Design of Digital Circuits (252-0028-00L), Spring 2018

Optional HW 3: Basic Microarchitectural Design

SOLUTIONS

Instructor: Prof. Onur Mutlu
TAs: Juan Gomez Luna, Hasan Hassan, Arash Tavakkol, Minesh Patel, Jeremie Kim, Giray Yaglikci

Assigned: Thursday, Apr 25, 2018

1 Big versus Little Endian Addressing

Consider the 32-bit hexadecimal number 0xcafe2b3a.

1. What is the binary representation of this number in *little endian* format? Please clearly mark the bytes and number them from low (0) to high (3).

3a	2b	fe	ca
0	1	2	3

2. What is the binary representation of this number in *big endian* format? Please clearly mark the bytes and number them from low (0) to high (3).

ca	fe	2b	3a
0	1	2	3

2 The MIPS ISA

2.1 Warmup: Computing a Fibonacci Number

The Fibonacci number F_n is recursively defined as

$$F(n) = F(n-1) + F(n-2),$$

where $F(1) = 1$ and $F(2) = 1$. So, $F(3) = F(2) + F(1) = 1 + 1 = 2$, and so on. Write the MIPS assembly for the `fib(n)` function, which computes the Fibonacci number $F($`n`$)$:

```
int fib(int n)
{
  int a = 0;
  int b = 1;
  int c = a + b;
  while (n > 1) {
    c = a + b;
    a = b;
    b = c;
    n--;
  }
  return c;
}
```

Remember to follow MIPS calling convention and its register usage (just for your reference, you may not need to use all of these registers):

- The argument `n` is passed in register \$4.
- The result (i.e., `c`) should be returned in \$2.
- \$8 to \$15 are caller-saved temporary registers.
- \$16 to \$23 are callee-saved temporary registers.
- \$29 is the stack pointer register.
- \$31 stores the return address.

Note: A summary of the MIPS ISA is provided at the end of this handout.

NOTE: More than one correct solution exists, this is just one potential solution.

```
fib:
addi $sp, $sp, -16 // allocate stack space
sw   $16, 0($sp)   // save r16
add  $16, $4,  $0  // r16 for arg n
sw   $17, 4($sp)   // save r17
add  $17, $0,  $0  // r17 for a, init to 0
sw   $18, 8($sp)   // save r18
addi $18, $0,  1   // r18 for b, init to 1
sw   $31, 12($sp)  // save return address
add  $2,  $17, $18 // c = a + b

branch:
slti $3,  $16, 2   // use r3 as temp
bne  $3,  $0,  done
add  $2,  $17, $18 // c = a + b
add  $17, $18, $0  // a = b
add  $18, $2,  $0  // b = c
addi $16, $16, -1  // n = n - 1
j    branch

done:
lw   $31, 12($sp)  // restore r31
lw   $18, 8($sp)   // restore r18
lw   $17, 4($sp)   // restore r17
lw   $16, 0($sp)   // restore r16
addi $sp, $sp, 16  // restore stack pointer
jr   $31           // return to caller
```

2.2 MIPS Assembly for REP MOVSB

MIPS is a simple ISA. Complex ISAs—such as Intel's x86—often use one instruction to perform the function of many instructions in a simple ISA. Here you will implement the MIPS equivalent for a single Intel x86 instruction, REP MOVSB, which is specified as follows.

The REP MOVSB instruction uses three fixed x86 registers: ECX (count), ESI (source), and EDI (destination). The "repeat" (REP) prefix on the instruction indicates that it will repeat ECX times. Each iteration, it moves one byte from memory at address ESI to memory at address EDI, and then increments both pointers by one. Thus, the instruction copies ECX bytes from address ESI to address EDI.

(a) Write the corresponding assembly code in MIPS ISA that accomplishes the same function as this instruction. You can use any general purpose register. Indicate which MIPS registers you have chosen to correspond to the x86 registers used by REP MOVSB. Try to minimize code size as much as possible.

Assume: \$1 = ECX, \$2 = ESI, \$3 = EDI

```
beq    $1, $0, AfterLoop          // If counter is zero, skip
CopyLoop:
lb     $4, 0($2)                  // Load 1 byte
sb     $4, 0($3)                  // Store 1 byte
addiu  $2, $2, 1                  // Increase source pointer by 1 byte
addiu  $3, $3, 1                  // Increase destination pointer by 1 byte
addiu  $1, $1, -1                 // Decrement counter
bne    $1, $0, CopyLoop           // If not zero, repeat
AfterLoop:
Following instructions
```

(b) What is the size of the MIPS assembly code you wrote in (a), in bytes? How does it compare to REP MOVSB in x86 (note: REP MOVSB occupies 2 bytes)?

The size of the MIPS assembly code is 4 bytes $\times$ 7 = 28 bytes, as compared to 2 bytes for x86 REP MOVSB.

(c) Assume the contents of the x86 register file are as follows before the execution of the REP MOVSB:

```
EAX: 0xccccaaaa
EBP: 0x00002222
ECX: 0xFEE1DEAD
EDX: 0xfeed4444
ESI: 0xdecaffff
EDI: 0xdeaddeed
EBP: 0xe0000000
ESP: 0xe0000000
```

Now, consider the MIPS assembly code you wrote in (a). How many total instructions will be executed by your code to accomplish the same fuction as the single REP MOVSB in x86 accomplishes for the given register state?

The count (value in ECX) is 0xfee1dead = 4276215469. Therefore, loop body is executed (4276215469 * 6) = 25657292814 times. Total instructions executed = 25657292814 + 1 (beq instruction outside of the loop) = 25657292815.

(d) Assume the contents of the x86 register file are as follows before the execution of the REP MOVSB:

```
EAX: 0xccccaaaa
EBP: 0x00002222
ECX: 0x00000000
EDX: 0xfeed4444
ESI: 0xdecaffff
EDI: 0xdeaddeed
EBP: 0xe0000000
ESP: 0xe0000000
```

Now, answer the same question in (c) for the above register values.

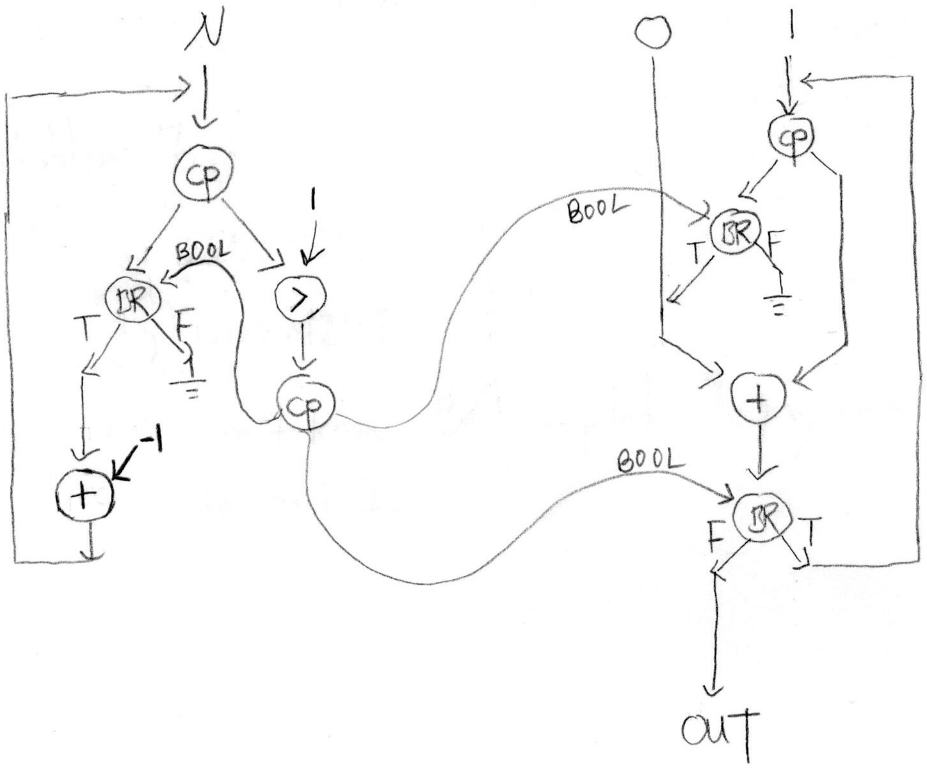
The count (value in ECX) is 0x00000000 = 0. Therefore, loop body is executed 0 times. Total instructions executed = 1 (beq instruction outside of the loop).

