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:

Problem I (50 Points)	:	
Problem II (50 Points)	:	
Problem III (60 Points)	:	
Problem IV (50 Points)	:	
Problem V (50 Points)	:	
Problem VI (50 Points)	:	
Problem VII (20 Points)	:	
Bonus Problem VIII (40 Points)	:	
Legibility and Name (5 Points)	:	
Total (335 + 40 Points)	:	

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):

2) Data Dependencies (5 Points)

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

3) Program Counter (5 Points)

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

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?

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) _____ Two-level global branch prediction

(ii) _____ Predicated instruction execution

(iii) _____ A floating-point unit that uses wide floating-point values for additional accuracy

(iv) _____ A 32-bit wide data bus to memory

(v) _____ 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.

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 _____ 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 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?

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)?

(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:

Disadvantage:

Compiler:

Approach:

Disadvantage:

Microarchitecture:

Approach:

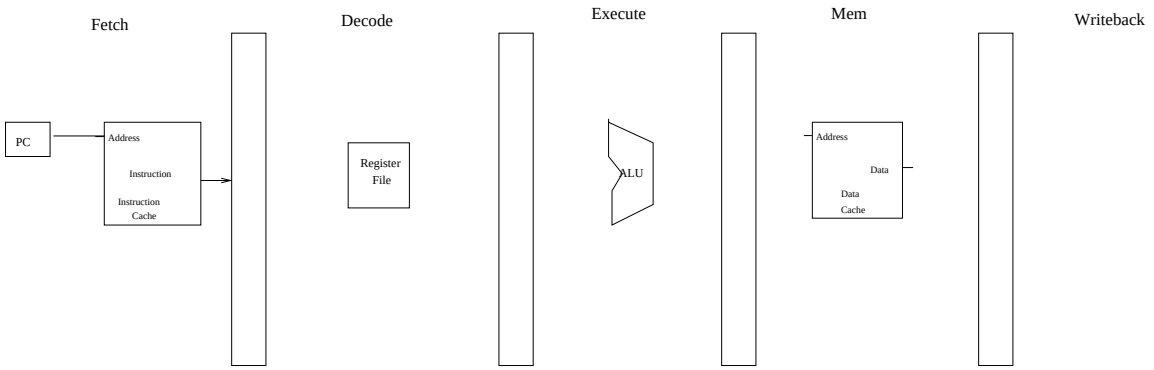
Disadvantage:

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?

Why?

(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?

Why?

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?

If no, how many threads are required?

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

(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?

If no, how many threads are required?

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

(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.

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:

IF CONDITION 1:

IF CONDITION 2:

(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:

IF CONDITION 1:

IF CONDITION 2:

(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:

IF CONDITION 2:

(ii) "weakly taken"?

IF CONDITION 1:

IF CONDITION 2:

(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:

IF CONDITION 1:

IF CONDITION 2:

