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

EE 460N Spring 2015
Y. N. Patt, Instructor
Ben Lin, Kishore Punniyamurthy, Will Hoenig TAs
Exam 2
April 22, 2015

Name:____

Problem 1 (20 points):_______

Problem 2 (15 points):_______

Problem 3 (15 points):_______

Problem 4 (25 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!

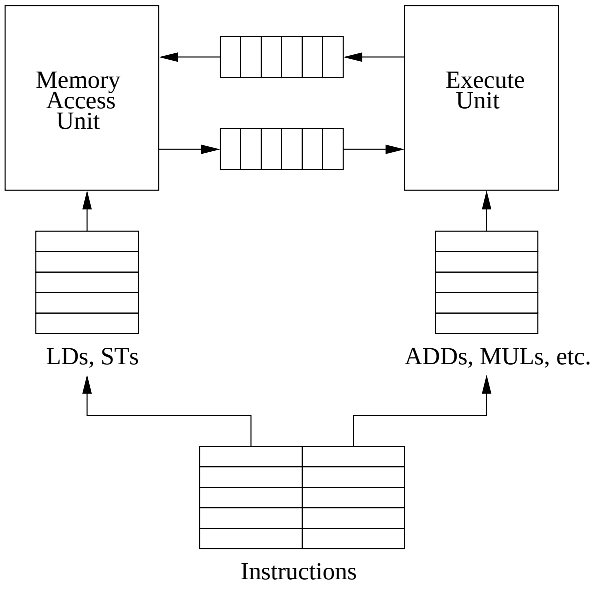
Name:___________________________________

Problem 1 (20 points)

Part a (5 points): A 7 by 6 word array (i.e. 7 rows by 6 columns) is stored in Row major order. Memory is word-addressable. We wish to load column 1 (i.e. the second column; column 0 is the first column) into a vector register. Before we initiate the vector load, what values must we load into VSTRIDE and VLEN registers.

VSTRIDE $\boxed{}$ VLEN $\boxed{}$.

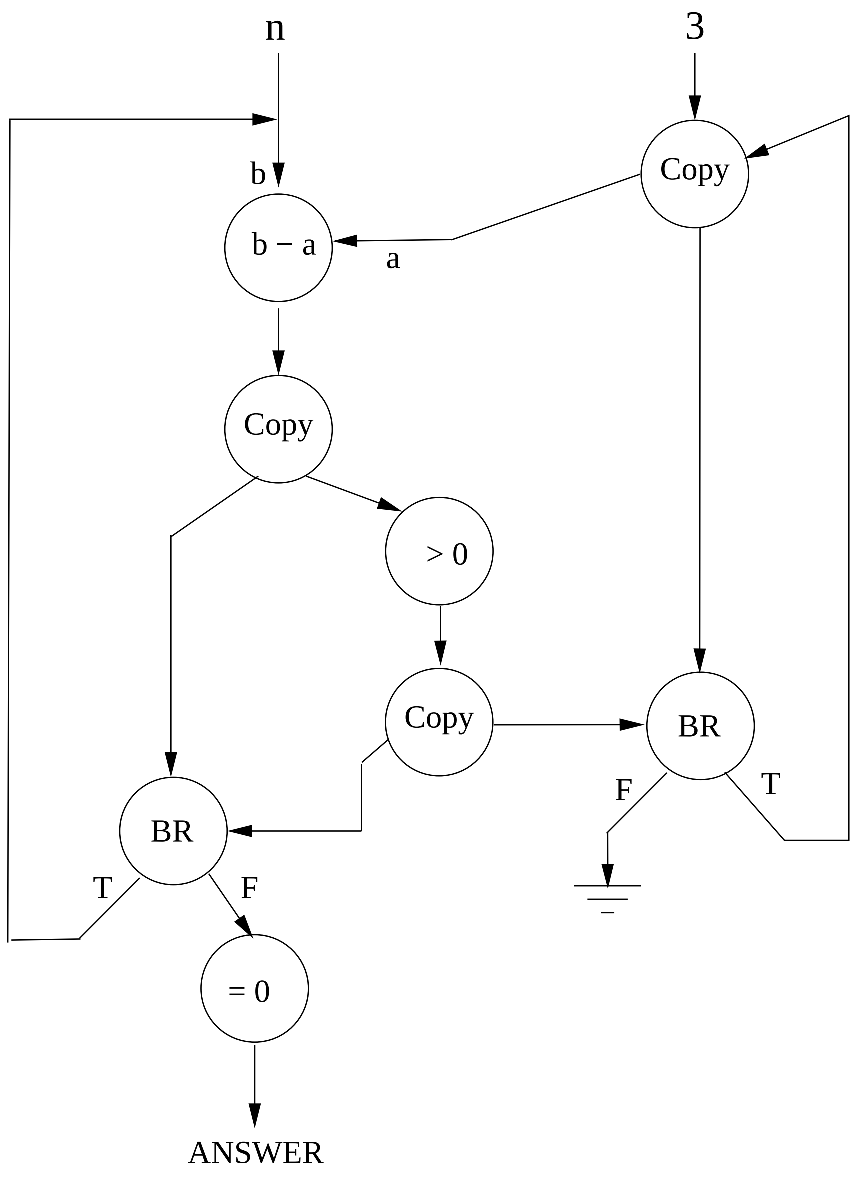
Part b (5 points): The Decoupled-Access-Execute (DAE) execution relaxed one constraint from VLIW, i.e., the parts of a wide instruction did not have to execute in lock-step. In fact, the access stream (LDs and STs) could slip ahead or behind the execute stream (ADDs, MULs, etc.). The diagram below shows the two streams.



