

18-447 Intro to Computer Architecture, Spring 2012

Midterm Exam I

Instructor: Onur Mutlu

Teaching Assistants: Chris Fallin, Lavanya Subramanian, Abeer Agrawal

Date: March 7, 2012

Name:

SOLUTION

Problem I (50 Points) :		<input type="text"/>
Problem II (50 Points) :		<input type="text"/>
Problem III (60 Points) :		<input type="text"/>
Problem IV (50 Points) :		<input type="text"/>
Problem V (50 Points) :		<input type="text"/>
Problem VI (50 Points) :		<input type="text"/>
Problem VII (20 Points) :		<input type="text"/>
Bonus Problem VIII (40 Points) :		<input type="text"/>
Legibility and Name (5 Points) :		<input type="text"/>
Total (335 + 40 Points) :		<input type="text"/>

Instructions:

1. This is a closed book exam. You are allowed to have one letter-sized cheat sheet.
2. No electronic devices may be used.
3. This exam lasts 1 hour 50 minutes.
4. Clearly indicate your final answer.
5. Please show your work when needed.
6. Please write your initials on every odd page.
7. Please make sure that your answers to all questions (and all supporting work that is required) are contained in the space provided.

- **Be cognizant of time.** Do not spend too much time on one question.
- **Be concise.** You will be penalized for verbosity.
- **Show work when needed.** You will receive partial credit at our discretion.
- **Write legibly.** Show your final answer.

I. Potpourri (50 Points)

1) Pipelining (5 Points)

a) Circle one of A, B, C, D. As pipeline depth increases, the latency to process a single instruction:

A. decreases

B. increases

C. stays the same

D. could increase, decrease, or stay the same, depending on

b) Explain your reasoning (in no more than 20 words):

Each additional pipeline latch adds processing overhead (additional latency).

2) Data Dependencies (5 Points)

In a superscalar processor, the hardware detects the data dependencies between *concurrently fetched* instructions. In contrast, in a VLIW processor, the compiler does.

3) Program Counter (5 Points)

In the MIPS ISA, which instruction(s) do not change the program counter?

None

4) Endianness (5 Points)

Say that we were to take a processor architecture designed as big-endian and change it to be little-endian. Assuming the subset of the MIPS ISA which you have implemented in the labs, and assuming a memory with a word-level interface, which instructions' operation will be affected by this endianness change?

Halfword- and byte-sized load and store instructions (LH, LHU, SH, LB, LBU, SB). Word-size loads and stores do not change because the memory has a word-level interface.

5) ISA vs Microarchitecture (5 Points)

Indicate whether each of the following design choices in a processor is a feature of the ISA or of the microarchitecture:

(i) Microarchitecture Two-level global branch prediction

(ii) ISA Predicated instruction execution

- (iii) ISA A floating-point unit that uses wide floating-point values for additional accuracy
- (iv) Microarchitecture A 32-bit wide data bus to memory
- (v) ISA An additional set of user-visible registers

6) Interlocking (5 Points)

- (a) Remember that MIPS was initially designed such that hardware did not need to provide interlocking, and it was the compiler's job to reorder instructions accordingly to ensure the pipeline operates correctly. Now, consider a pipelined MIPS implementation that implements hardware based interlocking. Could compile-time instruction reordering provide any benefit in this implementation? **YES** **NO**
(Circle one)
- (b) Why? Why not? Explain in less than 20 words.

Software reordering can still avoid stall conditions (e.g., use immediately after a load).

7) INSQUE Instruction in VAX (5 Points)

The VAX has an instruction INSQUE ptr1, ptr2 which causes the node pointed to by ptr1 to be inserted into a doubly linked list immediately after the node pointed to by ptr2. We say the doubly linked list is a data type in the VAX ISA.

8) Branch Prediction (5 Points)

A snapshot of the taken/not-taken behavior of a branch is:

... T T T T T T T N N T **T** N N T N N T

If the branch predictor used is a 2-bit saturating counter, how many of the last ten branches are predicted correctly?

2 (correctly predicted branches are marked in bold.)

9) Register Dependencies (10 Points)

- (a) What is the fundamental cause of false register dependencies (output and anti, or write-after-read, write-after-write dependencies)?

Not enough architectural registers

- (b) What can be changed in the ISA, compiler, and microarchitecture to eliminate false dependencies, if at all possible, in each of the three levels above? Describe one disadvantage of each approach.

ISA:

Approach: **Add additional architectural registers.**

Disadvantage: **More bits in instruction (less dense code).**

Compiler:

Approach: **Modify register allocation to avoid re-using registers.**

Disadvantage: **Register allocation algorithms can be complex.**

Microarchitecture:

Approach: **Use register renaming with more physical than architectural registers.**

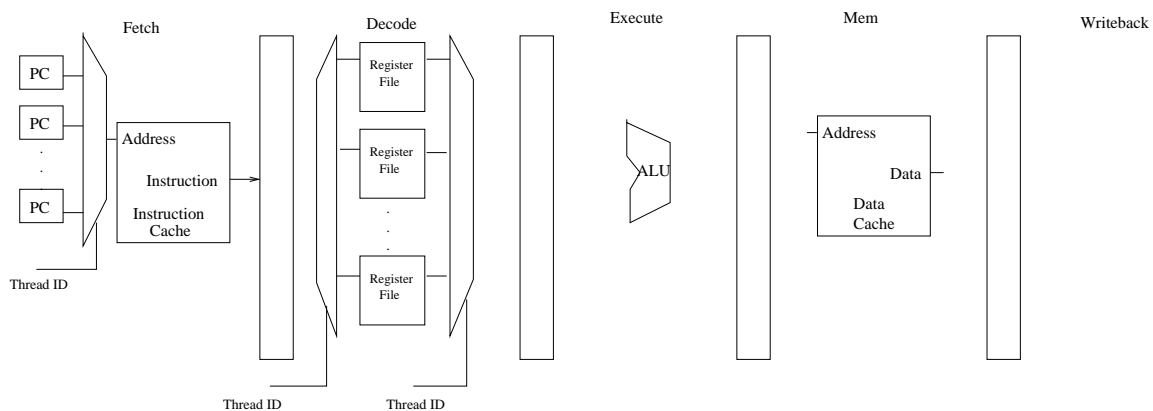
Disadvantage: **Hardware cost (area, power, critical path) for larger register file.**