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

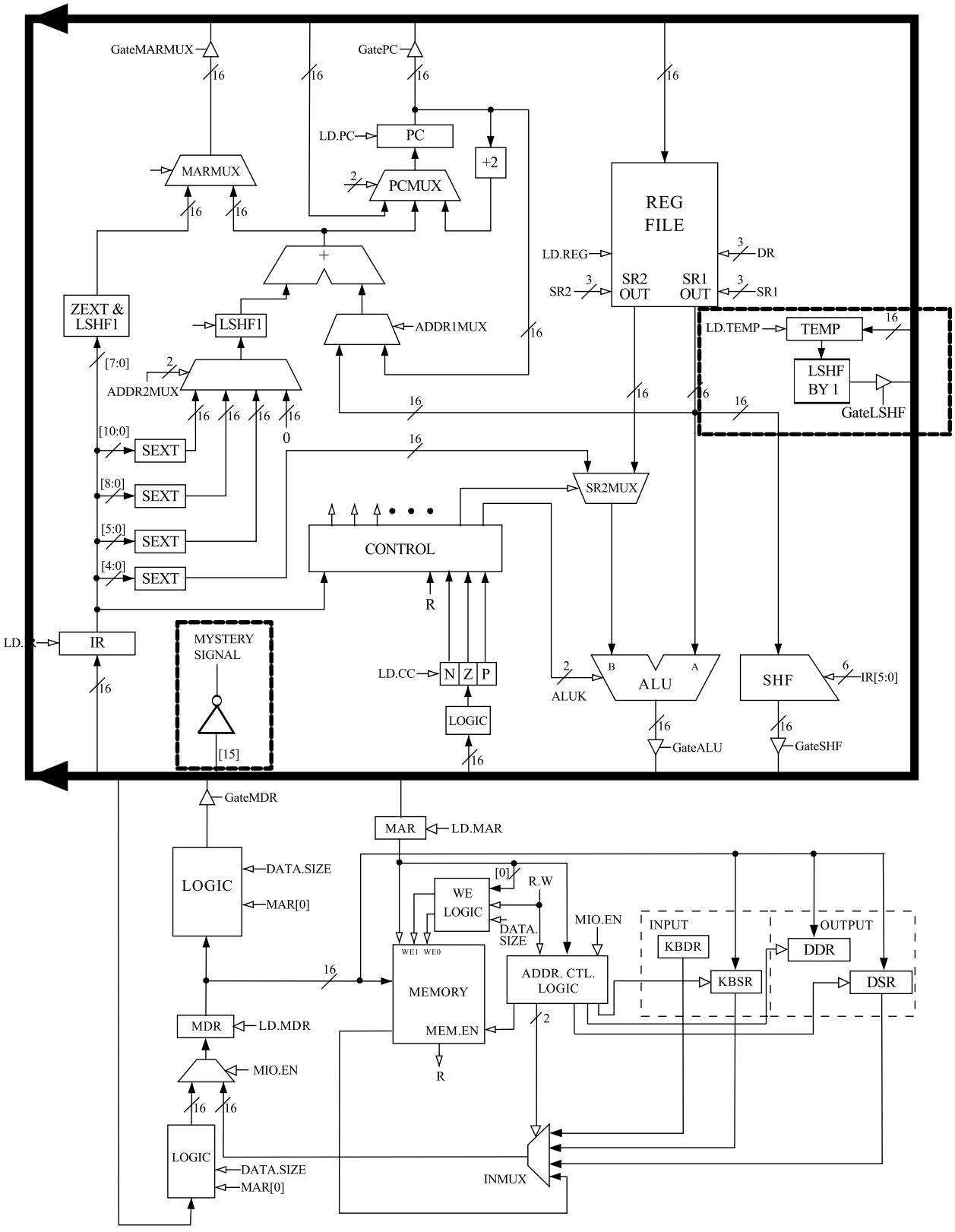
ALUK/3: AND, ADD, XOR, PASSA, PASSB, DECA (Decrement A)