3 Data Flow Programs

Draw the data flow graph for the `fib(n)` function from Question 2.1. You may use the following data flow nodes in your graph:

- + (addition)
- > (left operand is greater than right operand)
- `Copy` (copy the value on the input to both outputs)
- `BR` (branch, with the semantics discussed in class, label the True and False outputs)

You can use constant inputs (e.g., 1) that feed into the nodes. Clearly label all the nodes, program inputs, and program outputs. Try to the use fewest number of data flow nodes possible.



4 Microarchitecture vs. ISA

a) Briefly explain the difference between the *microarchitecture* level and the *ISA* level in the transformation hierarchy. What information does the compiler need to know about the microarchitecture of the machine in order to compile a given program correctly?

The ISA level is the interface a machine exposes to the software. The microarchitecture is the actual underlying implementation of the machine. Therefore, the microarchitecture and changes to the microarchitecture are transparent to the compiler/programmer (except in terms of performance), while changes to the ISA affect the compiler/programmer. The compiler does not need to know about the microarchitecture of the machine in order to compile the program correctly

b) Classify the following attributes of a machine as either a property of its microarchitecture or ISA:

Microarchitecture?	**ISA?**	**Attribute**
	✓	The machine does not have a subtract instruction
✓		The ALU of the machine does not have a subtract unit
	✓	The machine does not have condition codes
	✓	A 5-bit immediate can be specified in an ADD instruction
✓		It takes n cycles to execute an ADD instruction
	✓	There are 8 general purpose registers
✓		A 2-to-1 mux feeds one of the inputs to ALU
✓		The register file has one input port and two output ports

5 Performance Metrics

- If a given program runs on a processor with a higher frequency, does it imply that the processor always executes more instructions per second (compared to a processor with a lower frequency)? (Use less than 10 words.)

  No, the lower frequency processor might have much higher IPC (instructions per cycle).
  More detail: A processor with a lower frequency might be able to execute multiple instructions per cycle while a processor with a higher frequency might only execute one instruction per cycle.

- If a processor executes more of a given program's instructions per second, does it imply that the processor always finishes the program faster (compared to a processor that executes fewer instructions per second)? (Use less than 10 words.)

  No, because the former processor may execute many more instructions.
  More detail: The total number of instructions required to execute the full program could be different on different processors.

6 Performance Evaluation

Your job is to evaluate the potential performance of two processors, each implementing a different ISA. The evaluation is based on its performance on a particular benchmark. On the processor implementing ISA A, the best compiled code for this benchmark performs at the rate of 10 IPC. That processor has a 500 MHz clock. On the processor implementing ISA B, the best compiled code for this benchmark performs at the rate of 2 IPC. That processor has a 600 MHz clock.

- What is the performance in Millions of Instructions per Second (MIPS) of the processor implementing ISA A?

ISA A: $10\frac{instructions}{cycle} * 500,000,000\frac{cycle}{second} = 5000$ MIPS

- What is the performance in MIPS of the processor implementing ISA B?

ISA B: $2\frac{instructions}{cycle} * 600,000,000\frac{cycle}{second} = 1200$ MIPS

- Which is the higher performance processor: A B Don't know
  Briefly explain your answer.

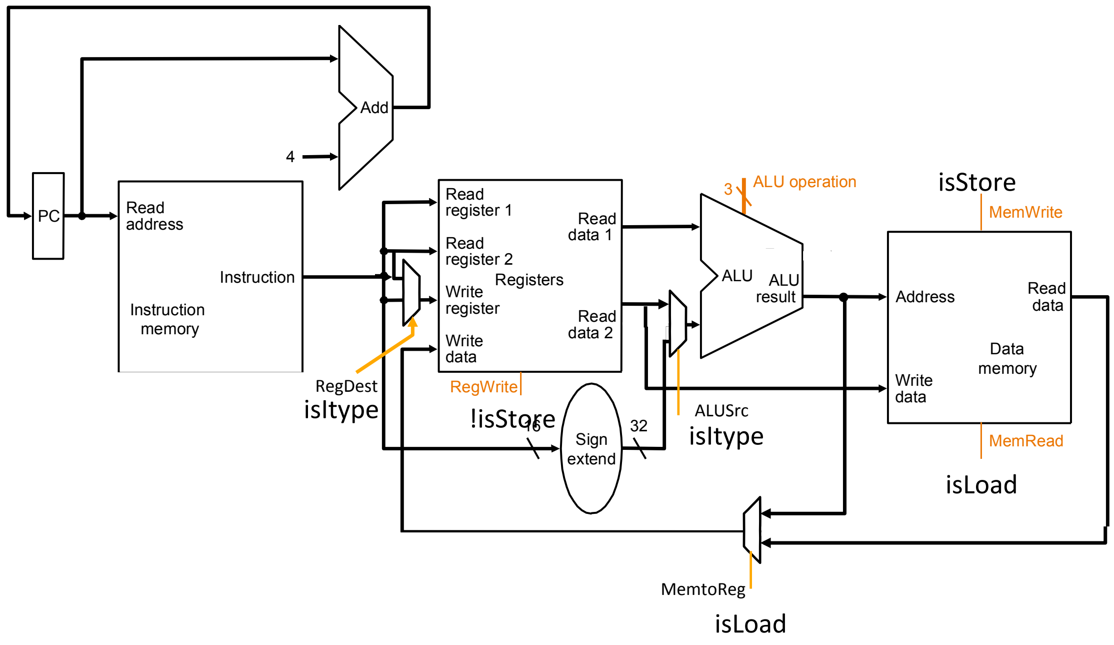
Don't know.
The best compiled code for each processor may have a different number of instructions.

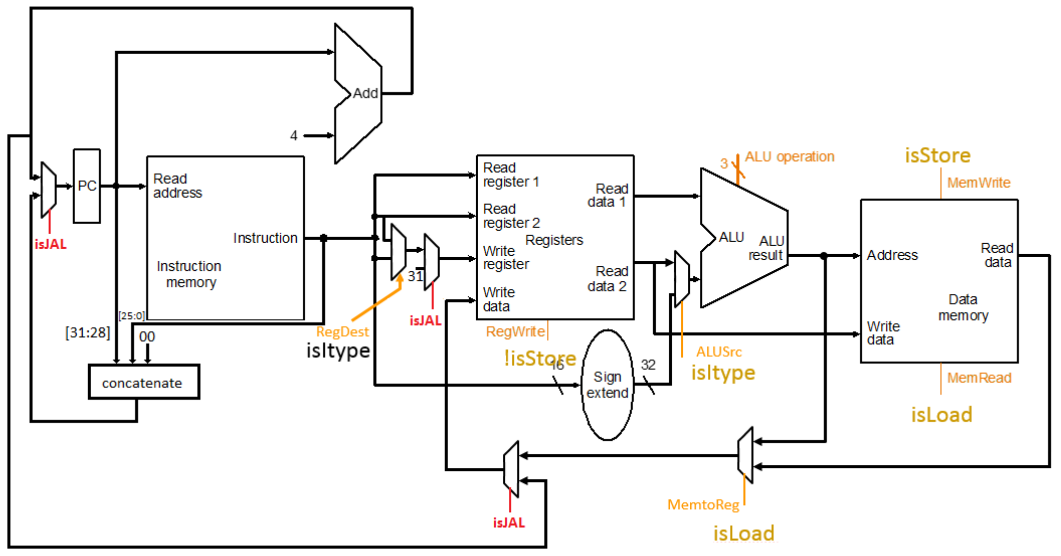
7 Single-Cycle Processor Datapath

In this problem, you will modify the single-cycle datapath we built up in Lecture 11 to support the `JAL` instruction. The datapath that we will start with is provided below. Your job is to implement the necessary data and control signals to support the `JAL` instruction, which we define to have the following semantics:

$$\begin{aligned} \texttt{JAL}:\quad & \texttt{R31} \leftarrow \texttt{PC} + 4 \\ & \texttt{PC} \leftarrow \texttt{PC}_{31\ldots28} \;||\; \texttt{Immediate} \;||\; 0^2 \end{aligned}$$

Add to the datapath on the next page the necessary data and control signals to implement the `JAL` instruction. Draw and label all components and wires very clearly (give control signals meaningful names; if selecting a subset of bits from many, specify exactly which bits are selected; and so on).