II. Fine-Grained Multithreading (50 Points)

Consider a design “Machine I” with five pipeline stages: fetch, decode, execute, memory and writeback. Each stage takes 1 cycle. The instruction and data caches have 100% hit rates (i.e., there is never a stall for a cache miss). Branch directions and targets are resolved in the execute stage. The pipeline stalls when a branch is fetched, until the branch is resolved. Dependency check logic is implemented in the decode stage to detect flow dependences. The pipeline does not have any forwarding paths, so it must stall on detection of a flow dependence.

In order to avoid these stalls, we will consider modifying Machine I to use fine-grained multithreading.

- (a) In the five stage pipeline of Machine I shown below, clearly show what blocks you would need to add in each stage of the pipeline, to implement fine-grained multithreading. You can replicate any of the blocks and add muxes. You don’t need to implement the mux control logic (although provide an intuitive name for the mux control signal, when applicable).



- (b) The machine’s designer first focuses on the branch stalls, and decides to use fine-grained multithreading to keep the pipeline busy no matter how many branch stalls occur. What is the minimum number of threads required to achieve this?

3

Why?

Since branches are resolved in the Execute stage, it is necessary that the Fetch stage does not fetch for a thread until the thread’s previous instruction has passed Execute. Hence three threads are needed to cover Fetch, Decode, Execute.

- (c) The machine’s designer now decides to eliminate dependency-check logic and remove the need for flow-dependence stalls (while still avoiding branch stalls). How many threads are needed to ensure that no flow dependence ever occurs in the pipeline?

4

Why?

The designer must ensure that when an instruction is in Writeback, the next instruction in the same thread has not reached Decode yet. Hence, at least 4 threads are needed.

A rival designer is impressed by the throughput improvements and the reduction in complexity that FGMT brought to Machine I. This designer decides to implement FGMT on another machine, Machine II. Machine II is a pipelined machine with the following stages.

Fetch	1 stage
Decode	1 stage
Execute	8 stages (branch direction/target are resolved in the first execute stage)
Memory	2 stages
Writeback	1 stage

Assume everything else in Machine II is the same as in Machine I.

- (d) Is the number of threads required to eliminate branch-related stalls in Machine II the same as in Machine I?

YES NO (Circle one)

If yes, why?

Branches are resolved at the third pipeline stage in both machines, and distance from fetch to branch resolution determines the minimum number of threads to avoid branch stalls.

If no, how many threads are required?

N/A

- (e) What is the minimum CPI (i.e., maximum performance) of each thread in Machine II when this minimum number of threads is used?

3 (if no flow dependence stalls occur)

- (f) Now consider flow-dependence stalls. Does Machine II require the same minimum number of threads as Machine I to avoid the need for flow-dependence stalls?

YES NO (Circle one)

If yes, why?

N/A

If no, how many threads are required?

12 (the Decode, Execute 1 – 8, Memory, and Writeback stages must all have instructions from independent threads.)

- (g) What is the minimum CPI of each thread when this number of threads (to cover flow-dependence stalls) is used?

12

- (h) After implementing fine grained multithreading, the designer of Machine II optimizes the design and compares the pipeline throughput of the original Machine II (without FGMT) and the modified Machine II (with FGMT) both machines operating at their maximum possible frequency, for several code sequences. On a particular sequence that has no flow dependences, the designer is surprised to see that the new Machine II (with FGMT) has lower overall throughput (number of instructions retired by the pipeline per second) than the old Machine II (with no FGMT). Why could this be? Explain concretely.

The additional FGMT-related logic (MUXes and thread selection logic) could increase the critical path length, which will reduce maximum frequency and thus performance.

III. Branch Prediction (60 Points)

Consider the following high level language code segment:

```
int array[1000] = { /* random values */ };
int sum1 = 0, sum2 = 0, sum3 = 0, sum4 = 0;

for (i = 0; i < 1000; i ++) // LOOP BRANCH
{
    if (i % 4 == 0) // IF CONDITION 1
        sum1 += array[i]; // TAKEN PATH
    else
        sum2 += array[i]; // NOT-TAKEN PATH

    if (i % 2 == 0) // IF CONDITION 2
        sum3 += array[i]; // TAKEN PATH
    else
        sum4 += array[i]; // NOT-TAKEN PATH
}
```

Your task is to find the prediction accuracy for the LOOP BRANCH (which is taken whenever the loop repeats, and not taken when the loop exits) and both of the IF CONDITION branches inside the loop (which are taken when the if-condition is *true*, and not taken when the if-condition is *false*), for different kinds of branch predictors.

Show all your work for credit.

- (a) What is the prediction accuracy for each individual branch using a per-branch last-time predictor (assume that every per-branch counter starts at “not-taken”) for the following branches?

LOOP BRANCH:

998/1000 = 99.8%. The branch is mispredicted the first time it's executed.

IF CONDITION 1:

$500/1000 \times 100 = 50\%$.

IF CONDITION 2:

0%. The branch changes direction every time it's executed.

- (b) What is the prediction accuracy when a per-branch 2-bit saturating counter-based predictor is used (assume that every per-branch counter starts at “strongly not-taken”), for the following branches?

LOOP BRANCH:

$997/1000 \times 100 = 99.7\%$. The branch is mispredicted the first two times it's executed and the last time (when the loop exits).

IF CONDITION 1:

750/1000 \times 100 = 75%. The branch repeats the pattern T N N N T N N N ... The saturating counter moves between “strongly not-taken” and “weakly not-taken” (once out of every four predictions, after the branch is actually taken), and the prediction is always not-taken.

