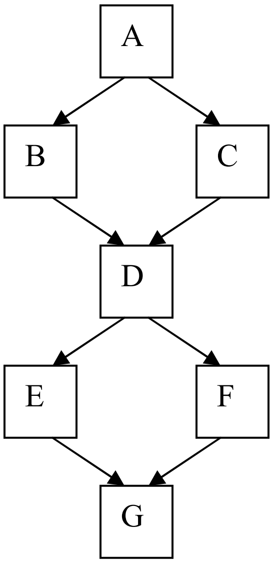
Trace scheduling is a compiler technique that increases ILP by removing control dependencies, allowing operations following branches to be moved up and speculatively executed in parallel with operations before the branch. It was originally developed for statically scheduled VLIW machines, but it is a general technique that can be used in different types of machines and in this question we apply it to an in-order superscalar processor.

Consider the following piece of C code (% is modulus) with basic blocks labeled:

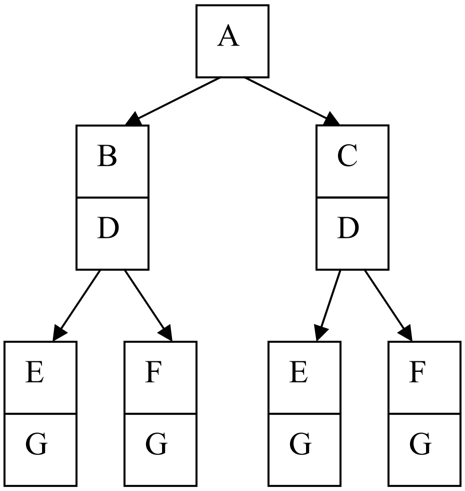
```
A    if (data % 8 == 0)
B      X = V0 / V1;
  else
C      X = V2 / V3;
D    if (data % 4 == 0)
E      Y = V0 * V1;
  else
F      Y = V2 * V3;
G
```

Assume that **data** is a uniformly distributed integer random variable that is set sometime before executing this code.

The program's control flow graph is



The decision tree is



Path probabilities for Q29: _____ _____ _____ _____

A control flow graph and the decision tree both show the possible flow of execution through basic blocks. However, the control flow graph captures the static structure of the program, while the decision tree captures the dynamic execution (history) of the program.

Question 29 (5 points)

On the decision tree, label each path with the probability of traversing that path. For example, the leftmost block will be labeled with the total probability of executing the path ABDEG. (Hint: you might want to write out the cases)

Question 30 (2 points)

On the control flow graph, circle the path that is most likely to be executed.

Question 31 (10 points)

This is the DLX code (no delay slots):

```
A:   ld   r1, data
     andi r2, r1, 7  ;; r2 <- r1%8
     bnez r2, C
B:   div  r3, r4, r5 ;; X <- V0/V1
     j    D
C:   div  r3, r6, r7 ;; X <- V2/V3
D:   andi r2, r1, 3  ;; r2 <- r1%4
     bnez r2, F
E:   mul  r8, r4, r5 ;; Y <- V0*V1
     j    G
F:   mul  r8, r6, r7 ;; Y <- V2*V3
G:
```

This code is to be executed on an in-order superscalar **without** branch speculation. Assume that the memory, divider, and multiplier are all separate long latency, **unpipelined** units that can be run in parallel. On the next page, rewrite the above code using trace scheduling. Optimize only for the most common path. Just get the other paths to work. Don't spend your time performing any other optimizations. Ignore the possibility of exceptions. (Hint: Write the most common path first then add fix-up code.)

Name_________________________________

Space for Question 31

Question 32 (8 points)

Assume that the load takes x cycles, divide takes y cycles, and multiply takes z cycles. Approximately how many cycles does the original code take? (ignore small constants)

Approximately how many cycles does the new code take in the best case?

Name________________________________

Part H: Vectorization of DAXPY (10 points)

The following high-level code calculates DAXPY (Double precision A times X[i] Plus Y[i]), a popular routine in linear algebra programs.

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

Assume that N is a very large integer and a multiple of 32 and arrays X, Y, and S do not overlap in memory.

