Department of Electrical and Computer Engineering
The University of Texas at Austin

EE 460N Fall 2014
Y. N. Patt, Instructor
Stephen Pruett, Emily Bragg, Siavash Zangeneh TAs
Exam 1
October 8, 2014

Name:_____

Problem 1 (20 points):_____

Problem 2 (15 points):_____

Problem 3 (20 points):_____

Problem 4 (20 points):_____

Problem 5 (25 points):_____

Total (100 points):_____

Note: Please be sure that your answers to all questions (and all supporting work that is required) are contained in the space provided.

Note: Please be sure your name is recorded on each sheet of the exam.

Please sign the following. I have not given nor received any unauthorized help on this exam.

Signature:_____

GOOD LUCK!

Name: _____

Problem 1 (20 points)

Part a (5 points): [   ] is a type of exception that allows the processor to complete the instruction before

taking the exception. [   ] does not allow the processor to complete the instruction before taking the

exception.

Part b (5 points): In class, we said a controller that does not want the bus forwards BG until it gets the NOT(BGin) signal from the PAU. On the I/O handout, we say the controller forwards BG until it gets the SACK signal. The correct answer is NOT(BGin), i.e., SACK does not work. Why?

Part c (5 points): The first Alpha processor wasted a clock cycle every time a branch was taken because it could not fetch the target address until after it decoded the branch instruction. Intel's Pentium chip had no such penalty. Why?

Part d (5 points): We use a bus to source some value, and then load it to a desired register. We wish to make our control store work with as few bits as necessary. If I have 16 sources for the bus, how many bits of control store do I need? If I have 16 destinations for the value, how many bits of control store do I need?

Source [   ]

Destination [   ]

Problem 2 (15 points)

A processor implements the GAg branch predictor as part of its microarchitecture. Assume the Branch History Register and Pattern History Table are as shown below when a branch is encountered. The direction of the most recent branch is the right-most bit of the BHR; i.e., 1=taken, 0=not_taken.

	PHT-before		PHT-after
0	11	0	
1	00	1	
2	01	2	
3	11	3	
4	01	4	
5	00	5	
6	10	6	
7	11	7	
8	10	8	
9	10	9	
10	00	10	
11	01	11	
12	10	12	
13	01	13	
14	11	14	
15	01	15	

1101

Table 1: BHR

Table 2: PHT-before

Table 3: PHT-after

Part a (1 point): Will the branch be predicted taken or not taken.

Part b (2 points): The branch is taken. Complete the table labeled PHT-after.

Part c (12 points): After this branch, the processor continues, executing three additional branches. At time T, the third additional branch has completed execution. Assume no additional branches have been encountered since that third branch was fetched. At time T, the counter in location #13 of the PHT is 01. Your job: complete the BHR at time T.

BHR at time T:

Problem 3 (20 points)

Assume a Tomasulo-style, out-of-order execution machine that handles ADD and MUL instructions. Instructions are of the form ADD Rx,Ry,Rz and MUL Rx,Ry,Rz, as discussed in class. Each instruction requires a fetch cycle, a decode cycle, two cycles of execution in the case of ADD and five cycles of execution in the case of MUL, and a final cycle to store the result into a register and/or a reservation station entry waiting for that result. A result is available to subsequent instructions after it is stored in a register or reservation station entry.

Assume a program consists of four instructions, with the first instruction fetched in cycle 1.

Shown below is a snapshot of the register file before cycle 1, the register file at the end of cycle x, and the reservation stations at the end of cycle x.

Reservation station entries are allocated in order, from top to bottom. That is, for example, the first MUL is allocated to the top reservation station entry associated with the mul functional unit, the second MUL with the second reservation station entry, etc. Each instruction remains in its reservation station until its result is stored.