8 LC-3b Microprogramming

As we learned in class, the LC-3b is a microprogrammed design, which means that *control signals* throughout the LC-3b datapath (Patt+Patel, Appendix C, Figure 3) determine what the processor does during any given cycle. The control signals are stored in a memory called the *control store*, and they must be carefully chosen to correctly implement the LC-3b state machine (Patt+Patel, Appendix C, Figure 2). The contents of the control store can be easily described using a 2D spreadsheet with rows representing different LC-3b states and columns specifying all 35 control signal bits required for each state.

Unfortunately, a clumsy TA has accidentally erased the contents of the control store! Your task is to recover this data manually (i.e., determine *every* bit in the control store) using your knowledge of the LC-3b architecture.

Fill out the microcode in the *microcode.csv* file handed out with this homework. Enter a 1 or a 0 or an X as appropriate for the microinstructions corresponding to each state. We have filled out state 18 as an example. *Hint: all of the information can be inferred using the information and figures provided in Patt+Patel Appendix C.*

Instruction	state	IRD	Cond	J	LD.MAR	LD.MDR	LD.IR	LD.BEN	LD.REG	LD.CC	LD.PC	GatePC	GateMDR	GateALU	GateMARMUX	GateSHF	PCMUX	DRMUX	SR1MUX	ADDR1MUX	ADDR2MUX	MARMUX	ALUK	MIO.EN	R.W	DATA.SIZE	LSHF1
size		1	2	6	1	1	1	1	1	1	1	1	1	1	1	1	2	1	1	1	2	1	2	1	1	1	1
BR	0	0	2	10010	0	0	0	0	0	0	0	0	0	0	0	0	X	X	X	X	X	X	X	0	X	X	X
ADD	1	0	0	10010	0	0	0	0	1	1	0	0	0	1	0	0	X	0	1	X	X	X	0	0	X	X	X
LDB	2	0	0	11101	1	0	0	0	0	0	0	0	0	0	1	0	X	X	1	1	1	1	X	0	X	X	0
STB	3	0	0	11000	1	0	0	0	0	0	0	0	0	0	1	0	X	X	1	1	1	1	X	0	X	X	0
JSR	4	0	3	10100	0	0	0	0	0	0	0	0	0	0	0	0	X	X	X	X	X	X	X	0	X	X	X
AND	5	0	0	10010	0	0	0	0	1	1	0	0	0	1	0	0	X	0	1	X	X	X	1	0	X	X	X
LDW	6	0	0	11001	1	0	0	0	0	0	0	0	0	0	1	0	X	X	1	1	1	1	X	0	X	X	1
STW	7	0	0	10111	1	0	0	0	0	0	0	0	0	0	1	0	X	X	1	1	1	1	X	0	X	X	1
RTI	8	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
XOR	9	0	0	10010	0	0	0	0	1	1	0	0	0	1	0	0	X	0	1	X	X	X	2	0	X	X	X
RES1	10	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
RES2	11	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
JMP	12	0	0	10010	0	0	0	0	0	0	1	0	0	0	0	0	2	X	1	1	0	X	X	0	X	X	X
JMP	12	0	0	10010	0	0	0	0	0	0	1	0	0	1	0	0	1	X	1	X	X	X	3	0	X	X	X
SHF	13	0	0	10010	0	0	0	0	1	1	0	0	0	0	0	1	X	0	1	X	X	X	X	0	X	X	X
LEA	14	0	0	10010	0	0	0	0	1	0	0	0	0	0	1	0	X	X	X	0	2	1	X	0	X	X	1
TRAP	15	0	0	11100	1	0	0	0	0	0	0	0	0	0	1	0	X	X	X	X	X	0	X	0	X	X	X
STW	16	0	1	10000	0	0	0	0	0	0	0	0	0	0	0	0	X	X	X	X	X	X	X	1	1	1	X
STB	17	0	1	10001	0	0	0	0	0	0	0	0	0	0	0	0	X	X	X	X	X	X	X	1	1	0	X
ALL	18	0	0	100001	1	0	0	0	0	0	1	1	0	0	0	0	0	X	X	X	X	X	X	0	X	X	X
ALL	18	0	0	100001	1	0	0	0	0	0	1	0	0	0	1	0	X	X	X	0	0	1	X	0	X	X	X
ALL	19	0	0	100001	1	0	0	0	0	0	1	1	0	0	0	0	0	X	X	X	X	X	X	0	X	X	X
ALL	19	0	0	100001	1	0	0	0	0	0	1	0	0	0	1	0	X	X	X	0	0	1	X	0	X	X	X
JSR	20	0	0	10010	0	0	0	0	1	0	1	1	0	0	0	0	2	1	1	1	0	X	X	0	X	X	X
JSR	20	0	0	10010	0	0	0	0	1	0	1	1	0	0	0	0	2	1	1	1	1	X	X	0	X	X	X
JSR	21	0	0	10010	0	0	0	0	1	0	1	1	0	0	0	0	2	1	X	0	3	X	X	0	X	X	1
BR	22	0	0	10010	0	0	0	0	0	0	1	0	0	0	0	0	2	X	X	0	2	X	X	0	X	X	1
STW	23	0	0	10000	0	1	0	0	0	0	0	0	0	1	0	0	X	X	0	X	X	X	3	0	X	X	X
STW	23	0	0	10000	0	1	0	0	0	0	0	0	0	0	1	0	X	X	0	1	0	1	X	0	X	X	X
STB	24	0	0	10001	0	1	0	0	0	0	0	0	0	1	0	0	X	X	0	X	X	X	3	0	X	X	X
STB	24	0	0	10001	0	1	0	0	0	0	0	0	0	0	1	0	X	X	0	1	0	1	X	0	X	X	X
LDW	25	0	1	11001	0	1	0	0	0	0	0	0	0	0	0	0	X	X	X	X	X	X	X	1	0	X	X
FREE	26	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
LDW	27	0	0	10010	0	0	0	0	1	1	0	0	1	0	0	0	X	0	X	X	X	X	X	0	X	1	X
TRAP	28	0	1	11100	0	1	0	0	1	0	0	1	0	0	0	0	X	1	X	X	X	X	X	1	0	X	X
LDB	29	0	1	11101	0	1	0	0	0	0	0	0	0	0	0	0	X	X	X	X	X	X	X	1	0	X	X
TRAP	30	0	0	10010	0	0	0	0	0	0	1	0	1	0	0	0	1	X	X	X	X	X	X	0	X	1	X
LDB	31	0	0	10010	0	0	0	0	1	1	0	0	1	0	0	0	X	0	X	X	X	X	X	0	X	0	X
ALL	32	1	X	X	0	0	0	1	0	0	0	0	0	0	0	0	X	X	X	X	X	X	X	0	X	X	X
ALL	33	0	1	100001	0	1	0	0	0	0	0	0	0	0	0	0	X	X	X	X	X	X	X	1	0	X	X
FREE	34	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
ALL	35	0	0	100000	0	0	1	0	0	0	0	0	1	0	0	0	X	X	X	X	X	X	X	0	X	1	X