IF CONDITION 2:

500/1000 \times 100 = 50%. The branch repeats the pattern T N T N ... The saturating counter moves between “strongly not-taken” and “weakly not-taken” every prediction, and every prediction is not-taken, which is correct half the time.

- (c) What is the prediction accuracy of IF CONDITION 1 and IF CONDITION 2, when the counter starts at
- (i) “weakly not-taken”?

IF CONDITION 1:

749/1000 \times 100 = 74.9%.

IF CONDITION 2:

0%. The predictor oscillates between “weakly not-taken” and “weakly taken”

- (ii) “weakly taken”?

IF CONDITION 1:

749/1000 \times 100 = 74.9%. The branch’s pattern is T N N N T N N N ... The first four iterations, the predictor’s counter starts at weakly taken, moves to strongly taken, then weakly taken, then weakly not-taken. It thus predicts T T T N. Starting for the next group of four branches, the counter is strongly not-taken, weakly not-taken, strongly not-taken, strongly not-taken, yielding N N N N predictions. Thus $249 \times 3 + 2 = 749$ correct predictions are made.

IF CONDITION 2:

500/1000 \times 100 = 50%. The branch pattern is T N T N ... The counter is at weakly-taken for the first branch, moves to strongly-taken, then weakly taken, etc. Thus the predictions are T T T T ... so the predictor is 50% accurate.

Note that for IF CONDITION 2, the prediction accuracy is strongly dependent on the initial state of the branch predictor.

- (d) What is the prediction accuracy when a two-level global branch predictor with a two-bit global history register and a separate pattern history table per branch, consisting of 2-bit saturating counters for every entry, is used? Assume that both bits of the global history register are initialized to “not-taken” and

the 2-bit saturating counters in the pattern history tables are initialized to “strongly not-taken”. When calculating prediction accuracy, ignore the first 500 loop iterations.

LOOP BRANCH:

$499/500 \times 100 = 99.8\%$ The predictor has already entered a “strongly taken” state for all global histories for this branch after 500 loop iterations. It always predicts taken. Only the last loop iteration (for which the loop branch is not taken) results in a misprediction.

IF CONDITION 1:

75%. Correlation between IF CONDITION 2 (of the previous loop iteration) and IF CONDITION 1 helps out here. The global branch history will include IF CONDITION 2’s last result, as well as the loop branch, which will always be taken. There is thus effectively a separate saturating counter for even loop iterations and odd loop iterations. When the last IF CONDITION 2 branch was not taken, IF CONDITION 1’s pattern is N T N T ... There are never two consecutive T’s in this subsequence so the saturating counter oscillates between strongly not-taken and weakly-not taken, resulting in all not-taken predictions and 50% accuracy. When the last IF CONDITION 2 branch was taken, IF CONDITION 1’s pattern is N N N N ... which will result in 100% accuracy. Hence the overall accuracy is 75%.

IF CONDITION 2:

75%. Similar to the above, IF CONDITION 1’s branch history correlates with IF CONDITION 2’s, and IF CONDITION 2 effectively uses two saturating counters, conditional on IF CONDITION 1’s outcome. When IF CONDITION 1 is not taken, IF CONDITION 2’s pattern is N T N N T N N T N ... (these branches result from iterations 1, 2, 3, 5, 6, 7, 9, 10, 11, ...). The predictor will oscillate between weakly and strongly not-taken, but will always predict N. When IF CONDITION 1 is taken, IF CONDITION 2 is also always taken, so the predictor will have 100% accuracy in this case. Altogether, only 1 out of every 4 iterations is mispredicted, so overall accuracy is 75%.

IV. What Does This Mystery Instruction Do? (50 Points)

An engineer implemented the mystery instruction described below on the LC-3b. Unfortunately, we do not know what this engineer was thinking, and we can't figure out what the instruction does. It is your job to determine this. The mystery instruction is encoded as:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1010				DR			SR1			1	0	1	1	1	1

The instruction is only defined if the value of SR1 is not equal to zero.

The modifications we make to the LC-3b datapath and the microsequencer are highlighted in the attached figures (see the next three pages).

We also provide the original LC-3b state diagram, in case you need it.

The additional control signals are

GateLSHF/1: NO, YES

LD.TEMP/1: NO, LOAD

ALUK/3: AND, ADD, XOR, PASSA, PASSB, DECA (Decrement A)

COND/3:

*COND*₀ ;Unconditional

*COND*₁ ;Memory Ready

*COND*₂ ;Branch

*COND*₃ ;Addressing mode

*COND*₄ ;Mystery

The microcode for the instruction is given in the table below.

State	Cond	J	Asserted Signals
001010 (10)	COND0	101000	ALUK = PASSB, GateALU, LD.REG, DRMUX = DR (IR[11:9])
101000 (40)	COND4	010010	ALUK = PASSA, GateALU, LD.TEMP, SR1MUX = SR1 (IR[8:6])
110010 (50)	COND0	110011	ALUK = DECA, GateALU, LD.REG, SR1MUX = DR, DRMUX = DR (IR[11:9])
110011 (51)	COND4	010010	GateLSHF, LD.TEMP

Describe what this instruction does.

The instruction finds the position of the most significant set bit in SR1 and places this in DR.

Mathematically, this can be expressed as $DR = \lfloor \log_2 SR1 \rfloor$

Note that this instruction is semantically the same as FINDFIRST in the VAX ISA.