R0	1	–	5
R1	1	–	7
R2	1	–	2
R3	1	–	3

Figure 1: Registers Before Cycle 1

R0	0	α	–
R1	0	π	–
R2	1	–	2
R3	1	–	4

Figure 2: Registers After Cycle X

	V	TAG	VALUE	V	TAG	VALUE
α	1	––	5	1	––	7
β	–	––	––	–	––	––
Σ	–	––	––	–	––	––

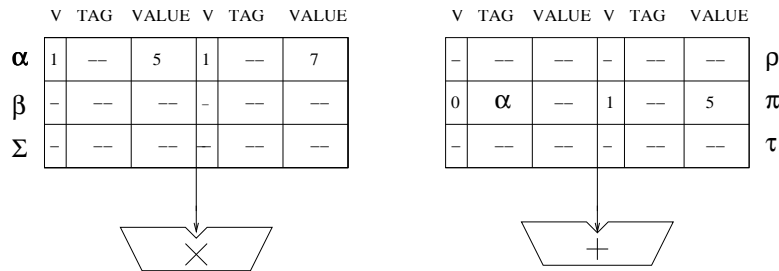|	V	TAG	VALUE	V	TAG	VALUE	
–	––	––	–	––	––	ρ
0	α	––	1	––	5	π
–	––	––	–	––	––	τ

Figure 3: Reservation Stations after Cycle X

The machine has one add functional unit and one mul functional unit. Neither is pipelined. The timing diagram below indicates the cycles (with the letter E) that each functional unit is busy in the execution phase of an instruction.

	1	2	3	4	5	6	7	8	9	10	11	12
Adder			E	E		E	E			E	E	
Multiplier				E	E	E	E	E				

Problem 3 continued

Part a (5 points): What is x?

Part b (15 points): Identify the four-instruction program that results in the snapshot of the register file and reservation stations at the end of cycle x. (Note: Identifying an instruction means identifying its opcode, its destination register, and its two source registers.)

| Instruction	Opcode	DR	SR1	SR2
I1				
I2				
I3				
I4				

Problem 4 (20 points)

In class, we discussed what is called a "pending" bus, which is characterized by the fact that the bus master holds onto the bus until the transaction is complete, regardless of how long it takes. There is a better way: a "split transaction" bus, whereby the master releases the bus while waiting for the slave to respond, allowing other transactions to occur. When the slave is ready to respond, it (the slave) acquires the bus as bus master and initiates the completion of the transaction.

Assume we have a multiplexed bus, and the processor wants to read from memory. Assume the memory access time is sufficiently long that a split-transaction bus is the right answer. Assume, for this problem only, no bus master can assert the same address lines until the transaction in progress has completed.

Your job:

Part a (10 points): Complete the timing diagram for the transaction.