9 REP MOVSB

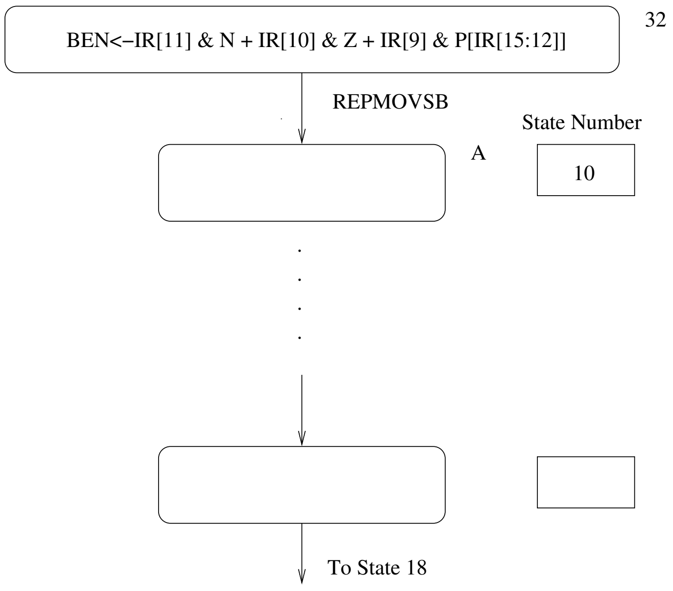
Let's say you are the lead architect of the next flagship processor at Advanced Number Devices (AND). You have decided that you want to use the LC-3b ISA for your next product, but your customers want a smaller semantic gap and marketing is on your case about it. So, you have decided to implement your favorite x86 instruction, REP MOVSB, in LC-3b.

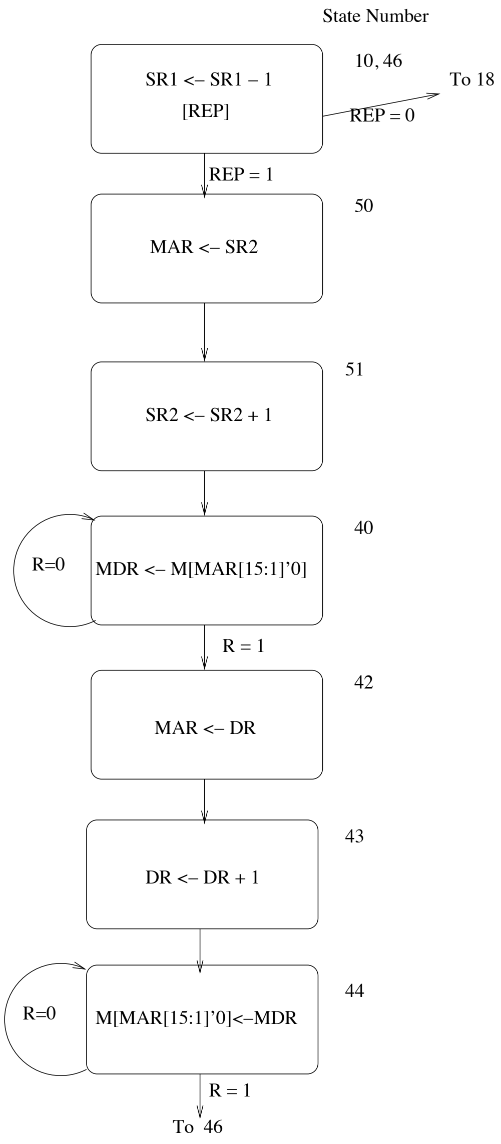
Specifically, you want to implement the following definition for REP MOVSB (in LC-3b parlance): REP-MOVSB SR1, SR2, DR which is encoded in LC-3b machine code as:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1010				DR			SR1			0	0	0	SR2		

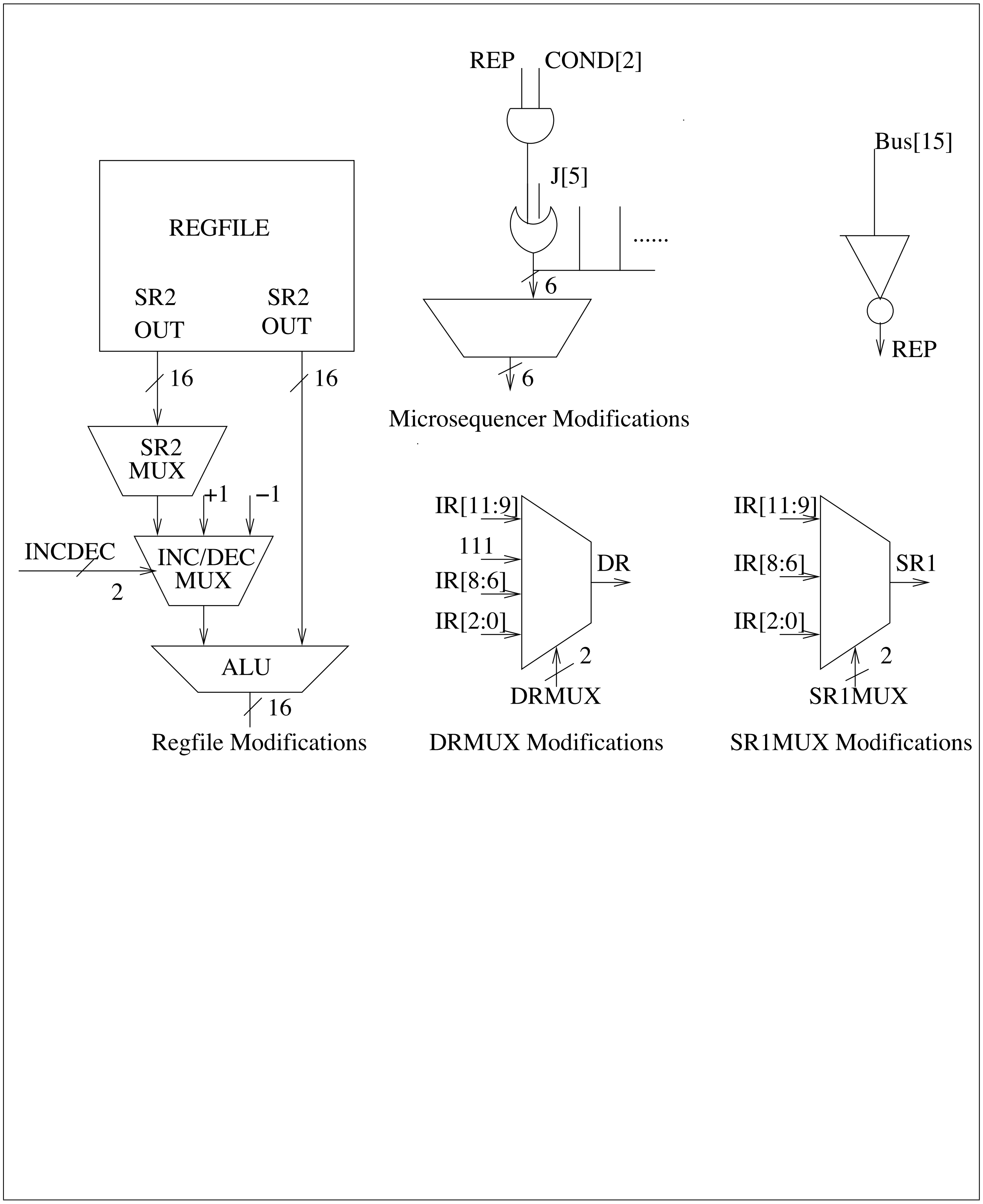
REPMOVSB uses three registers: SR1 (count), SR2 (source), and DR (destination). It moves a byte from memory at address SR2 to memory at address DR, and then increments both pointers by one. This is repeated SR1 times. Thus, the instruction copies SR1 bytes from address SR2 to address DR. Assume that the value in SR1 is greater than or equal to zero.