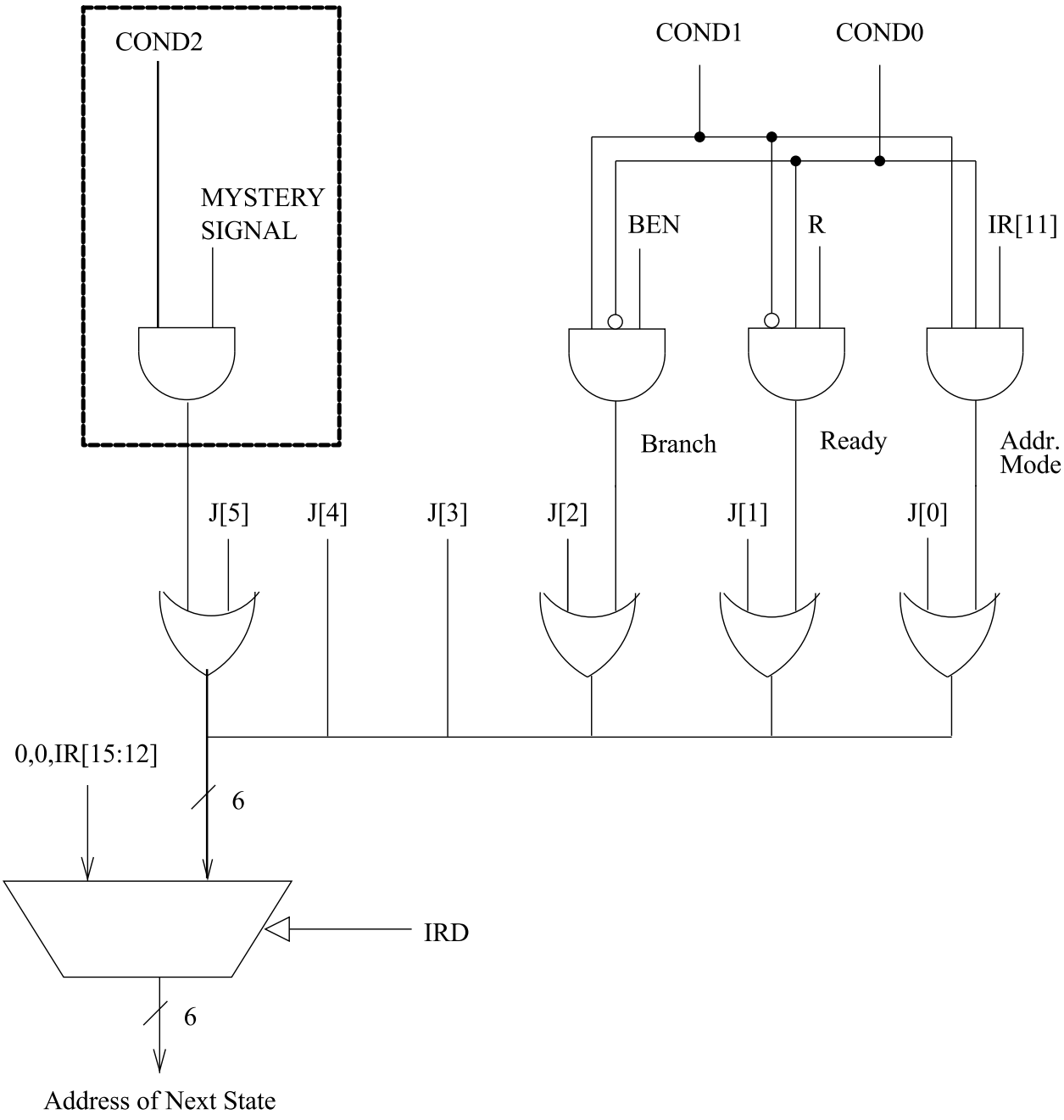
COND/3:
$COND_0$;Unconditional
$COND_1$;Memory Ready
$COND_2$;Branch
$COND_3$;Addressing mode
$COND_4$;Mystery

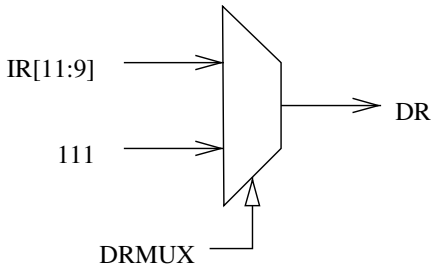
The microcode for the instruction is given in the table below.

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

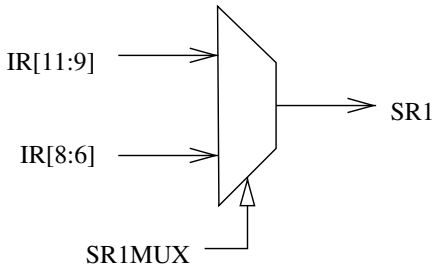
Describe what this instruction does.