Note: In order for this to work, when the Processor first asserts MSYN, BT, and Address, it (the processor) must save that Address. When the slave initiates the second half of the bus transaction, it asserts MSYN and the same address.
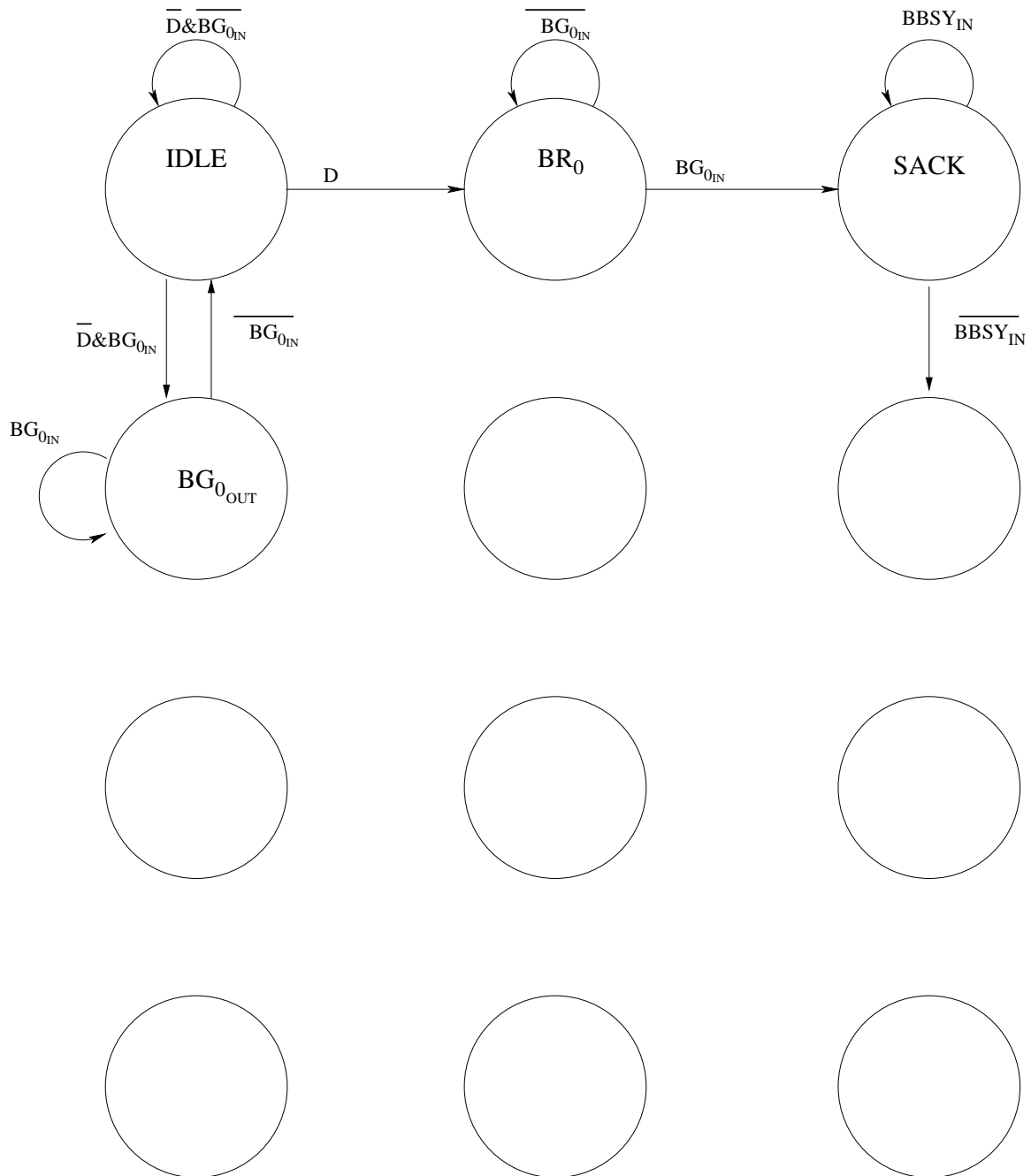
Processor   **Mem**

BBSY, MSYN, ADDRESS, BT=READ

Time

BBSY, MSYN, ADDRESS
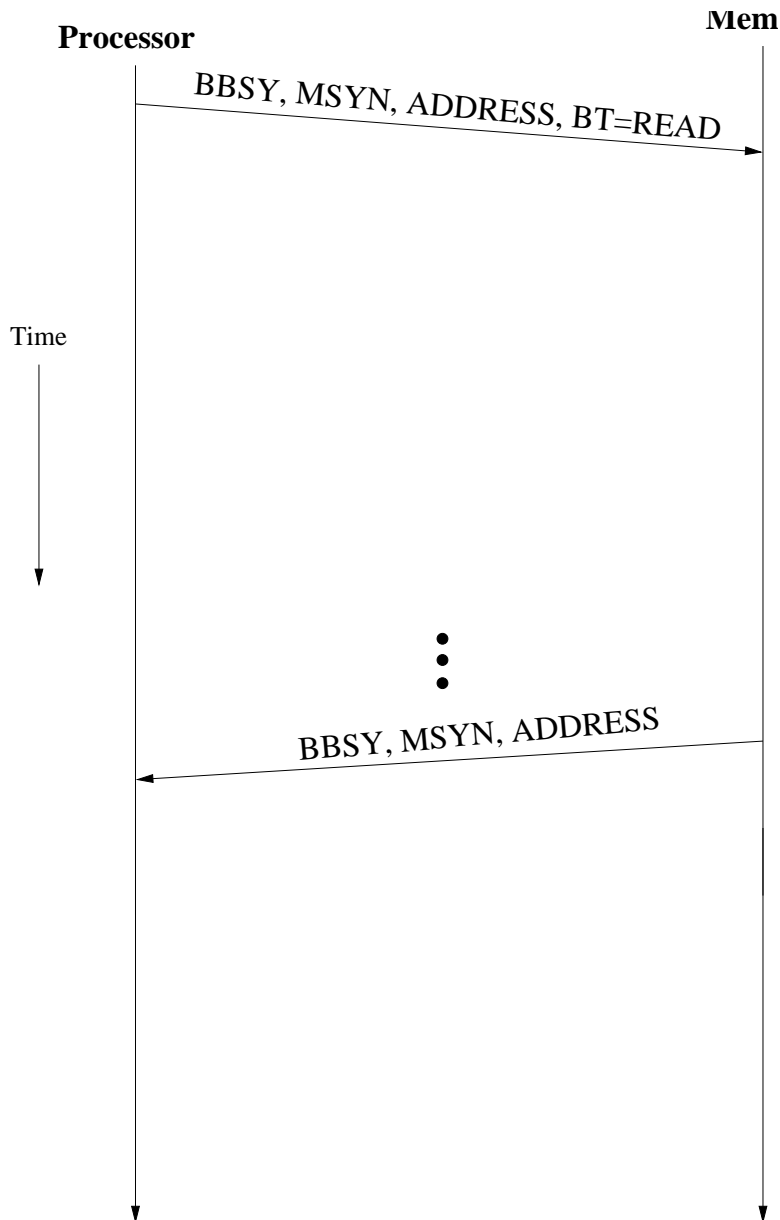
Problem 4 continued
Part b (10 points): Complete the state machine for the processor with those states that are necessary to complete the transaction (that is, ALL states AFTER the processor has asserted SACK).