Specifically, here is what happens in each state:

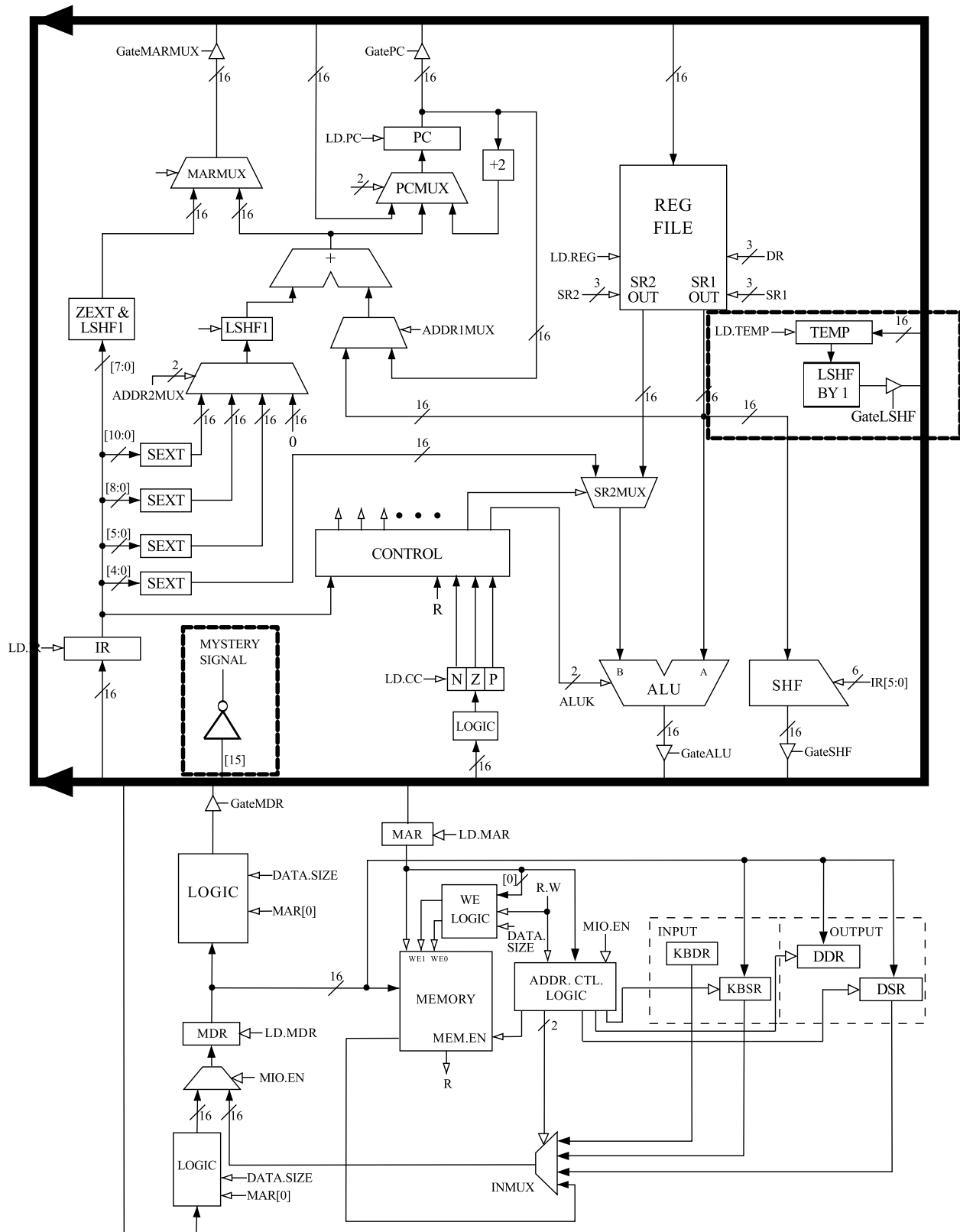
State 10: DR = 15

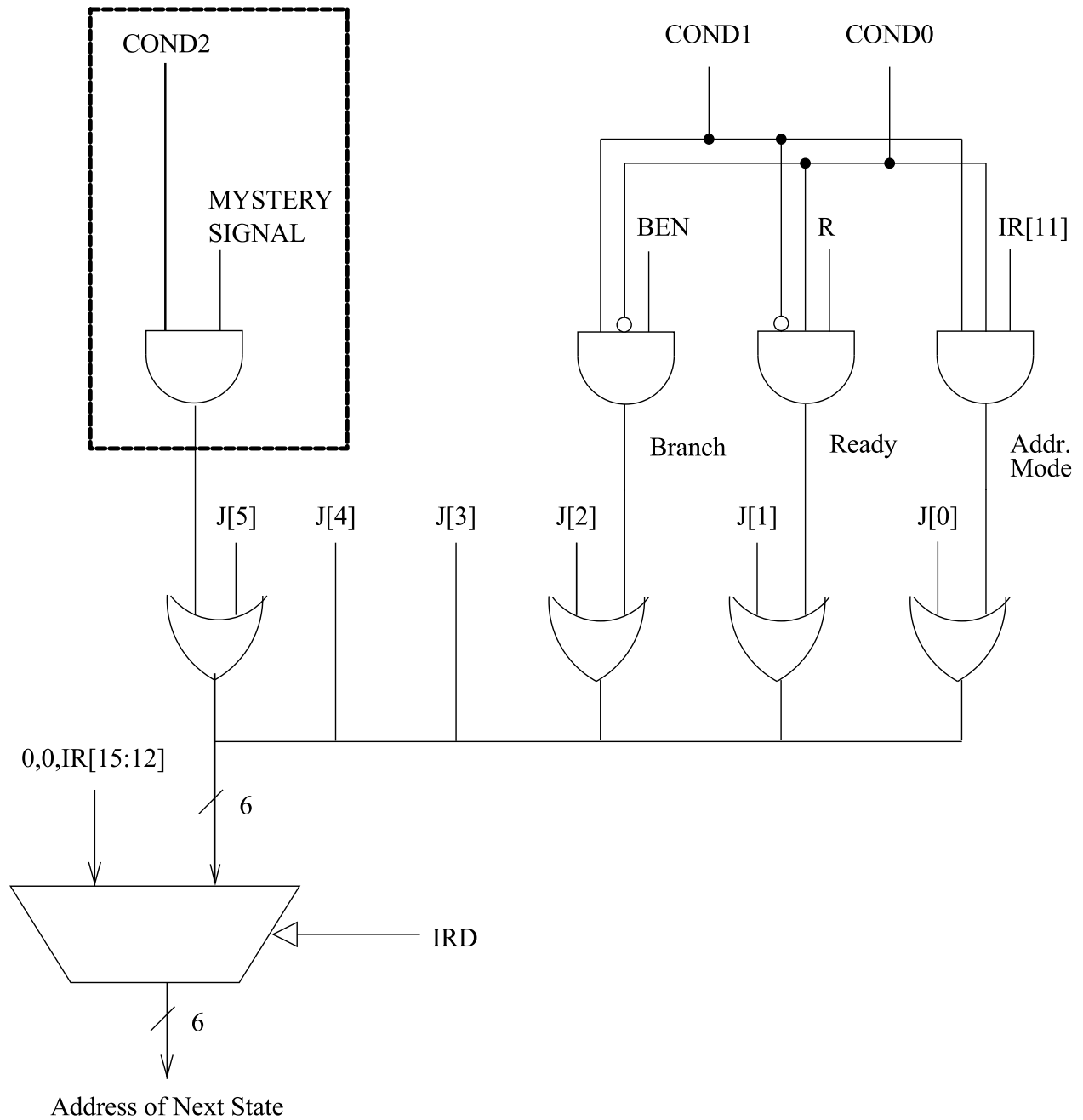
State 40: TEMP = SR1

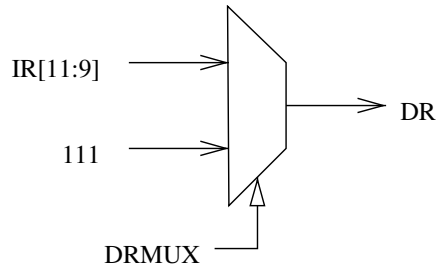
if (SR1 is negative)

Go to State 18/19 (Fetch)

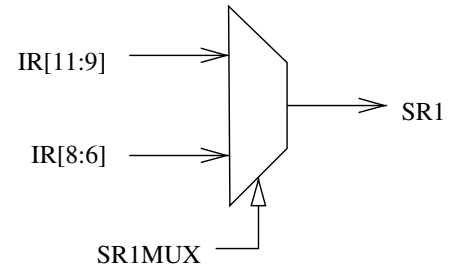
