CMU 18-447 Introduction to Computer Architecture, Spring 2014

Midterm Exam 1

Date: Wed., 3/5

Instructor: Onur Mutlu
TAs: Rachata Ausavarungnirun, Varun Kohli, Xiao Bo Zhao, Paraj Tyle

Name:

Legibility & Name (5 Points):	
Problem 1 (90 Points):	
Problem 2 (35 Points):	
Problem 3 (25 Points):	
Problem 4 (40 Points):	
Problem 5 (35 Points):	
Problem 6 (45 Points):	
Problem 7 (35 Points):	
Bonus (16 Points):	
Total (310 + 16 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 and 50 minutes.
4. Clearly indicate your final answer for each problem.
5. Please show your work when needed.
6. Please write your initials at the top of every page.
7. Please make sure that your answers to all questions (and all supporting work that is required) are contained in the space required.

Tips:

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

1. Potpourri [90 points]

(a) **Static scheduling [9 points]**

Consider the following code example:

```
    mul   R1     <-- R5, R6
    add   R3     <-- R1, R2
    beqz  R3, target
    add   R5     <-- R6, R7
    st    R7(0)  <-- R8
    mul   R8     <-- R8, R8
    ld    R2     <-- R10(0)
    mul   R11    <-- R11, R11
    mul   R12    <-- R12, R12
    jmp   reconv
target:
    add   R13    <-- R13, R13
    mul   R14    <-- R14, R14
    sub   R15    <-- R13, R14
    st    R4(0)  <-- R17
    mul   R16    <-- R16, R16
    mul   R17    <-- R17, R17
    ld    R2     <-- R10(0)
reconv:
    add   R8     <-- R2, R5
    mul   R8     <-- R8, R8
```

Give three reasons as to why it could be difficult for the compiler to perform effective static scheduling of instructions in the above code example. Refer to specific instructions when necessary for clarity.

Reason 1:

Load/store addresses are unknown

Reason 2:

Branch direction is unknown to the compiler

Reason 3:

Load and store hit status is unknown to the compiler

(b) **Load Store Handling [36 points]**

A modern out-of-order execution processor has a store buffer (also called a store queue) as discussed in class. Remember that this buffer is implemented as a searchable structure, i.e., content addressable memory. Such a processor also has a load buffer (also called a load queue), which contains all information about decoded but not yet retired load instructions, in program order. Also as discussed in class, most modern processors intelligently schedule load instructions in the presence of "unknown address" stores. The process of handling load/store dependencies and scheduling is called "memory disambiguation". For this question, assume a processor that aggressively schedules load instructions even in the presence of older store instructions with unknown addresses. Within this context, answer the following questions.

1. What is the purpose of the store buffer?

1) to enable in-order commit of stores 2) data forwarding for different loads

2. Exactly when is the store buffer searched (if at all)?

When the load instruction is scheduled.

3. What is the purpose of the load buffer?

To verify whether a load was correctly scheduled in the presense of previously unknown address store

4. Exactly when is the load buffer searched (if at all)?

When a store instruction executes

5. You are given the following state of the store buffer and the load buffer at a given instant of time. Assume the ISA allows unaligned loads and stores, registers are 4 bytes, and byte and double-byte loads are sign-extended to fill the 4-byte registers. Assume no pipeline flushes during the execution of the code that lead to this state. Starting addresses and sizes are denoted in bytes; memory is little-endian. Data values are specified in hexadecimal format.

Store buffer

Inst num.	Valid	Starting Address	Size	Data
3	Yes	10	1	x00000005
5	Yes	36	4	x00000006
6	Yes	64	2	x00000007
7	Yes	11	1	x00000004
9	Yes	44	4	x00000005
10	Yes	72	2	x00000006
11	Yes	12	1	x00000006
13	Yes	52	4	x00000008
14	Yes	80	2	x00000007
15	Yes	13	1	x00000006
32	Yes	Unknown	4	x00000009

Load buffer

Inst num.	Valid	Starting Address	Size	Destination
8	Yes	10	2	R1
12	Yes	11	2	R2
22	Yes	12	4	R3
28	Yes	10	4	R4
29	Yes	46	2	R5
30	Yes	13	2	R6
36	Yes	10	4	R7
48	Yes	54	2	R8

What is the value (in hexadecimal notation) loaded by each load into its respective destination register after the load is committed to the architectural state?

R1: 0x00000405

R2: 0x00000604

R3: 0xXXXX0606

R4: 0x06060405

R5: 0x00000000

R6: 0xXXXXXX06

R7: Unknown

R8: Unknown

(c) **Tomasulo's Algorithm [15 points]**

Remember that Tomasulo's algorithm requires tag broadcast and comparison to enable wake-up of dependent instructions. In this question, we will calculate the number of tag comparators and size of tag storage required to implement Tomasulo's algorithm, as described in Lecture 14, in a machine that has the following properties:

- 8 functional units where each functional unit has a dedicated separate tag and data broadcast bus
- 32 64-bit architectural registers
- 16 reservation station entries per functional unit
- Each reservation station entry can have two source registers

Answer the following questions. Show your work for credit.

a) What is the number of tag comparators per reservation station entry?

8*2

b) What is the total number of tag comparators in the entire machine?

16*8*2*8+8*32

c) What is the (minimum possible) size of the tag?

$log(16 * 8) = 7$

d) What is the (minimum possible) size of the register alias table (or, frontend register file) in bits?

$72 * 32$ (64 bits for data, 7 bits for the tag, 1 valid bit)

e) What is the total (minimum possible) size of the tag storage in the entire machine in bits?