You are given a vector machine, called the Vectium, which is based on the VDLX ISA (DLX ISA with vector instruction extension). The Vectium has 32 elements in each vector register and a large number of vector registers.

The Vectium has one vector addition unit, one vector multiplication unit, one vector store unit, and two vector load units. The vector addition unit and the vector multiplication unit both have 4 cycle latencies. The vector load and the vector store unit have 20 cycle latencies. All units have dedicated read and write ports for the vector register file and are fully-pipelined.

The Vectium has 8 lanes and all units have 4 cycle dead times. Assume the Vectium has full and flexible vector chaining.

Question 33 (10 points)

Calculate the maximum possible execution rate for DAXPY running on the Vectium. Give the rate as the number of loop iterations computed per cycle. Note that you don't need to write down the assembly code.

Part I: Cache Coherence (28 points)

After learning about cache coherence protocols from the second half of the course, Ben decided to put two Bentium I processors together to make a multiprocessor system. This system uses the snooping cache coherence protocol introduced in Problem 1 of Problem Set 6. Each processor has a 256KB on-chip cache with cache line size of 16 bytes.

Ben runs the following simple benchmark. Assume that integer arrays A, B, and C do not overlap in memory and that integers are four bytes.

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

Since there are two processors now, the `ben-cc` compiler parallelizes the code and executes

```
for (int i=0; i<N; i+=2) {
    A[i] = B[i] + C[i];
}
```

on Processor 0 and

```
for (int i=1; i<N; i+=2) {
    A[i] = B[i] + C[i];
}
```

on Processor 1.

Question 34 (8 points)

After observing the steady state performance of the system, Ben sees that sometimes the system has poor performance. Assume that address `a` starts at the beginning of a cache line, fill in the state of the cache line containing words `a` through `a+3.` The cache line state is the final state after the processor action finishes. Remember the five possible states are Invalid (I), Clean Exclusive (CE), Owned Exclusive (OE), Clean Shared (CS), and Owned Shared (OS).

Time

Action	Proc 0 Cache Line State	Proc 1 Cache Line State
	I	**I**
Proc 0 writes `a`		
Proc 1 writes `a+1`		
Proc 1 writes `a+3`		
Proc 0 writes `a+2`		

Question 35 (2 points)

How many invalidations are required by the above code sequence?

Question 36 (6 points)

Alyssa comes along and tells Ben that by changing the parallelization method a little bit in Ben-cc, this benchmark can be sped up significantly. Show what Alyssa means by rewriting the **simplest C code** for both processors 0 and 1. How many invalidations does your code require per loop body?

Question 37 (5 points)

Alyssa advises Ben that even though this benchmark can be easily fixed, there could still be potential problems for other benchmarks. She recommends that for Bentium II, Ben should add a dirty bit for each word in the cache line, while keeping the rest of the design identical to Bentium I. However, this modification does not seem to speed up the original benchmark. Explain why.

Question 38 (7 points)

Please help Ben by suggesting one alternative hardware modification that improves performance of the system on the original benchmark.

Part J: Sequential Consistency (10 points)

We have the following code sequence on processors P1 and P2.

Initially, `A = 0, B = 0, C = 0, D = 0.`

```
P1                        P2

A = 1;                    B = 1;
C = B;                    D = A;
```

Question 39 (4 points)

Circle all answers that are not a sequentially consistent result of the execution of the above code sequences.

a) `C = 0; D = 0;`
b) `C = 0; D = 1;`
c) `C = 1; D = 0;`
d) `C = 1; D = 1;`

Question 40 (6 points)

Assume now that the code is run on a system that does not guarantee sequentially consistency, but that memory dependencies are not violated for the accesses made by any individual processor. The system has a MEMBAR memory barrier instruction that guarantees the effects of all memory instructions executed before the MEMBAR will be made globally visible before any memory instruction after the MEMBAR is executed. Indicate where MEMBAR instructions should be added to the above code sequences to give the same results as if the system was sequentially consistent. Use the minimum number of MEMBARs.