GateMARMUX
GatePC
16
LD.PC
PC
+2
MARMUX
2
PCMUX
REG
FILE
LD.REG
3
DR
SR2
SR2
OUT
SR1
OUT
SR1
ZEXT &
LSHF1
LSHF1
ADDR1MUX
LD.TEMP
TEMP
LSHF
BY 1
GateLSHF
[7:0]
2
ADDR2MUX
[10:0]
SEXT
[8:0]
SEXT
[5:0]
SEXT
[4:0]
SEXT
0
SR2MUX
CONTROL
R
LD.IR
IR
MYSTERY
SIGNAL
[15]
LD.CC
N
Z
P
LOGIC
2
ALUK
B
A
ALU
SHF
6
IR[5:0]
GateALU
GateSHF
GateMDR
MAR
LD.MAR
LOGIC
DATA.SIZE
MAR[0]
[0]
R.W
WE
LOGIC
DATA.
SIZE
MIO.EN
INPUT
KBDR
OUTPUT
DDR
WE1
WE0
MEMORY
MEM.EN
ADDR. CTL.
LOGIC
KBSR
DSR
MDR
LD.MDR
MIO.EN
LOGIC
DATA.SIZE
MAR[0]
INMUX

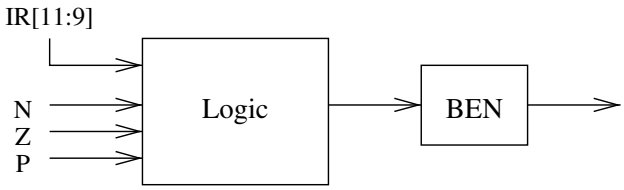

COND2
MYSTERY
SIGNAL
COND1
COND0
BEN
R
IR[11]
Branch
Ready
Addr.
Mode
J[5]
J[4]
J[3]
J[2]
J[1]
J[0]
0,0,IR[15:12]
6
IRD
6
Address of Next State