$7 * 32 + 7 * 16 * 8 * 2$

(d) **Branch Prediction [4 points]**

In class, we discussed the heterogeneous element processor (HEP), which used fine-grained multithreading. Assuming you can change only the branch predictor for this design, what branch predictor would you choose for HEP?

No. No branch prediction necessary on an FGMT processor.

(e) **Philosophy [16 points]**

In class, we have discussed many times the tradeoff between programmer versus microarchitect. Among the following sets of two concepts, circle the one that makes programmer's life harder **and at the same time** microarchitect's (or hardware designer's) life easier. Interpret "programmer" broadly as we did in class, e.g., a compiler writer is a programmer, too.

- Reduced instruction set versus complex instruction set
- VLIW versus superscalar execution
- Virtual memory versus overlays (i.e., no virtual memory)
- Multi-core processors versus single-core processors
- More addressing modes versus less addressing modes
- Dataflow execution model versus von Neumann execution model
  Unclear
- Unaligned access support versus alignment requirements in hardware
- Warp-based SIMD execution versus traditional lock-step SIMD execution

(f) **Virtual Memory [6 points]**

What is the main purpose of a translation lookaside buffer (TLB)?

To accelerate page translation.

Draw a TLB entry below and identify its fields.

Valid bit	VPN	PPN	Access control bits

(g) **GPUs [4 points]**

In a GPU, the unit of fetch and execution is at the granularity of a warp. A warp is a collection of scalar threads that are at the same instruction pointer .

What is the key benefit of grouping threads together in a warp in this manner as opposed to treating each thread as a completely separate entity?

Amortize the cost of fetch/decode of instructions across threads

2. Tomasulo's Dilemma [35 points]

Suppose an out-of-order dispatch machine implementing Tomasulo's algorithm with the following characteristics:

- The processor is fully pipelined with four stages: Fetch, decode, execute, and writeback
- The processor implements `ADD` and `MUL` instructions only.
- For all instructions, fetch takes 1 cycle, decode takes 1 cycle, and writeback takes 2 cycles.
- For `ADD` instructions, execute takes 1 cycle. For `MUL` instructions, execute takes 3 cycles.
- There are enough functional units to have 3 of each instructions executing at the same time.
- The machine implements precise exceptions using a reorder buffer and a future file.
- A reservation station entry and a reorder buffer entry are allocated for an instruction at the end of its decode stage.
- An instruction broadcasts its results at the end of its first writeback cycle. This updates the values in the reservation stations, future file, and reorder buffer.
- Writebacks of different instructions can happen concurrently.

Suppose the pipeline is initially empty and the machine fetches exactly 5 instructions. 7 cycles after fetching the first instruction, the reservation stations and future file are as follows:

`ADD` **Reservation Station**

Tag	V	Tag	Data	V	Tag	Data
A	1	–	3	1	–	2
B	1	–	9	1	–	3
C	1	–	5	1	–	9

`MUL` **Reservation Station**

Tag	V	Tag	Data	V	Tag	Data
X	1	–	2	1	A	5
Y	1	B	12	1	–	2
Z	0	–	–	0	–	–

Future File

	V	Tag	Data
R0	1	–	5
R1	1	–	2
R2	1	–	3
R3	0	C	12
R4	0	Y	1
R5	1	–	9
R6	1	–	0
R7	1	–	21

In the reservation stations, the youngest instruction is at the bottom and some instructions might have already completed.

(a) What are the 5 instructions? Please write them in program order and show all work for full credit.

```
ADD r0 <- r2, r1
MUL r3 <- r1, r0
ADD r3 <- r5, r2
MUL r4 <- r3, r1
ADD r3 <- r0, r5
```

```
       1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10| 11| 12| 13| 14
ADD    F | D | E | W | W
MUL        F | D | E | E | E | W | W
ADD            F | D | E | W | W
MUL                F | D | E | E | E | W | W
ADD                    F | D | E | W | W
```

Now suppose this machine always raises a floating point exception after the last execute stage of `MUL` instructions (the engineer who designed the multiplication unit did not verify his design).

(b) Please fill out the reorder buffer right after the first floating point exception is raised. Oldest instructions should be on top and youngest instructions at the bottom. After an instruction is retired, its reorder buffer entry should be marked as invalid. Use `?` to indicate any unknown information at that moment.

Reorder Buffer

Valid	Tag	Opcode	Rd	Dest. Value	Dest. Value Ready	Exception?