Note: we have provided far more states than are necessary to complete this problem. Use only what you need.


States and transitions shown:

- $\overline{D} \& \overline{BG_{0_{IN}}}$ (self-loop on IDLE)
- $\overline{BG_{0_{IN}}}$ (self-loop on BR_0)
- $BBSY_{IN}$ (self-loop on SACK)
- IDLE $\xrightarrow{D}$ BR_0 $\xrightarrow{BG_{0_{IN}}}$ SACK
- IDLE $\xrightarrow{\overline{D} \& BG_{0_{IN}}}$ $BG_{0_{OUT}}$
- $BG_{0_{OUT}} \xrightarrow{\overline{BG_{0_{IN}}}}$ IDLE
- $BG_{0_{IN}}$ (self-loop on $BG_{0_{OUT}}$)
- SACK $\xrightarrow{\overline{BBSY_{IN}}}$ (next state)

Problem 5 (25 points):
Many ISAs provide an instruction that allows procedures to save all registers that the procedure will need to use, and then another instruction to restore those registers after the work of the procedure is done. We wish to add an instruction to the LC-3b to do exactly that.

The instruction SAVE_REGS has the following format:

	15		12	11		9	8	7		0
SAVE_REGS	1	0	1	0	BaseR		0		MASK	
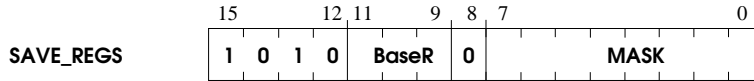
We use one of the unused opcodes, 1010. BaseR contains the starting address of the memory locations that will be used to save the registers. MASK is an 8bit bit vector that identifies which registers are to be saved. We encode them as follows:

Instruction[7] =1 means save register 7
Instruction[6] =1 means save register 6
Instruction[5] =1 means save register 5
...
Instruction[0] =1 means save register 0

As a result of execution of SAVE_REGS, the registers that are saved will be stored in contiguous memory locations, starting at the address specified by the intial contents of BaseR.

The ARM ISA has such an instruction and its Programmer's Reference Manual provides a footnote: If the register BaseR is to be saved, the instruction is not guaranteed to execute correctly. This is because ARM modifies BaseR during the execution of their version of that instruction. We will allow you that luxury in your implementation of SAVE_REGS, also.

Your job: Implement SAVE_REGS for the LC-3b.