```
    else
      Go to State 50
State 50: DR = DR - 1
State 51: Left Shift TEMP
      if (TEMP is negative)
        Go to State 18/19 (Fetch)
      else
        Go to State 50
```



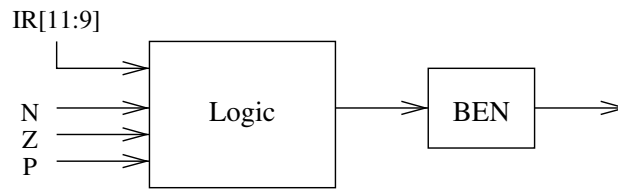




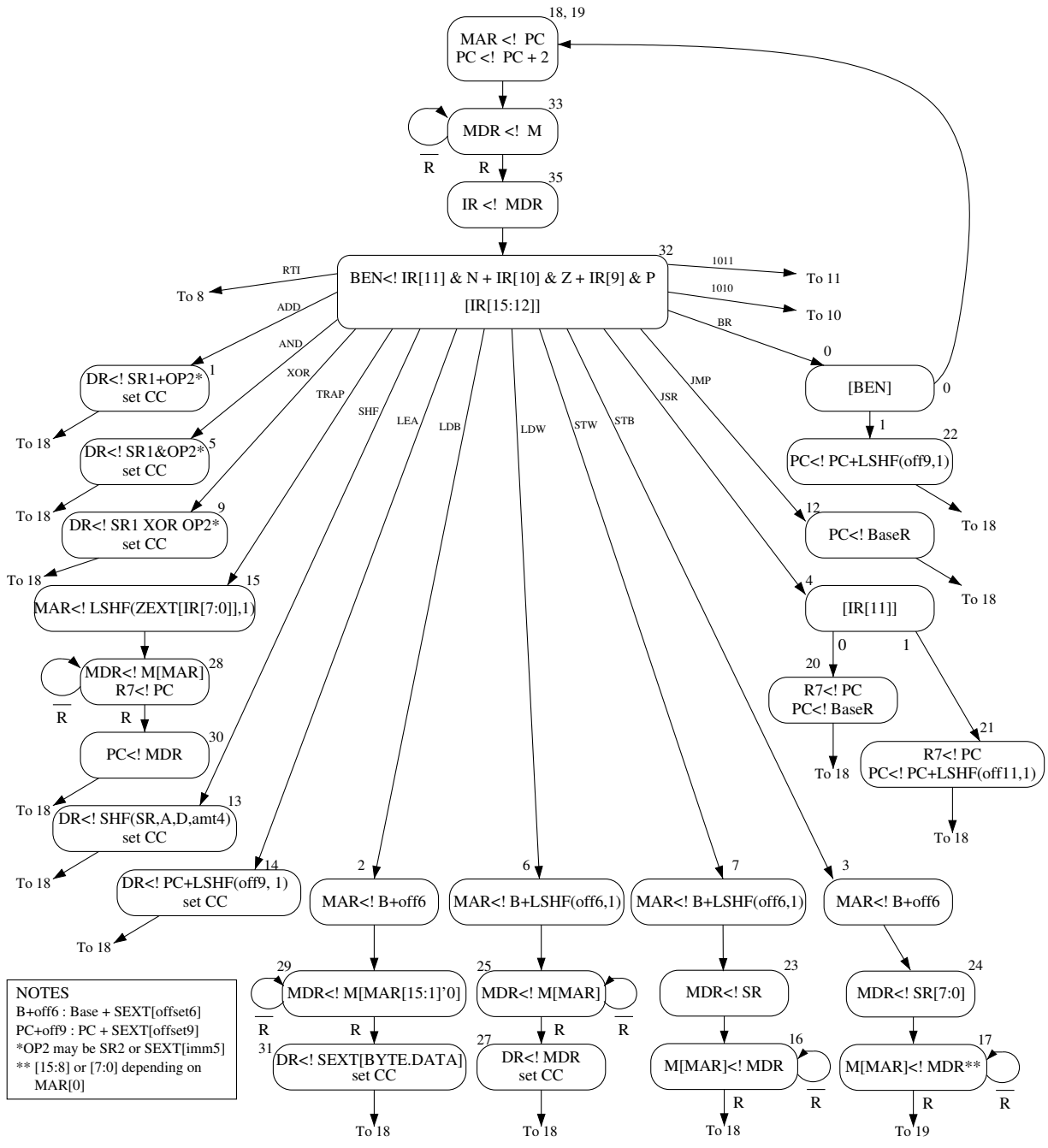
(a)