Reorder Buffer

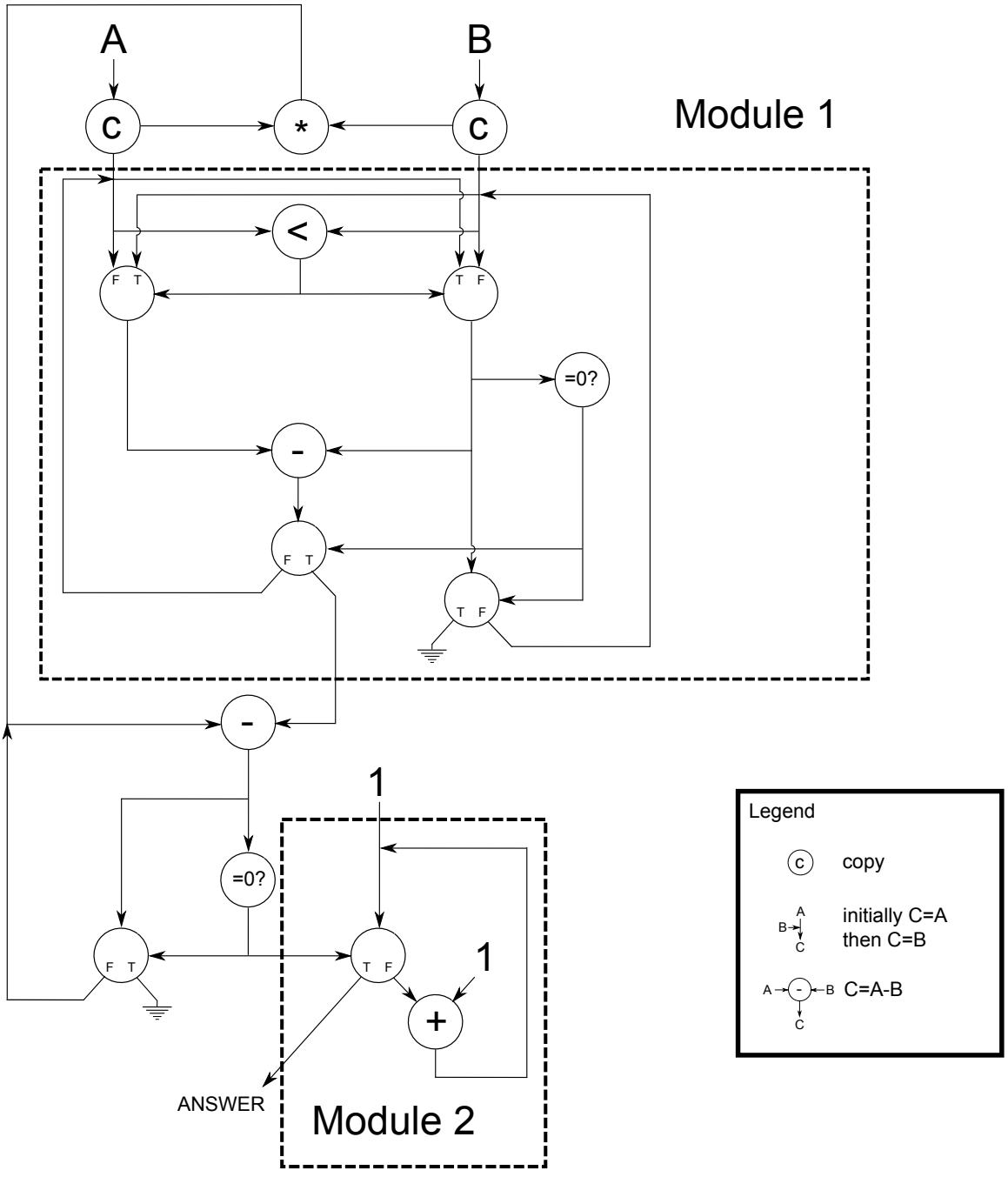
Valid	Tag	Opcode	Rd	Dest. Value	Dest. Value Ready	Exception?
0	A	ADD	r0	5	1	0
1	X	MUL	r3	?	0	1
1	B	ADD	r3	12	1	0
1	Y	MUL	r4	?	0	?
1	C	ADD	r3	?	0	?
0						
0						

3. Dataflow [25 points]

Consider the dataflow graph on the following page and answer the following questions. Please restrict your answer to 10 words or less.

What does the whole dataflow graph do (10 words or less)? (Hint: Identify what Module 1 and Module 2 perform.)

The dataflow graph deadlocks unless the greatest common divisor (GCD) of A and B is the same as the least common multiple (LCM) of A and B. If $(GCD(A, B) == LCM(A, B))$ then ANSWER = 1
If you assume that A and B are fed as inputs continuously, the dataflow graph finds the least common multiple (LCM) of A and B.


A
B
c
*
c
Module 1
<
F T
T F
=0?
-
F T
T F
-
1
=0?
F T
T F
1
+
ANSWER
Module 2
Legend
c
copy
A
B
C
initially C=A
then C=B
A
-
B
C
C=A-B

4. Mystery Instruction [40 points]

A pesky engineer implemented a mystery instruction on the LC-3b. It is your job to determine what the instruction does. The mystery instruction is encoded as:

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

The instruction is only defined if the value of SR1 is greater than zero.

The modifications we make to the LC-3b datapath and the microsequencer are highlighted in the attached figures (see the last four pages). We also provide the original LC-3b state diagram, in case you need it. (As a reminder, the selection logic for SR2MUX is determined internally based on the instruction.)

The additional control signals are

GateTEMP1/1: NO, YES

GateTEMP2/1: NO, YES

LD.TEMP1/1: NO, LOAD

LD.TEMP2/1: NO, LOAD

COND/3:
$COND_{000}$;Unconditional
$COND_{001}$;Memory Ready
$COND_{010}$;Branch
$COND_{011}$;Addressing mode
$COND_{100}$;Mystery 1

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

State	Cond	J	Asserted Signals
001010 (10)	$COND_{000}$	001011	ALUK = PASSA, GateALU, LD.TEMP2, SR1MUX = SR1 (IR[8:6])
001011 (11)	$COND_{000}$	110001	LD.MAR, SR1MUX = BaseR (IR[11:9]), ADDR1MUX = BaseR, ADDR2MUX = 0, MARMUX = ADDER, GateMARMUX
110001 (49)	$COND_{001}$	110001	LD.MDR, MIO.EN, DATA.SIZE=WORD
110011 (51)	$COND_{000}$	100100	GateMDR, LD.TEMP1, DATA.SIZE=WORD
100100 (36)	$COND_{000}$	100101	GateTEMP1, LD.MDR, DATA.SIZE=WORD
100101 (37)	$COND_{001}$	100101	R.W=WRITE, DATA.SIZE=WORD
100111 (39)	$COND_{000}$	101000	GateMARMUX, MARMUX = ADDER, ADDR1MUX = BaseR (IR[11:9]), SR1MUX = BaseR (IR[11:9]), ADDR2MUX = SEXT[IR[5:0]], DRMUX = BaseR (IR[11:9]), LD.REG
101000 (40)	$COND_{100}$	001011	GateTEMP2, LD.TEMP2

Initials:

Describe what this instruction does.