Problem 5 continued:
Part a : Complete the state machine necessary to implement SAVE_REGS.
(Note: we start the state machine right after decode, with state #10.)
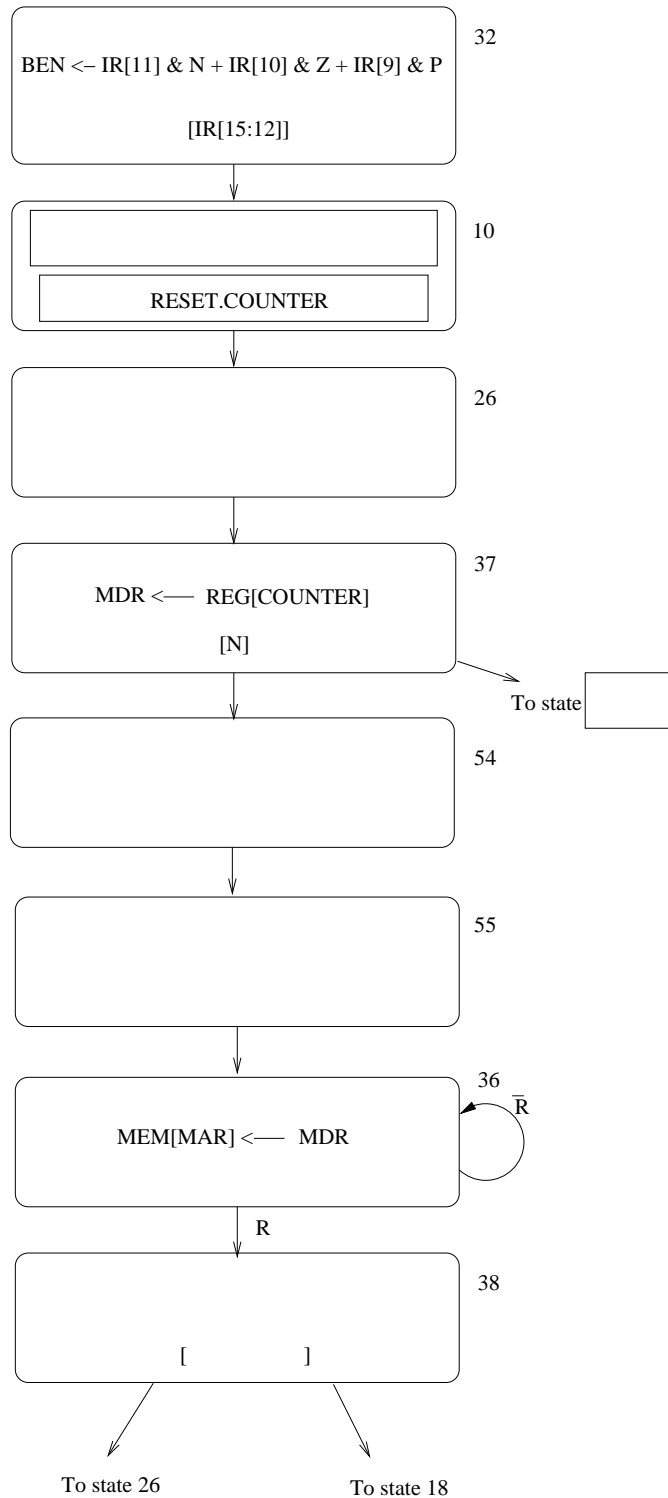(Hint: if you are having trouble with part a, it may help to first work on part b.)

Figure 1: State diagram for SAVE_REGS instruction

Name:

Problem 5 continued:

Part b : We have provided some of the structure needed in the data path to perform SAVE_REGS, in particular a TEMP register, and a 3-bit COUNTER. The COUNTER can be reset to 111 or decremented. Your job is to add whatever additional structures are needed in the dashed boxes in the data path diagram.

Figure 2: Modified Datapath for SAVE_REGS instruction

Problem 5 continued:

Part c : We have provided the original microsequencer for the LC-3b. You will note from the state machine that two additional microbranches are needed. Your job: Add the additional logic to account for these two microbranches.
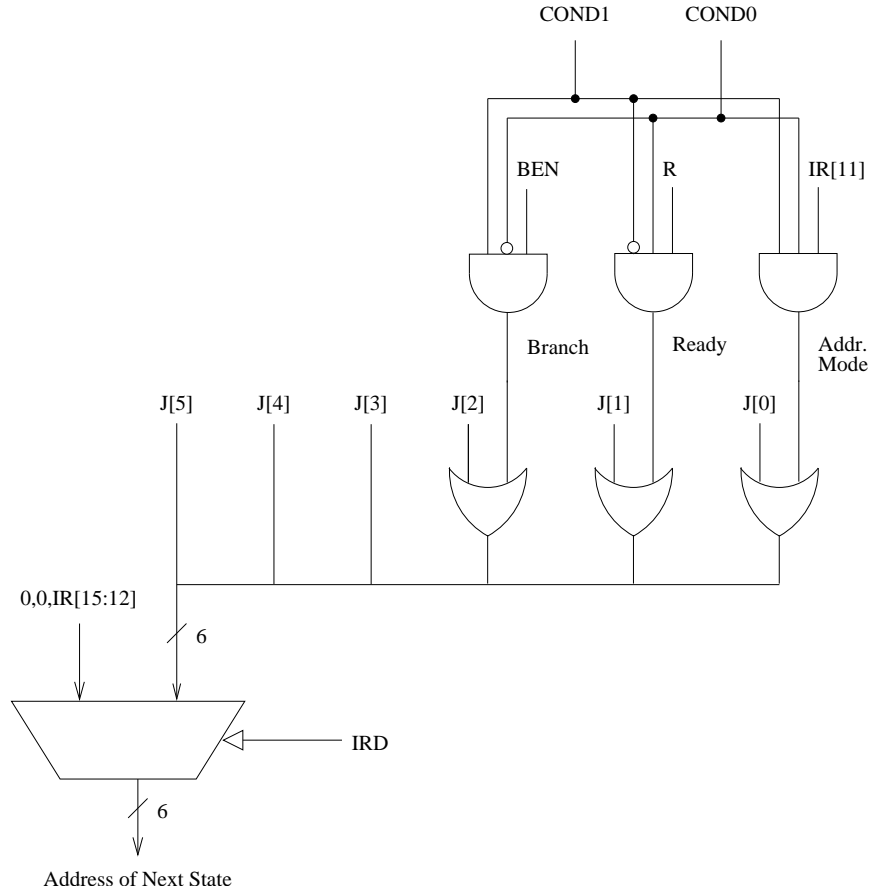
Part d : Complete the control store entries for states #37 and #38. (Note: We have provided a maximum of three additional control signals for your use. We have labeled them ECS1, ECS2, and ECS3 for Extra Control Signals. You may use as many of them as you need to do the job. Those you use should be properly identified on your data path. Mark "x" for don't care values.)

Instruction	15 14 13 12	11 10 9	8 7 6	5	4 3	2 1 0
ADD[+]	0001	DR	SR1	0	00	SR2
ADD[+]	0001	DR	SR1	1	imm5	
AND[+]	0101	DR	SR1	0	00	SR2
AND[+]	0101	DR	SR1	1	imm5	
BR	0000	n z p	PCoffset9			
JMP	1100	000	BaseR	000000		
JSR	0100	1	PCoffset11			
JSRR	0100	0 00	BaseR	000000		
LDB[+]	0010	DR	BaseR	boffset6		
LDW[+]	0110	DR	BaseR	offset6		
LEA[+]	1110	DR	PCoffset9			
NOT[+]	1001	DR	SR	1	11111	
RET	1100	000	111	000000		
RTI	1000	000000000000				
LSHF[+]	1101	DR	SR	0	0	amount4
RSHFL[+]	1101	DR	SR	0	1	amount4
RSHFA[+]	1101	DR	SR	1	1	amount4
STB	0011	SR	BaseR	boffset6		
STW	0111	SR	BaseR	offset6		
TRAP	1111	0000	trapvect8			
XOR[+]	1001	DR	SR1	0	00	SR2
XOR[+]	1001	DR	SR	1	imm5	
not used	1010					
not used	1011					