1. Complete the state diagram shown below, using the notation of the LC-3b state diagram. Describe inside each bubble what happens in each state and assign each state an appropriate state number. Add additional states not present in the original LC-3b design as you see fit.




State Number
SR1 <– SR1 – 1
[REP]
10, 46
To 18
REP = 0
REP = 1
MAR <– SR2
50
SR2 <– SR2 + 1
51
R=0
MDR <– M[MAR[15:1]'0]
40
R = 1
MAR <– DR
42
DR <– DR + 1
43
R=0
M[MAR[15:1]'0]<–MDR
44
R = 1
To 46

2. Add to the LC-3b datapath any additional structures and any additional control signals needed to implement REPMOVSB. Clearly label your additional control signals with descriptive names. Describe what value each control signal would take to control the datapath in a particular way.



3. Describe any changes you need to make to the LC-3b microsequencer. Add any additional logic and control signals you need. Clearly describe the purpose and function of each signal and the values it would take to control the microsequencer in a particular way.

Additional control signals

- INCDEC/2: PASSSR2, +1, -1
- DRMUX/2:
  - IR[11:9] ;destination IR[11:9]
  - R7 ;destination R7
  - IR[8:6] ;destination IR[8:6]
  - IR[2:0] ;destination IR[2:0]
- SR1MUX/2:
  - IR[11:9] ;source IR[11:9]
  - IR[8:6] ;source IR[8:6]
  - IR[2:0] ;source IR[2:0]
- COND/3:
  - COND0: Unconditional
  - COND1: Memory Ready
  - COND2: Branch
  - COND3: Addressing Mode
  - COND4: Repeat

10 Mystery LC-3b Instruction (I)

A pesky engineer has 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				DR			SR1			0	0	0	0	0	0

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

ALUK/3: OR1 (A|0x1), LSHF1 (A<<1), PASSA, PASS0 (Pass value 0), PASS16 (Pass value 16)

COND/4:
COND$_{0000}$;Unconditional
COND$_{0001}$;Memory Ready
COND$_{0010}$;Branch
COND$_{0011}$;Addressing mode
COND$_{0100}$;Mystery 1
COND$_{1000}$;Mystery 2

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

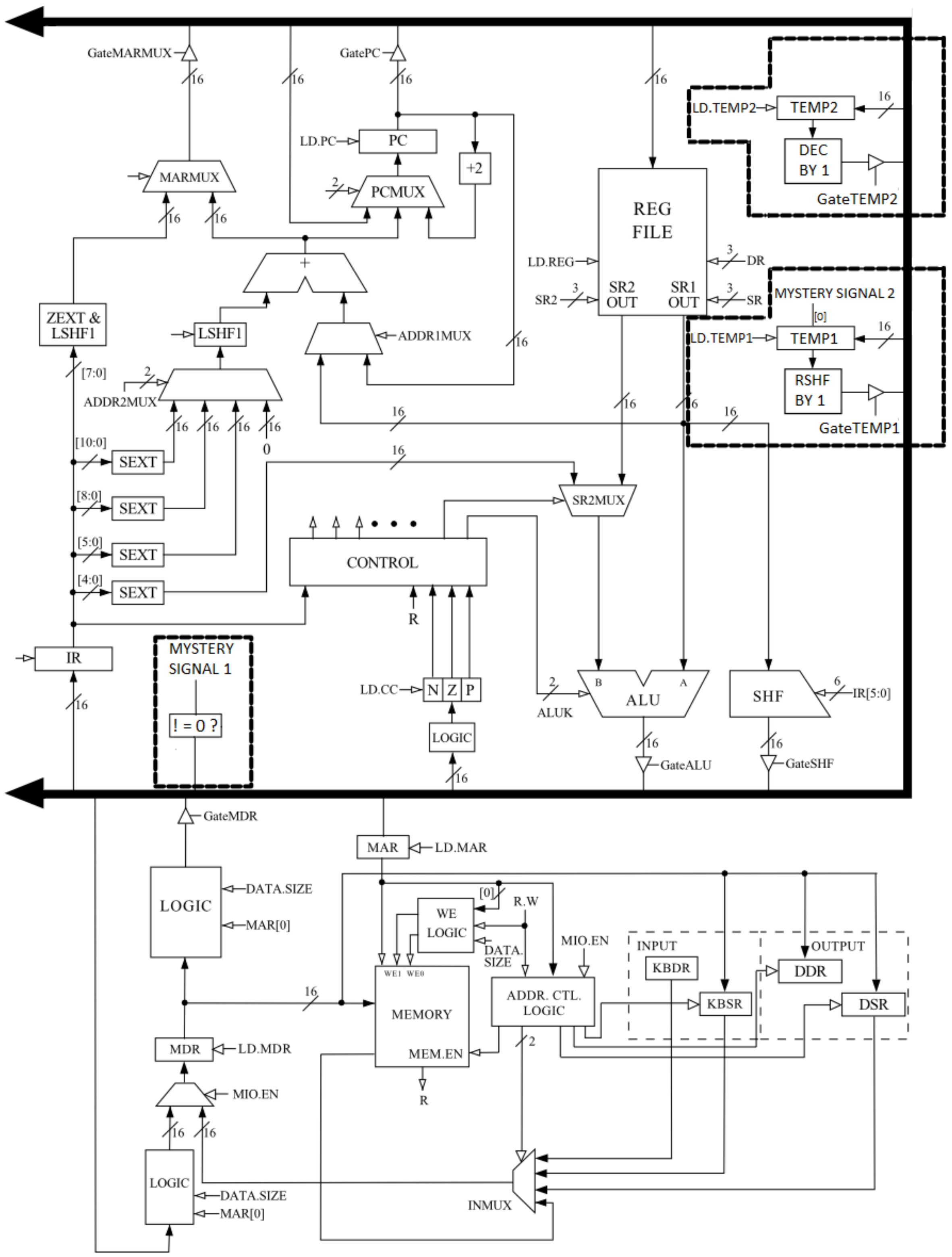
State	Cond	J	Asserted Signals
001010 (10)	COND$_{0000}$	001011	ALUK = PASS0, GateALU, LD.REG, DRMUX = DR (IR[11:9])
001011 (11)	COND$_{0000}$	101000	ALUK = PASSA, GateALU, LD.TEMP1, SR1MUX = SR1 (IR[8:6])
101000 (40)	COND$_{0000}$	110010	ALUK = PASS16, GateALU, LD.TEMP2
110010 (50)	COND$_{1000}$	101101	ALUK = LSHF1, GateALU, LD.REG, SR1MUX = DR, DRMUX = DR (IR[11:9])
111101 (61)	COND$_{0000}$	101101	ALUK = OR1, GateALU, LD.REG, SR1MUX = DR, DRMUX = DR (IR[11:9])
101101 (45)	COND$_{0000}$	111111	GateTEMP1, LD.TEMP1
111111 (63)	COND$_{0100}$	010010	GateTEMP2, LD.TEMP2