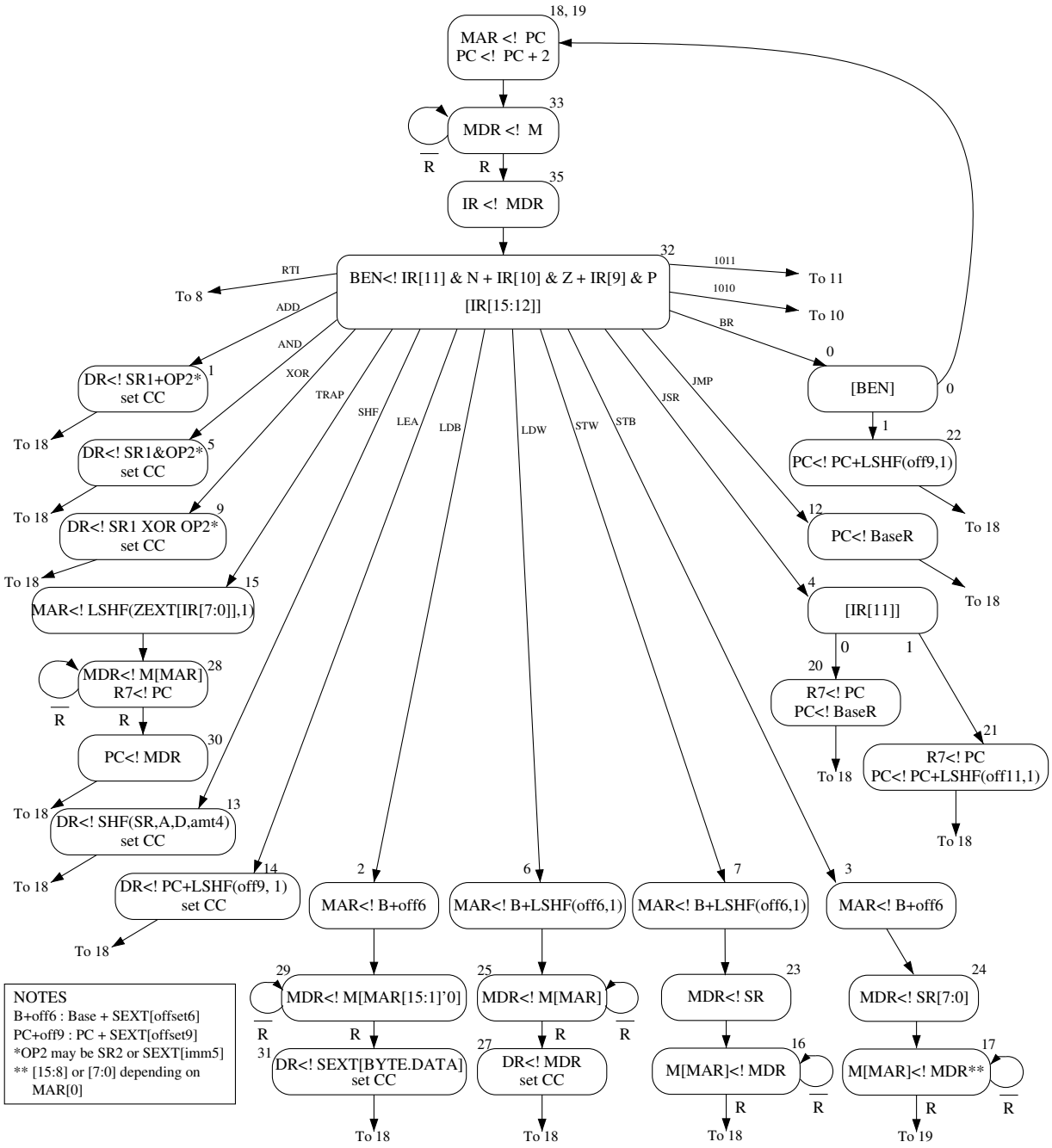
(a)



(b)



(c)


18, 19
MAR <! PC
PC <! PC + 2
33
MDR <! M
R
R
35
IR <! MDR
32
BEN<! IR[11] & N + IR[10] & Z + IR[9] & P
[IR[15:12]]
RTI
To 8
ADD
AND
XOR
TRAP
SHF
LEA
LDB
LDW
STW
STB
JSR
JMP
1011
To 11
1010
To 10
BR
0
[BEN]
0
1
22
PC<! PC+LSHF(off9,1)
To 18
12
PC<! BaseR
To 18
4
[IR[11]]
0
1
20
R7<! PC
PC<! BaseR
To 18
21
R7<! PC
PC<! PC+LSHF(off11,1)
To 18
1
DR<! SR1+OP2*
set CC
To 18
5
DR<! SR1&OP2*
set CC
To 18
9
DR<! SR1 XOR OP2*
set CC
To 18
15
MAR<! LSHF(ZEXT[IR[7:0]],1)
28
MDR<! M[MAR]
R7<! PC
R
R
30
PC<! MDR
To 18
13
DR<! SHF(SR,A,D,amt4)
set CC
To 18
14
DR<! PC+LSHF(off9, 1)
set CC
To 18
2
MAR<! B+off6
6
MAR<! B+LSHF(off6,1)
7
MAR<! B+LSHF(off6,1)
3
MAR<! B+off6
29
MDR<! M[MAR[15:1]'0]
R
R
31
DR<! SEXT[BYTE.DATA]
set CC
To 18
25
MDR<! M[MAR]
R
R
27
DR<! MDR
set CC
To 18
23
MDR<! SR
16
M[MAR]<! MDR
R
R
To 18
24
MDR<! SR[7:0]
17
M[MAR]<! MDR**
R
R
To 19
NOTES
B+off6 : Base + SEXT[offset6]
PC+off9 : PC + SEXT[offset9]
*OP2 may be SR2 or SEXT[imm5]
** [15:8] or [7:0] depending on
MAR[0]

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

ADD

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

MUL

	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 ⇐ source1, source2."