(b)



(c)



V. Out-of-Order Execution (50 Points)

In this problem, we will give you the state of the Register Alias Table (RAT) and Reservation Stations (RS) for a Tomasulo-like out-of-order execution engine. Your job is to determine the original sequence of **five instructions** in program order.

The out-of-order machine in this problem behaves as follows:

- The frontend of the machine has a one-cycle fetch stage and a one-cycle decode stage. The machine can fetch one instruction per cycle, and can decode one instruction per cycle.
- The machine dispatches one instruction per cycle into the reservation stations, in program order. Dispatch occurs during the decode stage.
- An instruction always allocates the first reservation station that is available (in top-to-bottom order) at the required functional unit.
- When a value is captured (at a reservation station) or written back (to a register) in this machine, the old tag that was previously at that location is *not cleared*; only the valid bit is set.
- When an instruction in a reservation station finishes executing, the reservation station is cleared.
- Both the adder and multiplier are fully pipelined. Add instructions take 2 cycles. Multiply instructions take 4 cycles.
- When an instruction completes execution, it broadcasts its result, and dependent instructions can begin execution in the next cycle if they have all operands available.
- When multiple instructions are ready to execute at a functional unit, the *oldest* ready instruction is chosen.

Initially, the machine is empty. Five instructions then are fetched, decoded, and dispatched into reservation stations, before any instruction executes. Then, one instruction completes execution. Here is the state of the machine at this point, after the single instruction completes:

RAT

Reg	V	Tag	Value
R0	1		20
R1	1		50
R2	0	A	37
R3	1	X	500
R4	0	Y	255
R5	1		17
R6	0	Z	73
R7	1		10

	Src 1			Src 2		
	Tag	V	Value	Tag	V	Value
A	X	1	500	Y	0	-
B	-	1	20	-	1	17
C						



	Src 1			Src 2		
	Tag	V	Value	Tag	V	Value
X						
Y	-	1	50	-	1	37
Z	A	0	-	B	0	-



- (a) Give the five instructions that have been dispatched into the machine, in program order. The source registers for the first instruction can be specified in either order. Give instructions in the following format: “opcode destination \leftarrow source1, source2.”

MUL R3 \leftarrow R1, R7

MUL R4 \leftarrow R1, R2

ADD R2 \leftarrow R3, R4

ADD R6 \leftarrow R0, R5

MUL R6 \leftarrow R2, R6

- (b) Now assume that the machine flushes all instructions out of the pipeline and restarts execution from the first instruction in the sequence above. Show the full pipeline timing diagram below for the sequence of five instructions that you determined above, from the fetch of the first instruction to the writeback of the last instruction. Assume that the machine stops fetching instructions after the fifth instruction.

As we saw in class, use “F” for fetch, “D” for decode, “E1,” “E2,” “E3,” and “E4” to signify the first, second, third and fourth cycles of execution for an instruction (as required by the type of instruction), and “W” to signify writeback. You may or may not need all columns shown.

Cycle:	1	2	3	4	5	6	7	8	9	10	11	12	13	14
MUL R3 \leftarrow R1, R7	F	D	E1	E2	E3	E4	W							
MUL R4 \leftarrow R1, R2		F	D	E1	E2	E3	E4	W						
ADD R2 \leftarrow R3, R4			F	D				E1	E2	W				
ADD R6 \leftarrow R0, R5				F	D	E1	E2	W						
MUL R6 \leftarrow R2, R6					F	D				E1	E2	E3	E4	W

Finally, show the state of the RAT and reservation stations after **8 cycles** in the blank figures below.

RAT

Reg	V	Tag	Value
R0	1		20
R1	1		50
R2	0	A	37
R3	1	X	500
R4	1	Y	1850
R5	1		17
R6	0	Z	73
R7	1		10

	Src 1			Src 2		
	Tag	V	Value	Tag	V	Value
A	X	1	500	Y	1	1850
B						
C						



	Src 1			Src 2		
	Tag	V	Value	Tag	V	Value
X						
Y						
Z	A	0	-	B	1	37



VI. Data Dependence Handling - A Different Way (50 Points)

When handling dependent instructions, an alternative to data forwarding is *value prediction*, as we discussed in class. You are going to use value prediction to resolve flow dependences in a processor with a 10-stage pipeline (Fetch, two stages of Decode, five stages of Execute, one stage of Memory, one stage of Writeback). Assume that the processor never stalls for memory, that it assumes branches are not taken until they are resolved, and that it resolves branches in the last stage of Execute. The value predictor is placed in the Decode stage, and produces a value prediction for a register whose value is not available. The processor is designed so that it *never stalls due to data dependences*, but always value-predicts to resolve stalls. This works as follows:

- When an instruction passes through the second stage of Decode and *reads* a register, data dependence detection logic detects whether an older instruction later in the pipeline is *writing* to that register.
- When a data dependence is detected, the processor uses its *value predictor* to predict the value that the register will eventually have, and (i) feeds that value to the dependent instruction, (ii) records the value for later checking. Therefore, the dependent instruction can proceed without a stall using the predicted value.
- When an instruction that *writes* a register reaches the Writeback stage, it writes its value back, and also checks the value against predictions that were made for that value. If a prediction was incorrect, then (i) the entire pipeline is flushed, and (ii) fetch restarts in the subsequent cycle from the first instruction that received a mispredicted value.