This instruction does a vector increment.

5. GPUs and SIMD [35 points]

We define the *SIMD utilization* of a program run on a GPU as the fraction of SIMD lanes that are kept busy with *active threads* during the run of a program.

The following code segment is run on a GPU. Each thread executes **a single iteration** of the shown loop. Assume that the data values of the arrays A, B, C, and D are already in vector registers so there are no loads and stores in this program. (Hint: Notice that there are 5 instructions in each thread.) A warp in the GPU consists of 64 threads, there are 64 SIMD lanes in the GPU. Assume that each instruction takes the same amount of time to execute.

```
for (i = 0; i < 32768; i++) {
    if (A[i] % 2 == 0) {     // Instruction 1
       A[i] = A[i] * C[i];   // Instruction 2
       B[i] = A[i] + B[i];   // Instruction 3
       C[i] = B[i] + 1;      // Instruction 4
       D[i] = C[i] * B[i];   // Instruction 5
    }
}
```

(a) How many warps does it take to execute this program?

Warps = (Number of threads) / (Number of threads per warp)
Number of threads = 2^{15} (i.e. one thread per loop iteration)
Number of threads per warp = $64 = 2^6$
Warps = $2^{15}/2^6 = 2^9$

(b) Assume arrays A and C have a repetitive pattern which have 32 ones followed by 96 zeros repetitively and arrays B and D have a different repetitive pattern which have 64 zeros followed by 64 ones. What is the SIMD utilization of this program?

A	1	1	...29 1s...	1	0	0	...93 0s...	0	...32 1s...	...96 0s...
B	0	0	...61 0s...	0	1	1	...61 1s...	1	...64 0s...	...64 1s...
C	1	1	...29 1s...	1	0	0	...93 0s...	0	...32 1s...	...96 0s...
D	0	0	...61 0s...	0	1	1	...61 1s...	1	...64 0s...	...64 1s...

((64*5-32*4+64*5)/(64*5+64*5))*100% = 80%

(c) Is it possible for this program to yield a SIMD utilization of 100% (circle one)?

YES NO

If YES, what should be true about arrays A, B, C, and D for the SIMD utilization to be 100%? Be precise and show your work. If NO, explain why not.

Yes. If all of A's elements are even, or if all are odd.

(d) Is it possible for this program to yield a SIMD utilization of 25% (circle one)?

YES NO

If YES, what should be true about arrays A, B, C, and D for the SIMD utilization to be 25%? Be precise.

Yes. If 4 out of every 64 elements in array A are even.

(e) Is it possible for this program to yield a SIMD utilization of 20% (circle one)?

YES NO

If YES, what should be true about arrays A, B, C, and D for the SIMD utilization to be 20%? Be precise.

No. The minimum is where 1/64 elements in array A are even. Which yields a 21.5% usage.

6. Branch Prediction [45 points]

Assume the following piece of code that iterates through two large arrays, j and k, each populated with completely (i.e., truly) random positive integers. The code has five branches (labeled `B1`, `B2`, `B3`, `B4`, and `B5`). When we say that a branch is *taken*, we mean that the code *inside* the curly brackets is executed. Assume the code is run to completion without any errors (there are no exceptions). For the following questions, assume that this is the only block of code that will ever be run on the machines and the loop condition branch is resolved first in the iteration.

```
for (int i = 0; i < 1000; i++) {   /* B1 */
                                   /* TAKEN PATH for B1 */
    if (i % 2 == 0) {              /* B2 */
        j[i] = k[i] * i;           /* TAKEN PATH for B2 */
    }
    if (i < 250) {                 /* B3 */
        j[i] = k[i] - i;           /* TAKEN PATH for B3 */
    }
    if (i < 500) {                 /* B4 */
       j[i] = k[i] + i;            /* TAKEN PATH for B4 */
    }
    if (i >= 500) {                /* B5 */
       j[i] = k[i] / i;            /* TAKEN PATH for B5 */
    }
}
```

You are given three machines whose components are identical in every way, except for their branch predictors.

- Machine A uses an always-taken branch predictor.
- Machine B uses a global two bit saturating counter branch predictor shared by all branches which starts at Weakly Taken (2'b10).
- Machine C uses local two bit saturating counters as branch predictor all of which start at Weakly Not Taken (2'b01).

The saturating counter values are as follows:
2'b00 - Strongly Not Taken
2'b01 - Weakly Not Taken
2'b10 - Weakly Taken
2'b11 - Strongly Taken

(a) What is the misprediction rate when the above piece of code runs on Machine A?

$45.01\% = \frac{2251}{5001}$.

`B1` will generate 1 misprediction out of 1001 executions. `B2` will generate 500 mispredictions out of 1000 iterations, `B3` will generate 750 mispredictions out of 1000 iterations, and both `B4` and `B5` will generate 500 mispredictions out of 1000 iterations.

(b) What is the misprediction rate of the above piece of code on Machine B?

59.97%.

From (0-249): 375 mispredictions for 1250 branches.
From (250-499): 874 mispredictions for 1250 branches.
From (500-1000): 1750 mispredictions for 2501 branches.

(c) What is the misprediction rate of the above piece of code given Machine C?

20.20% = $\frac{1010}{5001}$.

You can split this up by branch.
`B1` will mispredict at i = 0, and i = 1000 (2 mispredictions out of 1001).
`B2` will mispredict every time since it oscillates between WNT and WT (1000 mispredictions out of 1000).
`B3` will mispredict at i = 0, i = 250, and i = 251 (3 mispredictions out of 1000).
`B4` will mispredict at i = 0, i = 500, and i = 501 (3 mispredictions out of 1000).
`B5` will mispredict at i = 500, and i = 501 (2 mispredictions out of 1000).