Figure 3: LC-3b Instruction Encodings

12

Table 4: Data path control signals

Signal Name	Signal Values		
LD.MAR/1:	NO(0), LOAD(1)		
LD.MDR/1:	NO(0), LOAD(1)		
LD.IR/1:	NO(0), LOAD(1)		
LD.BEN/1:	NO(0), LOAD(1)		
LD.REG/1:	NO(0), LOAD(1)		
LD.CC/1:	NO(0), LOAD(1)		
LD.PC/1:	NO(0), LOAD(1)		
GatePC/1:	NO(0), YES(1)		
GateMDR/1:	NO(0), YES(1)		
GateALU/1:	NO(0), YES(1)		
GateMARMUX/1:	NO(0), YES(1)		
GateSHF/1:	NO(0), YES(1)		
PCMUX/2:	PC+2(0)	;select pc+2	
	BUS(1)	;select value from bus	
	ADDER(2)	;select output of address adder	
DRMUX/1:	11.9(0)	;destination IR[11:9]	
	R7(1)	;destination R7	
SR1MUX/1:	11.9(0)	;source IR[11:9]	
	8.6(1)	;source IR[8:6]	
ADDR1MUX/1:	PC(0), BaseR(1)		
ADDR2MUX/2:	ZERO(0)	;select the value zero	
	offset6(1)	;select SEXT[IR[5:0]]	
	PCoffset9(2)	;select SEXT[IR[8:0]]	
	PCoffset11(3)	;select SEXT[IR[10:0]]	
MARMUX/1:	7.0(0)	;select LSHF(ZEXT[IR[7:0]],1)	
	ADDER(1)	;select output of address adder	
ALUK/2:	ADD(0), AND(1), XOR(2), PASSA(3)		
MIO.EN/1:	NO(0), YES(1)		
R.W/1:	RD(0), WR(1)		
DATA.SIZE/1:	BYTE(0), WORD(1)		
LSHF1/1:	NO(0), YES(1)		