Assume that the processor is designed so that it can handle multiple predictions “in flight” for a single register at different instructions in the pipeline, and that a new prediction is made each time a register is read.

- (a) Assume that the processor runs a long sequence of straight-line code (no branches) in which every instruction depends only on the previous instruction. What is the *worst-case instruction throughput* (cycles per instruction) if every value is predicted incorrectly?

10 cycles per instruction.

- (b) For the same parameters and same straight-line code with the same sequential flow dependences, what is the instruction throughput for a processor that does not use value prediction, but, implements data forwarding and forwards results whenever possible (assume that the value produced by an instruction is available after the last Execute pipeline stage)?

5 cycles per instruction.

- (c) For what value-prediction accuracy would the value predictor-based design have the same instruction throughput on this straight-line code as the forwarding-based design?

$CPI = 1 + \text{misprediction bubble cycles} * \text{fraction of mispredictions}$
 $CPI = 1 + 9 * m$
 $5 = 1 + 9m$
 $m = \frac{4}{9} = 44.4\%$
 Hence, 44.4% or 4/9 misprediction rate.

- (d) You are simulating your value prediction-based system with the following benchmark loop, which sums an array of bytes. Assume that the loop will iterate a very large number of times.

```

LOOP:  addiu $t0, $t0, 1      # $t0 is loop counter
        beq $t0, $t2, OUT    # $t2 is count
        addu $t1, $t0, $a0   # $a0 points to a byte array
        lbu $t1, 0($t1)     # load the byte at array[$t0]
        add $v0, $v0, $t1    # add it to sum
        j LOOP
OUT:
  
```

OUT:

You first try a simple *last-value predictor* which, for each prediction made for a given operand for a given instruction, predicts the value that the operand had the *previous* time the same instruction was executed. If the predictor has never seen an instruction execute before, it predicts 0 for any predictions that are necessary.

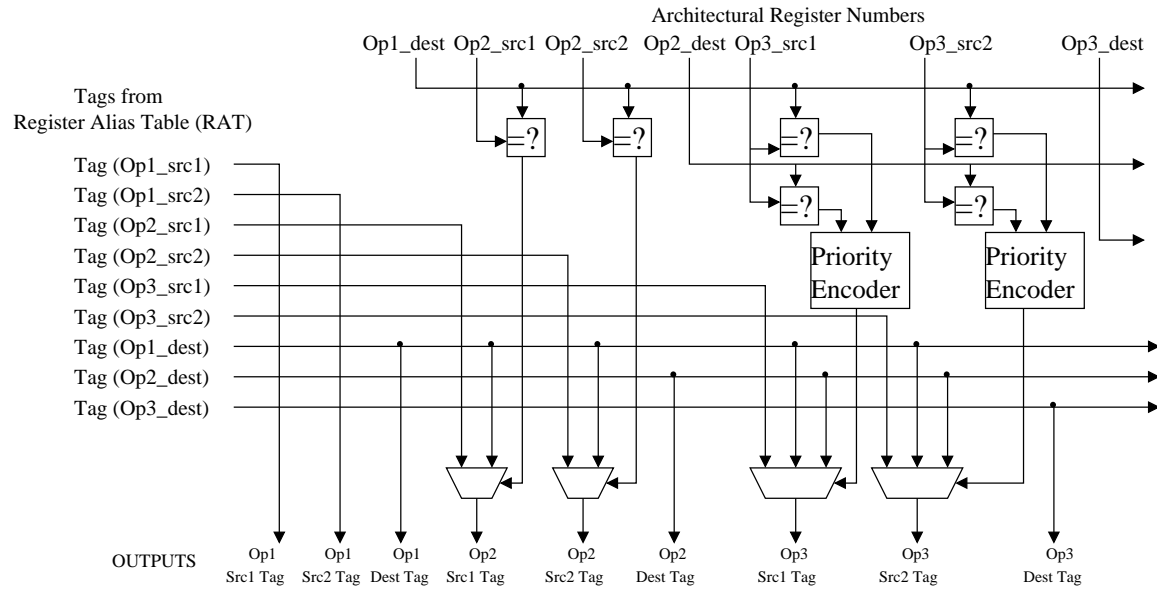
What is the prediction accuracy of this predictor on the two uses of the loop counter \$t0 (in the second instruction `beq` and the third instruction `addu`)?

0%.

- (e) Can you describe a simple value predictor that would give almost 100% accuracy on these two predictions?

Build a *stride predictor* that recognizes increment-by-one (or increment-by-N) patterns and predicts the next value in the sequence.

VII. What Does This Logic Do? (20 Points)



In each of the priority encoders, the input on the left takes precedence.

What does the above logic do? (In less than 20 words.)

Register rename logic of a superscalar processor

VIII. BONUS: CISC Tradeoffs (40 Points)

Many CISC processors have instructions that are designed to support high-level language function prologues and epilogues. For the purpose of this question, let us define two instructions that are designed to make function prologues and epilogues more efficient in the standard MIPS calling convention: *ENTER* and *LEAVE*. Let us define the instructions as follows:

- *ENTER imm16* decrements the stack pointer to create space, and then stores the following callee-save registers to the stack: \$ra, \$s0 – \$s7, \$fp (frame pointer). It copies the stack pointer at this point to \$fp (frame pointer). It takes an unsigned 16-bit immediate, *imm16*, and further decrements the stack pointer by this amount to create space for local variables.
- *LEAVE* moves the value in \$fp into the stack pointer \$sp. It then restores all of the callee-save registers that *ENTER* had saved, and increments \$sp to pop these values off the stack.