		⇐		,	
		⇐		,	
		⇐		,	
		⇐		,	
		⇐		,	

(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	15	16
Instruction:																
Instruction:																
Instruction:																
Instruction:																
Instruction:																

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

RAT

Reg	V	Tag	Value
R0			
R1			
R2			
R3			
R4			
R5			
R6			
R7			

	Src 1			Src 2		
	Tag	V	Value	Tag	V	Value
A						
B						
C						

ADD

	Src 1			Src 2		
	Tag	V	Value	Tag	V	Value
X						
Y						
Z						

MUL

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?

(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)?

(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?

(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.

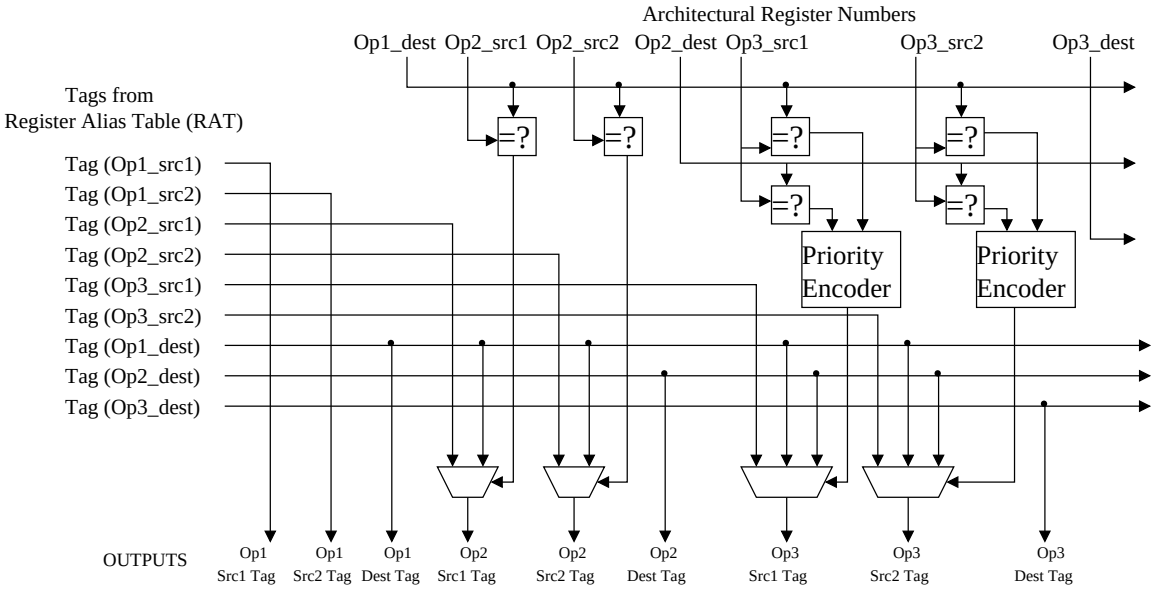
```
LOOP:  addiu $t0, $t0, 1     # $t0 is loop counter
       beq $t0, $t2, OUT     # $t2 is count
       addu $t1, $t0, $a0    # $a0 points to a byte array
       lbu $t1, 0($t1)       # load the byte at array[$t0]
       add $v0, $v0, $t1     # add it to sum
       j LOOP
OUT:
```

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`)?

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

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.)

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 with the minimum number of instructions.

ENTER:	LEAVE:

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?

(c) Now, assume ENTER/LEAVE is 50% faster than the equivalent code you wrote above. Answer the same question as in part (b):

(d) Now, assume ENTER/LEAVE is 10x faster than the equivalent code you wrote above. Answer the same question as in part (b):

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