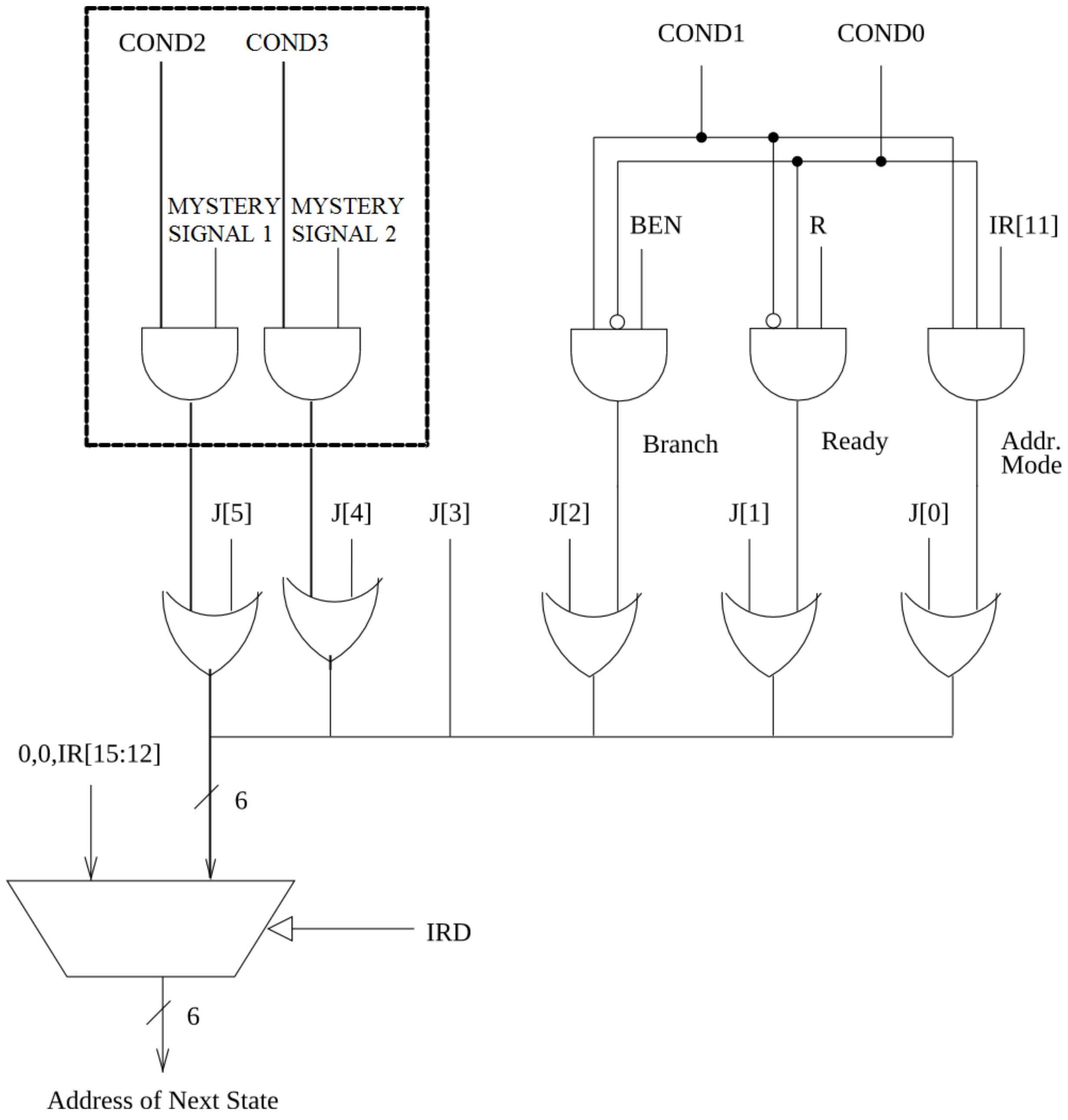
Describe what this instruction does.

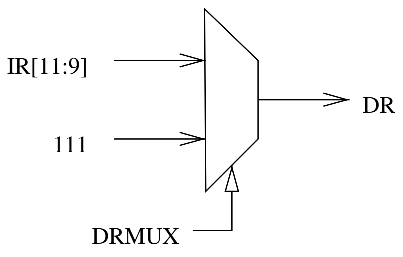
Bit-reverses the value in SR1 and puts it in DR.

Code:

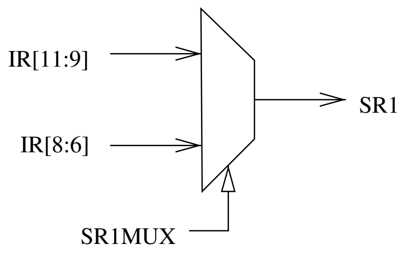
```
State 10: DR ← 0
State 11: TEMP1 ← value(SR1)
State 40: TEMP2 ← 16
State 50: DR = DR << 1
          if (TEMP1[0] == 0)
            goto State 45
          else
            goto State 61
State 61: DR = DR | 0x1
State 45: TEMP1 = TEMP1 >> 1
State 63: DEC TEMP2
          if (TEMP2 == 0)
            goto State 18
          else
            goto State 50
```


GateMARMUX
GatePC
16
LD.PC
PC
+2
PCMUX
MARMUX
REG FILE
LD.TEMP2
TEMP2
DEC BY 1
GateTEMP2
LD.REG
DR
SR2
SR2 OUT
SR1 OUT
SR
MYSTERY SIGNAL 2
[0]
LD.TEMP1
TEMP1
RSHF BY 1
GateTEMP1
ZEXT & LSHF1
LSHF1
ADDR1MUX
ADDR2MUX
[7:0]
[10:0]
[8:0]
[5:0]
[4:0]
SEXT
0
SR2MUX
CONTROL
R
IR
MYSTERY SIGNAL 1
! = 0 ?
LD.CC
N
Z
P
LOGIC
ALU
ALUK
B
A
SHF
IR[5:0]
GateALU
GateSHF
GateMDR
MAR
LD.MAR
DATA.SIZE
MAR[0]
R.W
WE LOGIC
WE1 WE0
MEMORY
MEM.EN
MIO.EN
ADDR. CTL. LOGIC
INPUT
KBDR
KBSR
OUTPUT
DDR
DSR
MDR
LD.MDR
INMUX

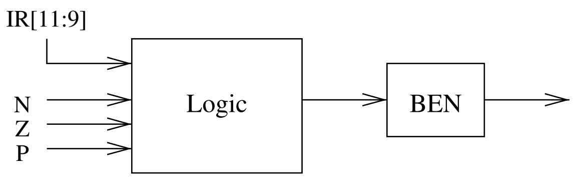

COND2
COND3
MYSTERY SIGNAL 1
MYSTERY SIGNAL 2
COND1
COND0
BEN
R
IR[11]
Branch
Ready
Addr. Mode
J[5]
J[4]
J[3]
J[2]
J[1]
J[0]
0,0,IR[15:12]
6
IRD
6
Address of Next State



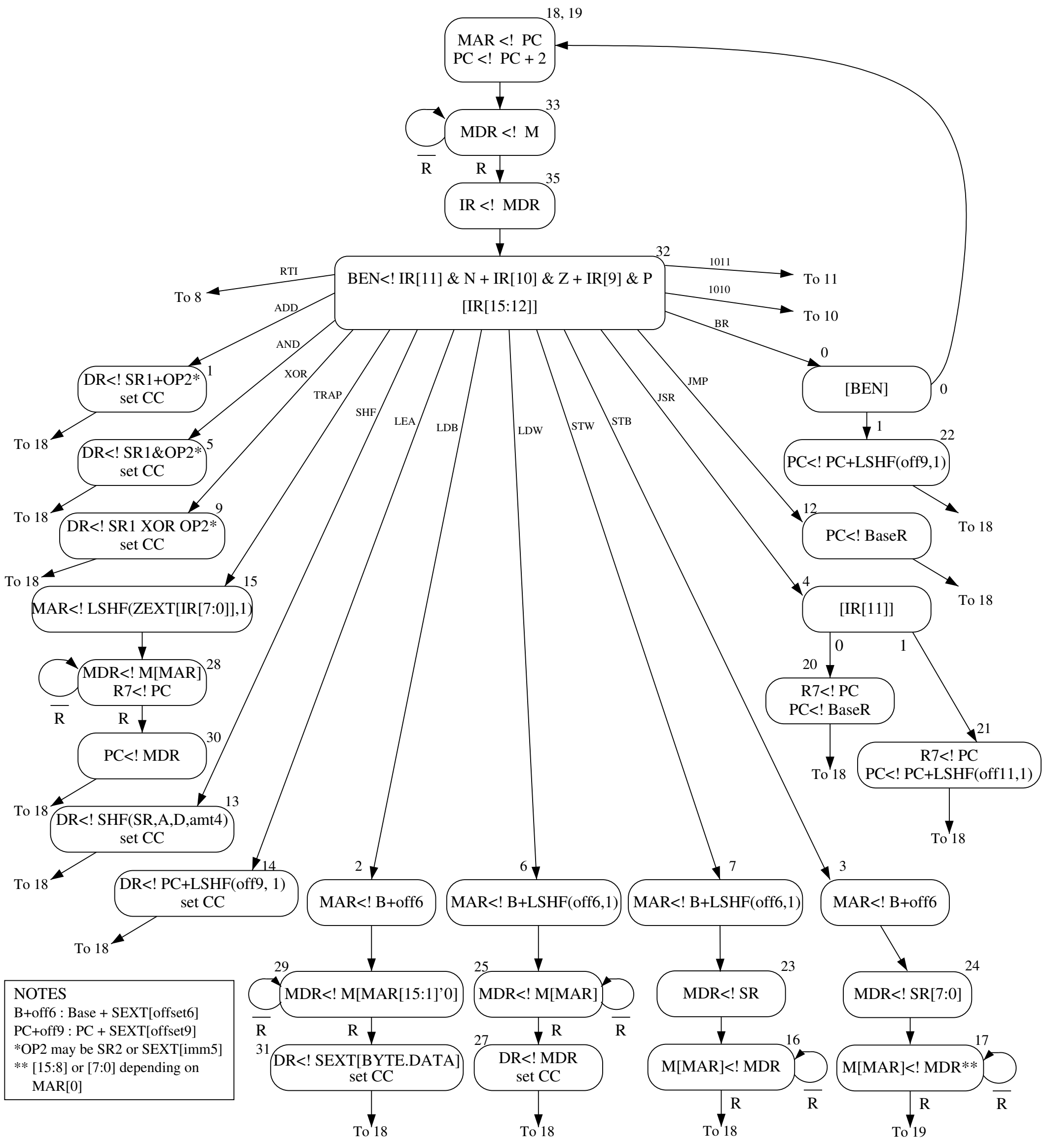
(a)