Now, the above branch predictors do pretty well, but the TAs think they can design better branch predictors for this code.

(d) Varun suggests a hybrid branch predictor, also called a tournament predictor. Using the branch predictors from Machine B and C respectively, the tournament predictor will use a two bit saturating counter, shared by all branches, as the choice predictor which will begin at 2'b10 (Weakly B). The saturating counter values are as follows:

2'b00 - Strongly C
2'b01 - Weakly C
2'b10 - Weakly B
2'b11 - Strongly B

Every time the two predictors disagree about the direction to take on a branch, the counter will move towards the correct branch predictor. So, if Machine C's predictor is correct and Machine B's is wrong, the choice predictor will move towards using Machine C's predictor i.e., it will decrement, unless it is at 2'b00 already. Would you use this branch predictor over the one with the lowest misprediction rate above? Why or why not?
(For this question only, assume steady state behavior for both predictors.)

No matter at what state the choice predictor starts, the local predictor is never wrong when the global predictor is correct. Thus, eventually this new hybrid branch predictor will begin using just the local predictions, meaning it would never do better than Machine C. For the above code, Machine C has a smaller datapath for similar misprediction penalty, so that is better.

(e) Rachata notices an interesting pattern for Machine C, and believes that he can improve it by simply tweaking the starting conditions of its branch predictor. He proposes a solution that involves the same predictor as Machine C, except the starting value of the counter is Weakly Taken (2'b10). Is this better than Machine C? Calculate the misprediction rate to justify your answer.

Yes. 10.16%.

You can split this up by branch.
`B1` will mispredict at i = 1000 iteration (1 misprediction).
`B2` will mispredict every other time since it oscillates between WT and T (500 mispredictions).
`B3` will mispredict at i = 250, and i = 251 (2 mispredictions).
`B4` will mispredict at i = 500, and i = 501 (2 mispredictions).
`B5` will mispredict at i = 0, i = 500, and i = 501 (3 mispredictions).

(f) Being a dedicated computer architecture student, you suggest using a two-level local branch predictor with a two-bit local history registers per branch and a separate pattern history table per branch, consisting of 2-bit saturating counters for every entry to improve the design. You suggest that both bits of the local history registers are initialized to Taken (2'b11) and the 2-bit saturating counters in the pattern history tables are initialized to Weakly Taken (2'b10). Is this better than Machine A, B and C? Calculate the misprediction rate to justify your answer.

Yes. 0.28%.

You can split this up by branch.
`B1` will mispredict at i = 1000 iteration (1 misprediction).
`B2` will mispredict at i = 1 and i = 3 before the histories settle into the pattern. (2 mispredictions)
`B3` will mispredict at i = 250, i = 251, and i = 252 (3 mispredictions).
`B4` will mispredict at i = 500, i = 501, and i = 502 (3 mispredictions).
`B5` will mispredict at i = 0, 1, 2, 500, and 502 (5 mispredictions).

7. Virtual Memory Deja Vu [35 points]

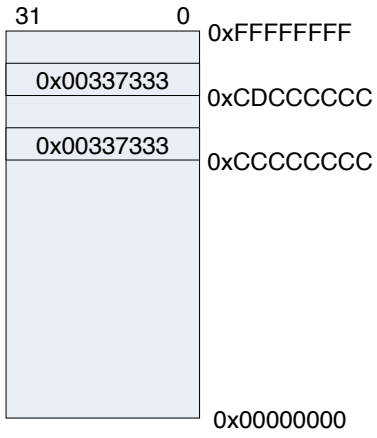
A dedicated computer architecture student (like you) bought a 32-bit processor that implements paging-based virtual memory using a single-level page table. Being a careful person, she also read the manual and learnt that

- A page table entry (PTE) is 4-bytes in size.
- A PTE stores the physical page number in the least-significant bits. Unused bits are zeroed out.
- The page table base address (PTBA) is page-aligned.
- If an instruction attempts to modify a PTE, an exception is raised.

However, the manual did not mention the page size and the PTBA. The dedicated student then wrote the following program to find out the missing information.

```
char *ptr = 0xCCCCCCCC;
*ptr = 0x00337333;
```

The code ran with no exceptions. The following figure shows relevant locations of the physical memory **after** execution.



Using these results, what is the PTBA of this machine? Please show your work for full credit.

Given that the page size is 1024 (from the next part), and either $0xCCCCCCCC$ or $0xCDCCCCCC$ can be the PTE, we can get PTBA as follow:

- If $0xCDCCCCCC$ is the PTE, $0xCDCCCCCC$ is the PA. In this case, there will be an exception.
- if $0xCCCCCCCC$ is the PTE, $0xCDCCCCCC$ is the PA. In this case, there will be no exception.

As a result, $0xCCCCCCCC$ is the PTE.

$PTBA = PTE - (2^{VPNbits} * 4) = 0xCCCCCCCC - 0xCCCCCC = 0xCC000000$

What is the page size (in bytes) of this machine? Write your answer in the form 2^n.

Because the two locations in the diagram are the two relevent locations of the physical memory, we can deduct that $0xCDCCCCCC$ and $0xCCCCCCCC$ are the locations of the PA as well as PTE for VA=$0xCCCCCCCC$.

This means $0x337333$ is a part of the PPN. In this case, we can deduce that the number of bits for the PPN is at least 22 bits.

In addtion, note that the PA contains $0xCCCCCC$ in the least significant bits. This means that the PO consist of $0xC$'s in the least significant bits.