Table 5: Microsequencer control signals

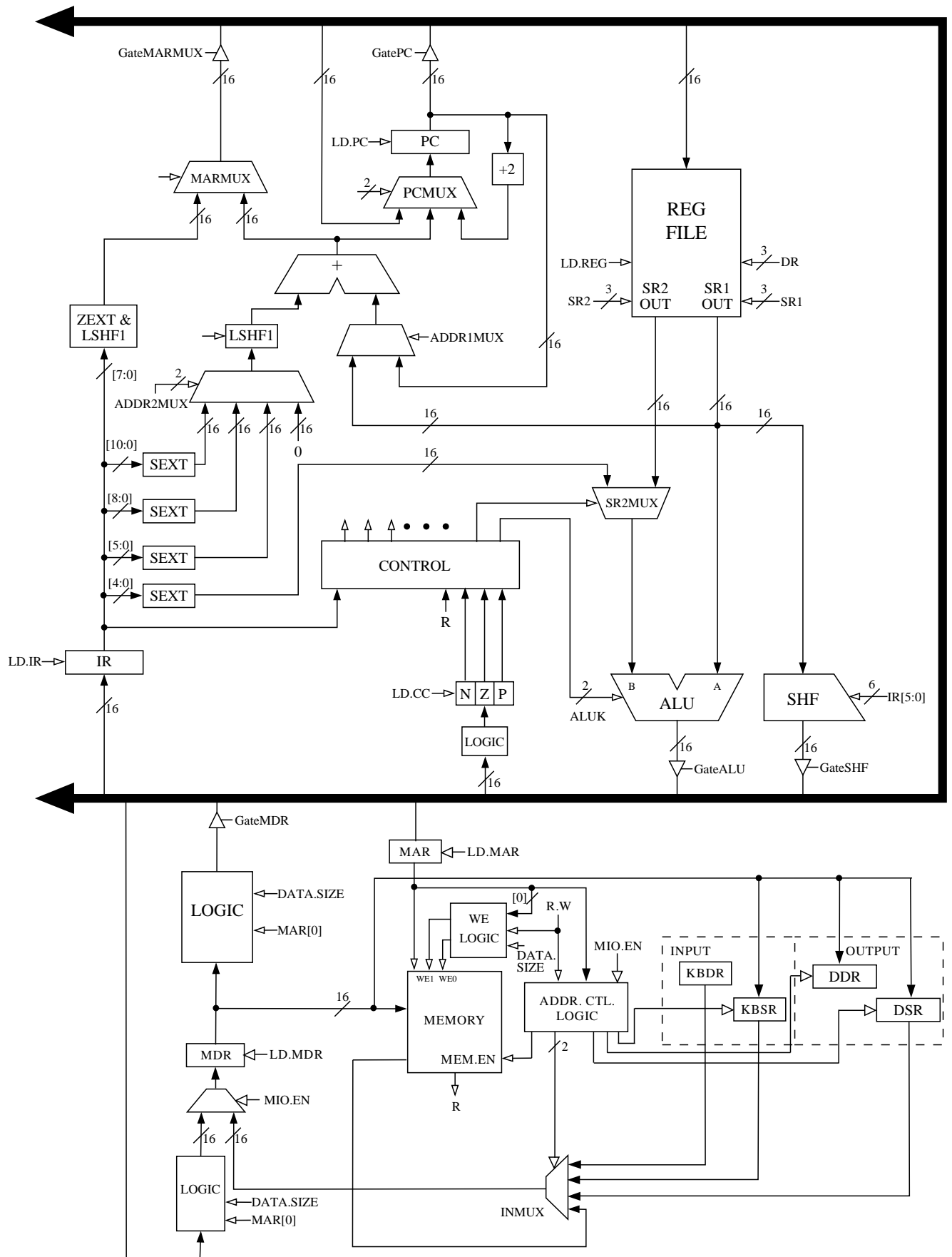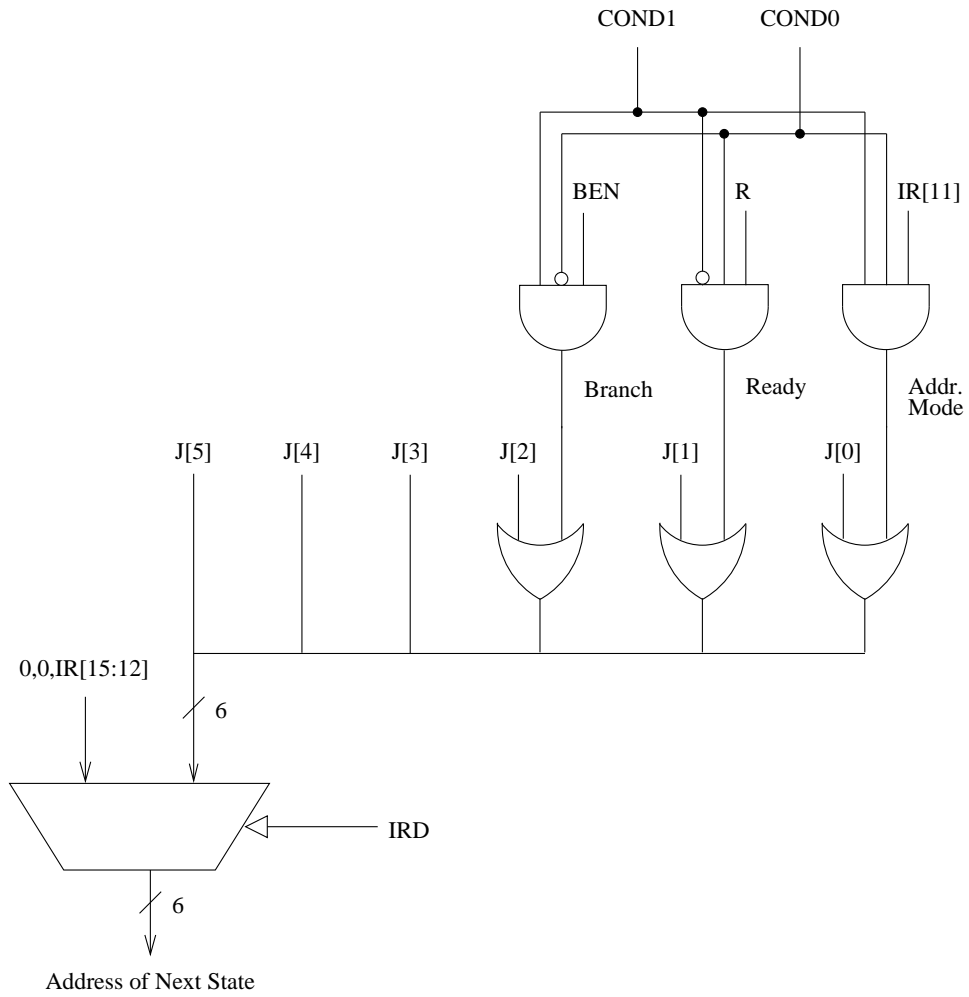| Signal Name	Signal Values	
J/6:		
COND/2:	$COND_0$	;Unconditional
	$COND_1$	;Memory Ready
	$COND_2$	;Branch
	$COND_3$	;Addressing Mode
IRD/1:	NO, YES	

Figure 4: A state machine for the LC-3b

Figure 5: The LC-3b data path

Figure 6: The microsequencer of the LC-3b base machine