(b)



(c)



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]

11 Mystery LC-3b Instruction (II)

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)	COND$_0$	101000	ALUK = PASSB, GateALU, LD.REG, DRMUX = DR (IR[11:9])
101000 (40)	COND$_4$	010010	ALUK = PASSA, GateALU, LD.TEMP, SR1MUX = SR1 (IR[8:6])
110010 (50)	COND$_0$	110011	ALUK = DECA, GateALU, LD.REG, SR1MUX = DR, DRMUX = DR (IR[11:9])
110011 (51)	COND$_4$	010010	GateLSHF, LD.TEMP

Describe what this instruction does.

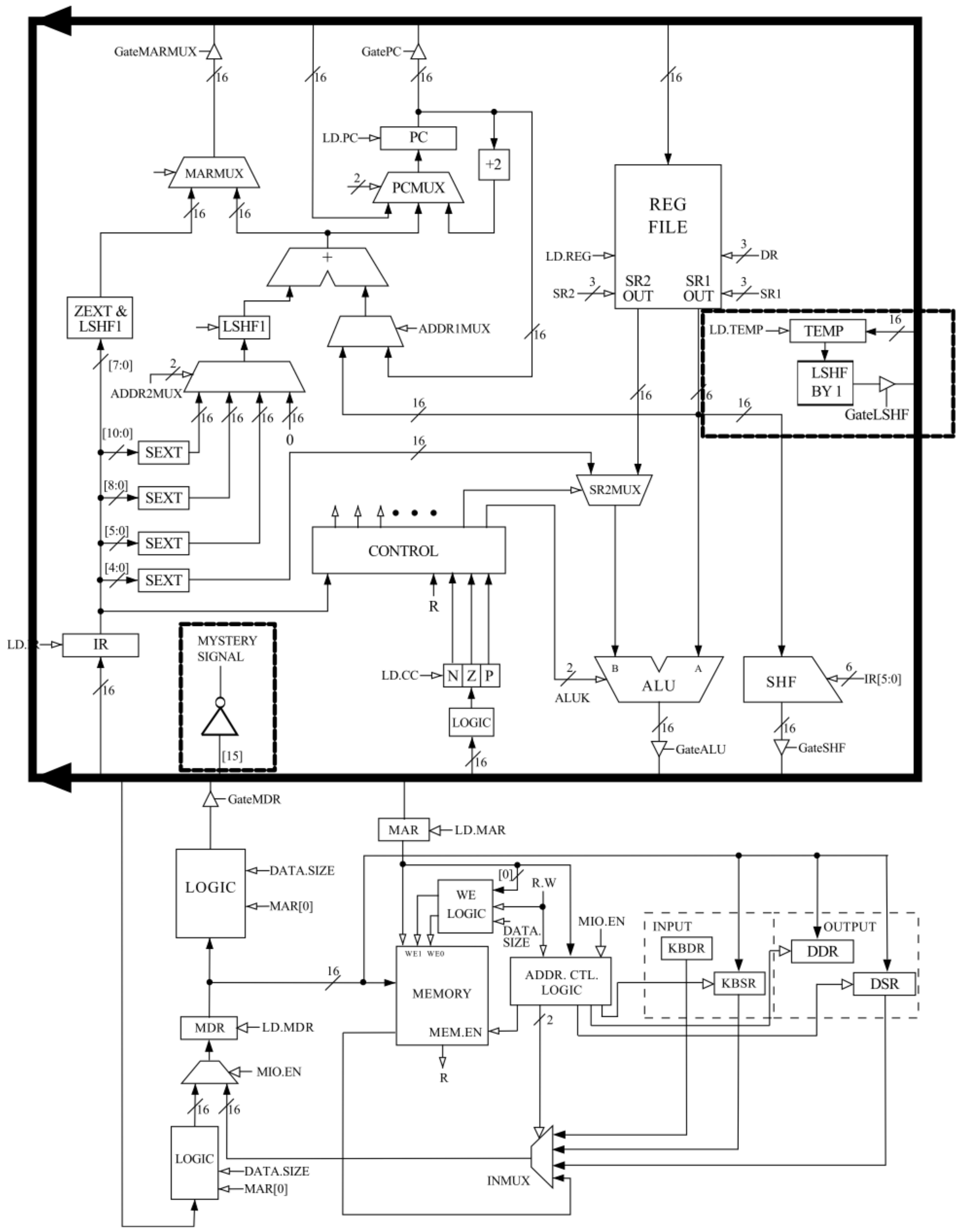
The instruction finds the position of the most significant set bit in SR1 and places this in DR.