With this information, we can deduce that PPN—PO = PA = $0xCDCCCCCC$ or $0xCCCCCCCC$. The only number of bits for the PPN that satisfies this condition is 10 bits.

So, the page size is $2^{10} = 1024$

8. [Extra credit] Value Prediction [16 points]

In class, we discussed the idea of value prediction as a method to handle data dependences. One method of value prediction for an instruction is "last-time prediction." The idea is to predict the value to be produced by the instruction as the value produced by the same instruction the last time the instruction was executed. If the instruction was never executed before, the predictor predicts the value to be 1. Value prediction accuracy of an instruction refers to the fraction of times the value of an instruction is correctly predicted out of all times the instruction is executed.

Assume the following piece of code, which has four load instructions in each loop iteration, loads to arrays x, y, z, t:

```
// initialize integer variables c, d, e, f to zeros
// initialize integer arrays x, y, z, t

for (i=0; i<1000; i++) {
    c += x[i];
    d += y[i];
    e += z[i];
    f += t[i];
}
```

Assume the following state of arrays before the loop starts executing:

- x consists of all 0's
- y consists of alternating 3's and 6's in consecutive elements
- z consists of random values between 0 and $2^(32) - 1$
- t consists of 0, 1, 2, 3, 4, ..., 999

a) What is the value prediction accuracy of the aforementioned predictor for the four load instructions in the program?

load of x[i]:

99.9%

load of y[i]:

0%

load of z[i]:

Very close to zero

load of t[i]:

0%

b) Can you design a predictor that can achieve higher accuracy for the prediction of x[i]?

[YES] NO

If so, explain your design.

Always predict zero.

c) Can you design a predictor that can achieve higher accuracy for the prediction of y[i]?

[YES] NO

If so, explain your design.

Use a two-bit counter based last value predictor (change the predicted value only if the same value is seen twice in a row).

d) Can you design a predictor that can achieve higher accuracy for the prediction of z[i]?

YES [NO]

If so, explain your design.

Cannot predict random values.

e) Can you design a predictor that can achieve higher accuracy for the prediction of t[i]?

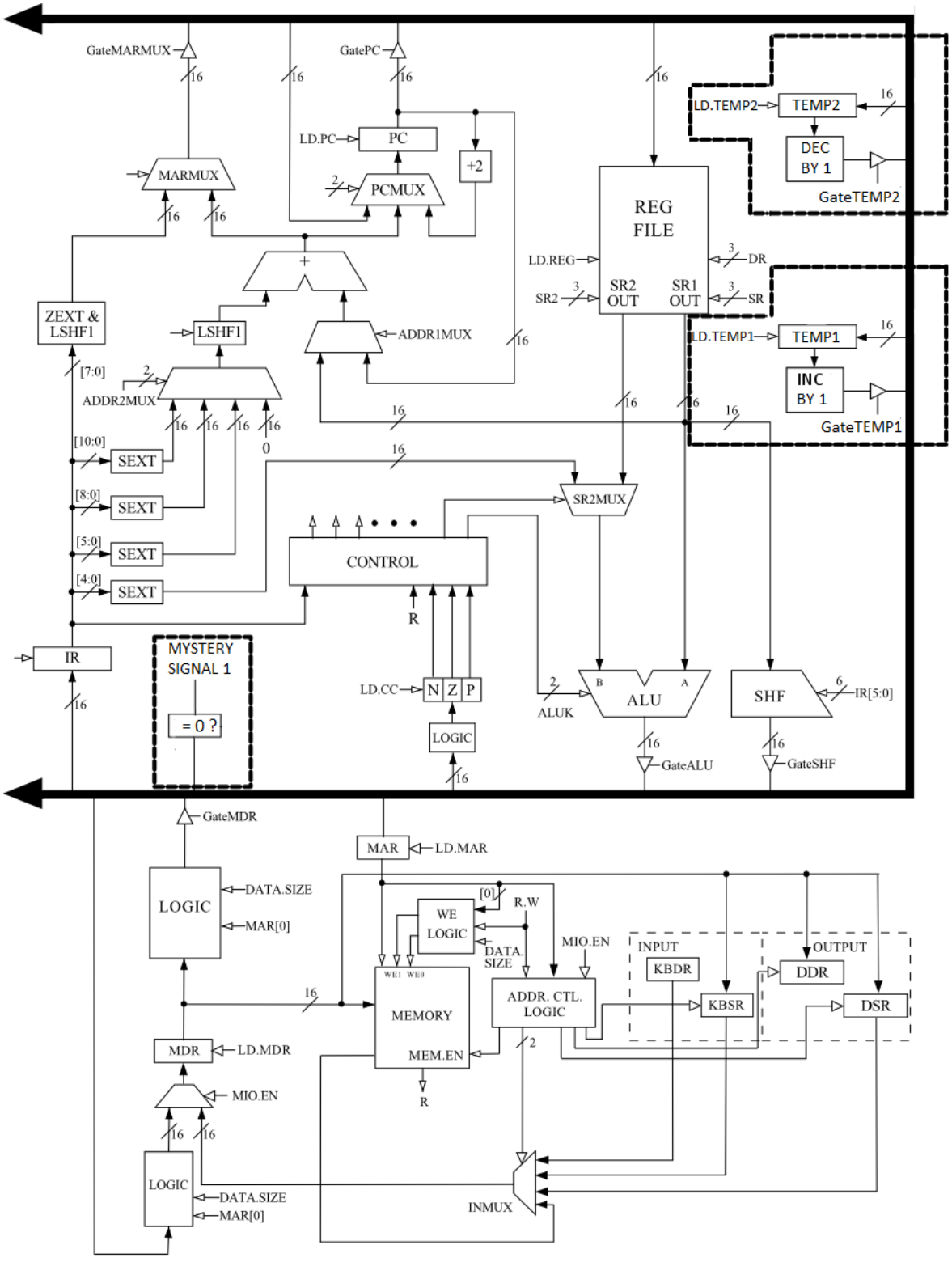
[YES] NO

If so, explain your design.