Thus, a simple subroutine could be written as:

```
my_func:
    enter 4          # one local variable
    addiu $s0, $a0, 0 # save argument 0
    jal func2        # call func2() with my arguments
    addu $v0, $v0, $s0 # return func2()'s return value plus saved arg
    leave           # restore callee-saves
    jr $ra
```

Answer the following questions:

- (a) Write MIPS assembly code that implements the behavior of the *ENTER* and *LEAVE* instructions.

ENTER:	LEAVE:
sw \$s0, -40(\$sp)	addiu \$sp, \$fp, 40
sw \$s1, -36(\$sp)	lw \$s0, -40(\$sp)
sw \$s2, -32(\$sp)	lw \$s1, -36(\$sp)
sw \$s3, -28(\$sp)	lw \$s2, -32(\$sp)
sw \$s4, -24(\$sp)	lw \$s3, -28(\$sp)
sw \$s5, -20(\$sp)	lw \$s4, -24(\$sp)
sw \$s6, -16(\$sp)	lw \$s5, -20(\$sp)
sw \$s7, -12(\$sp)	lw \$s6, -16(\$sp)
sw \$ra, -8(\$sp)	lw \$s7, -12(\$sp)
sw \$fp, -4(\$sp)	lw \$ra, -8(\$sp)
addiu \$fp, \$sp, -40	lw \$fp, -4(\$sp)
addiu \$sp, \$fp, -imm16	

For (b), (c), and (d), assume a 5-stage pipelined processor (fetch, decode, execute, memory, writeback, as described in class) where all memory accesses take one cycle (i.e., there are no caches). Say that the new ENTER/LEAVE instructions are implemented on this processor.

- (b) Assume ENTER/LEAVE is 1% faster than the equivalent code you wrote above. The compiler must save N registers that are clobbered by a function body, and it can choose between using ENTER/LEAVE or a sequence of MIPS instructions (similar to what you wrote) that saves only the registers clobbered by the function body. For what value of N will ENTER/LEAVE result in better performance than the sequence of MIPS instructions?

10. An ENTER/LEAVE pair replaces 23 MIPS instructions. The new instructions are as fast as 22.77 MIPS instructions. If the compiler needs to save only 9 registers, this requires 21 instructions. Hence ENTER/LEAVE are only faster if 10 registers are required to be saved.

- (c) Now, assume ENTER/LEAVE is 50% faster than the equivalent code you wrote above. Same question as part (b)?

5. Now ENTER/LEAVE are as fast as 11.5 MIPS instructions. If the compiler had to save only 4 registers, this would require 11 MIPS instructions. Hence it must save 5 or more registers for ENTER/LEAVE to be faster.

- (d) Now, assume ENTER/LEAVE is 10x faster than the equivalent code you wrote above. Same question as part (b)?

1. Now ENTER/LEAVE are as fast as 2.3 MIPS instructions. Saving even one register requires 5 instructions. Hence if the compiler needs to save only one register, ENTER/LEAVE is faster.

MIPS Instruction Set Summary (Subset)

Opcodes	Example Assembly	Semantics
add	add \$1, \$2, \$3	\$1 = \$2 + \$3
sub	sub \$1, \$2, \$3	\$1 = \$2 - \$3
add immediate	addi \$1, \$2, 100	\$1 = \$2 + 100
add unsigned	addu \$1, \$2, \$3	\$1 = \$2 + \$3
subtract unsigned	subu \$1, \$2, \$3	\$1 = \$2 - \$3
add imm. Unsigned	addiu \$1, \$2, 100	\$1 = \$2 + 100
multiply	mult \$2, \$3	hi, lo = \$2 * \$3
multiply unsigned	multu \$2, \$3	hi, lo = \$2 * \$3
divide	div \$2, \$3	lo = \$2/\$3, hi = \$2 mod \$3
divide unsigned	divu \$2, \$3	lo = \$2/\$3, hi = \$2 mod \$3
move from hi	mfhi \$1	\$1 = hi
move from low	mflo \$1	\$1 = lo
and	and \$1, \$2, \$3	\$1 = \$2 & \$3
or	or \$1, \$2, \$3	\$1 = \$2 \$3
and immediate	andi \$1, \$2, 100	\$1 = \$2 & 100
or immediate	ori \$1, \$2, 100	\$1 = \$2 100
shift left logical	sll \$1, \$2, 10	\$1 = \$2 << 10
shift right logical	srl \$1, \$2, 10	\$1 = \$2 >> 10
load word	lw \$1, \$2(100)	\$1 = ReadMem32(\$2 + 100)
store word	sw \$1, \$2(100)	WriteMem32(\$2 + 100, \$1)
load halfword	lh \$1, \$2(100)	\$1 = SignExt(ReadMem16(\$2 + 100))
store halfword	sh \$1, \$2(100)	WriteMem16(\$2 + 100, \$1)
load byte	lb \$1, \$2(100)	\$1 = SignExt(ReadMem8(\$2 + 100))
store byte	sb \$1, \$2(100)	WriteMem8(\$2 + 100, \$1)
load upper immediate	lui \$1, 100	\$1 = 100 << 16
branch on equal	beq \$1, \$2, Label	if (\$1 == \$2) goto Label
branch on not equal	bne \$1, \$2, Label	if (\$1 != \$2) goto Label
set on less than	slt \$1, \$2, \$3	if (\$2 < \$3) \$1 = 1 else \$1 = 0
set on less than immediate	slti \$1, \$2, 100	if (\$2 < 100) \$1 = 1 else \$1 = 0
set on less than unsigned	sltu \$1, \$2, \$3	if (\$2 < \$3) \$1 = 1 else \$1 = 0
set on less than immediate	sltui \$1, \$2, 100	if (\$2 < 100) \$1 = 1 else \$1 = 0
jump	j Label	goto Label
jump register	jr \$31	goto \$31
jump and link	jal Label	\$31 = PC + 4; goto Label

SCRATCH PAD

SCRATCH PAD

SCRATCH PAD