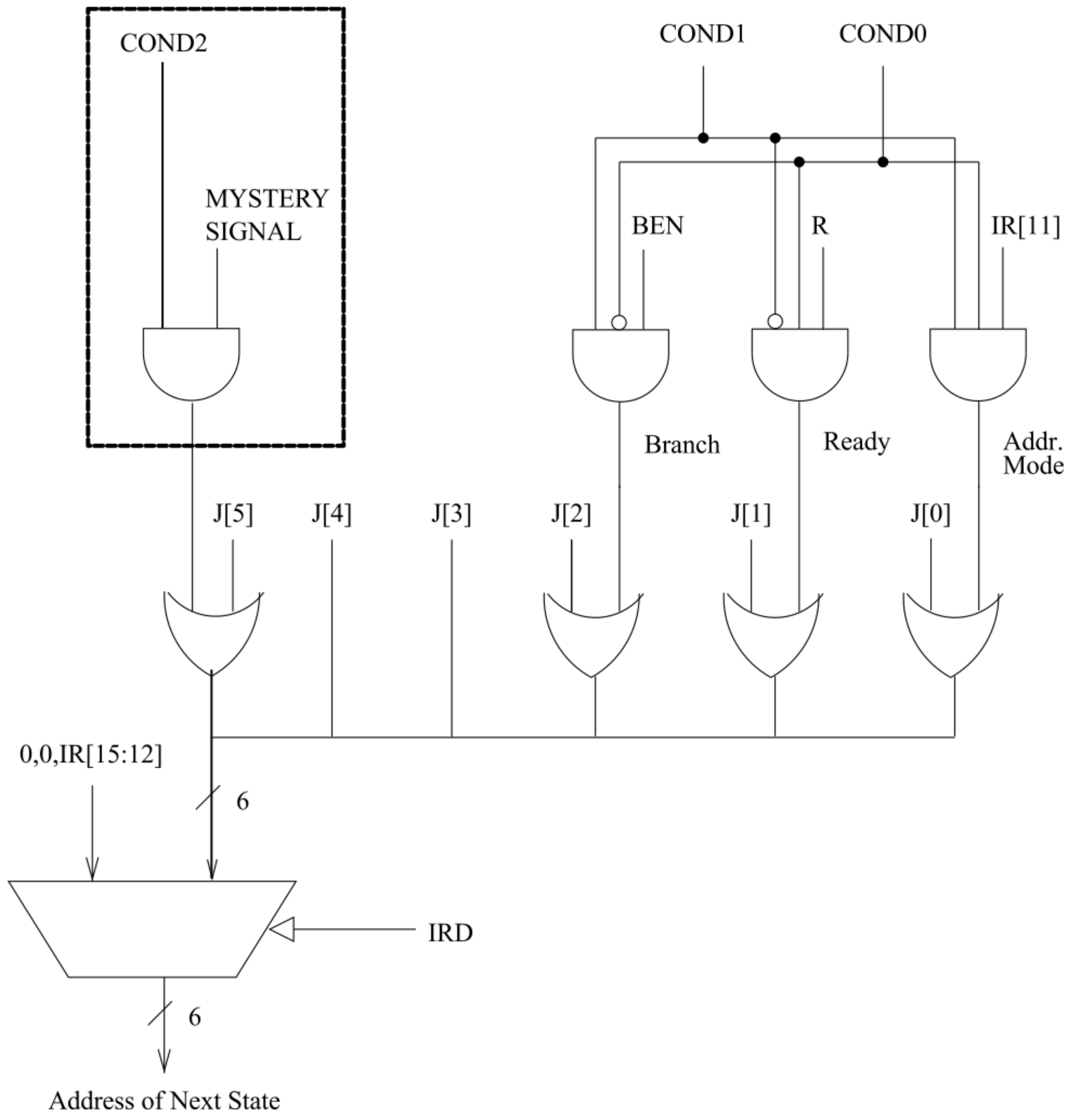
Mathematically, this can be expressed as $DR = \lfloor log_2 SR1 \rfloor$.

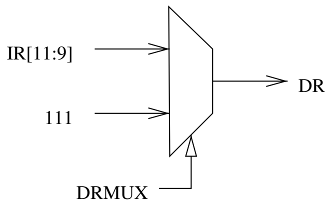
Note: This instruction is semantically the same as FINDFIRST in the VAX ISA.

Code:

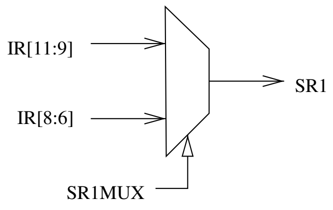
```
State 10: DR = 15
State 40: TEMP = SR1
          if (SR1 is negative)
             Go to State 18/19 (Fetch)
          else
             Go to State 50
State 50: DR = DR - 1
State 51: Left Shift TEMP
          if (TEMP is negative)
             Go to State 18/19 (Fetch)
          else
             Go to State 50
```


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

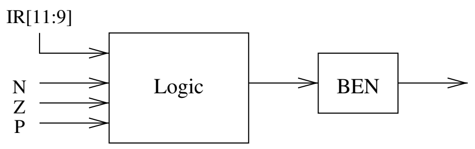

COND2
MYSTERY SIGNAL
COND1
COND0
BEN
R
IR[11]
Branch
Ready
Addr. Mode
J[5]
J[4]
J[3]
J[2]
J[1]
J[0]
0,0,IR[15:12]
6
IRD
6
Address of Next State



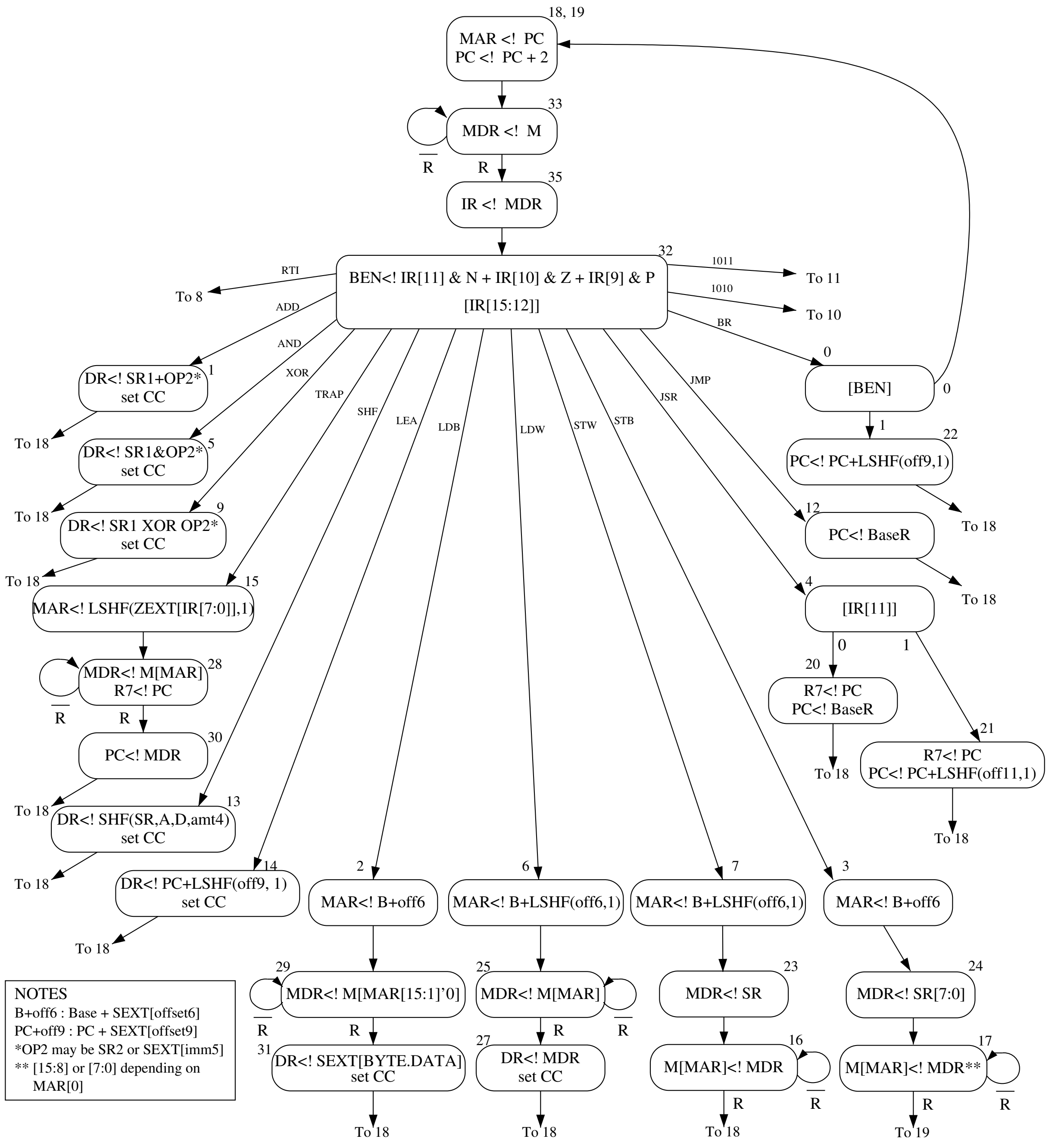
(a)



(b)



(c)



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]

MIPS Instruction Summary

Opcode	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 immediate 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, 100($2)`	`$1 = memory[$2 + 100]`
store word	`sw $1, 100($2)`	`memory[$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`