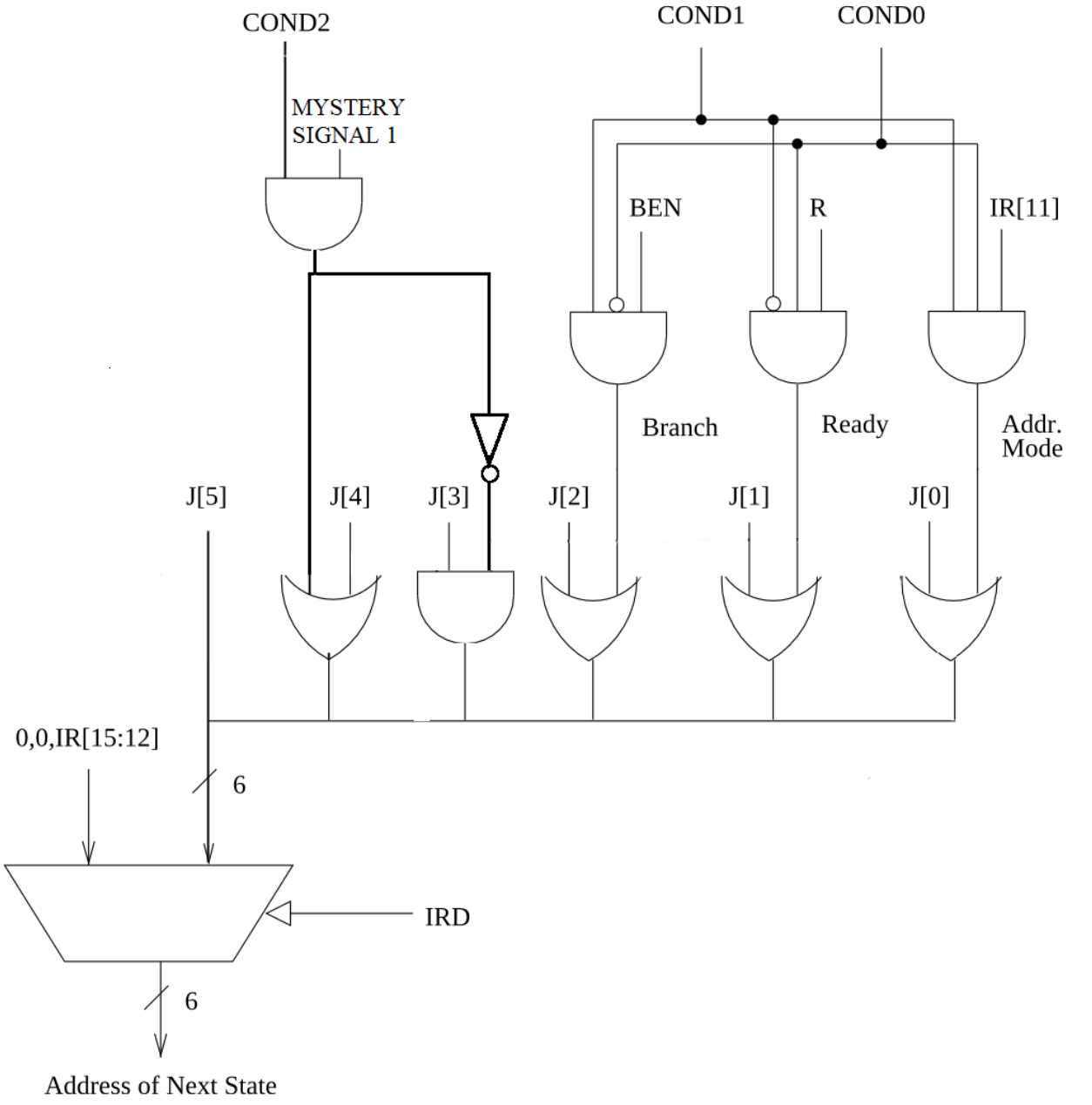
Use a stride predictor (i.e., initially predict zero, then predict one larger than the last predicted value for each instance).

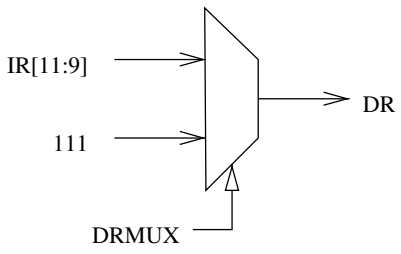
Stratchpad

Initials:

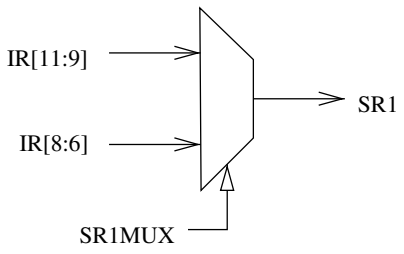

GateMARMUX
GatePC
LD.PC
PC
+2
PCMUX
MARMUX
ZEXT & LSHF1
LSHF1
ADDR1MUX
ADDR2MUX
[7:0]
[10:0]
[8:0]
[5:0]
[4:0]
SEXT
SEXT
SEXT
SEXT
0
REG FILE
LD.REG
SR2
SR2 OUT
SR1 OUT
DR
SR
LD.TEMP2
TEMP2
DEC BY 1
GateTEMP2
LD.TEMP1
TEMP1
INC BY 1
GateTEMP1
SR2MUX
CONTROL
R
IR
MYSTERY SIGNAL 1
= 0 ?
LD.CC
N Z P
LOGIC
ALUK
ALU
SHF
IR[5:0]
GateALU
GateSHF
GateMDR
MAR
LD.MAR
LOGIC
DATA.SIZE
MAR[0]
WE LOGIC
[0]
R.W
DATA. SIZE
MIO.EN
INPUT
KBDR
OUTPUT
DDR
KBSR
DSR
WE1 WE0
MEMORY
MEM.EN
ADDR. CTL. LOGIC
MDR
LD.MDR
MIO.EN
LOGIC
DATA.SIZE
MAR[0]
INMUX

Initials:

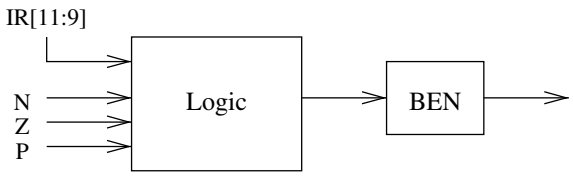

COND2
COND1
COND0
MYSTERY
SIGNAL 1
BEN
R
IR[11]
Branch
Ready
Addr.
Mode
J[5]
J[4]
J[3]
J[2]
J[1]
J[0]
0,0,IR[15:12]
6
IRD
6
Address of Next State



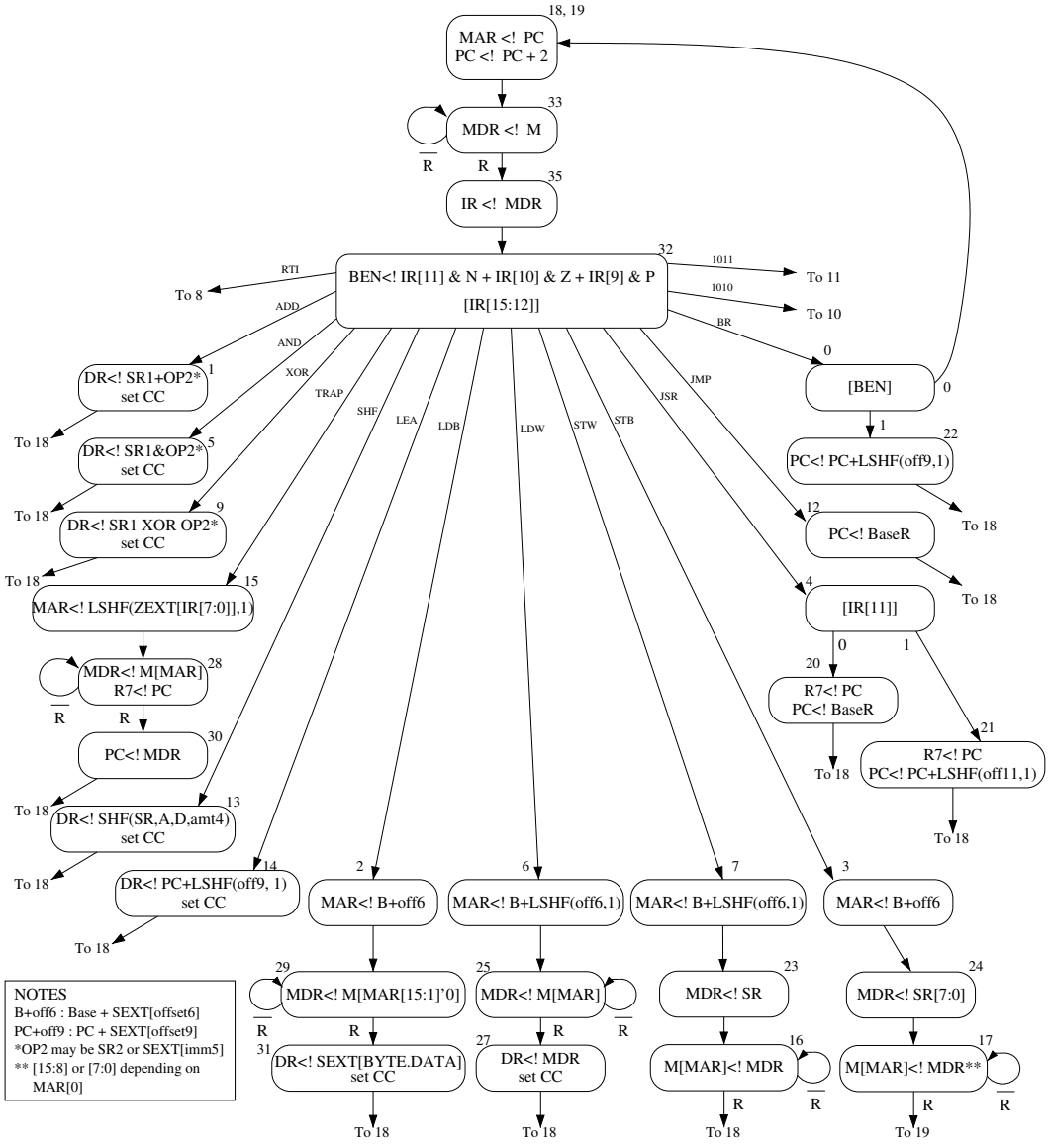
(a)



(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
BR
1011
To 11
1010
To 10
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
29
MDR<! M[MAR[15:1]'0]
R
R
31
DR<! SEXT[BYTE.DATA]
set CC
To 18
6
MAR<! B+LSHF(off6,1)
25
MDR<! M[MAR]
R
R
R
27
DR<! MDR
set CC
To 18
7
MAR<! B+LSHF(off6,1)
23
MDR<! SR
16
M[MAR]<! MDR
R
R
To 18
3
MAR<! B+off6
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]