One might think that allowing one stream to slip ahead or behind the other could cause the wrong result of a LD to be supplied to a subsequent ADD, for example. What characteristic of the execution model made this not a problem. (Ten words are more than enough to answer this question.

Name:__________________________________

Part c (10 points): The data flow graph shown below gets two inputs, the value N ($N > 0$) and the value 3. What is output at "Answer"?



Name:___________________________________

Problem 2 (15 points)

The numbers 25/8, 9/16, and 1/16 can all be representated exactly with a 7-bit floating point representation, in the format of the IEEE Floating Point standard. Note: One or more of the numbers may be represented as a subnormal number.

Part a: Show the representation of the three numbers, clearly identifying which bits are the fraction and which are the exponent.

25/8:

9/16:

The bias is:

1/16:

Part b: Add the three numbers in the following sequence. First add 25/8 and 9/16 and show the resulting 7-bit floating point number. Whenever necessary, round to zero, i.e., chop off the low order bits:

Now add 1/16 to the intermediate result, and show the final result here:

Part c: Add the same three numbers, this time by first adding 9/16 and 1/16. Show the result here. Again, whenever necessary, round to zero, i.e., chop off the low order bits:

Now add 25/8 to the intermediate result, and show the final result here.

Part d: Compare the final result of part b and of part c. If they are identical, explain why. If they are not identical, express the difference as a fraction and explain why.

Name:____________________________________

Problem 3 (15 points)

I learned recently that current ARM chips have improved on the standard method of processing interrupts in the following sense:

Suppose the machine is operating at priority 2 and an interrupt signals at priority 4. We push PSR and PC onto the system stack and initiate the interrupt. During the execution of the interrupt service routine, an interrupt at priority 3 signals. Since priority 3 is less than priority 4, we continue processing the priority 4 service routine. HOWEVER, When we execute RTI at the end of the priority 4 service routine, rather than do what we currently do, ARM immediately initiates the priority 3 service routine.

We wish to implement this improvement. Your job is to augment the state machine to make it happen. In this exercise, you can assume we do not have to be concerned with exceptions.

Part a (7 points): What do we currently do when we execute RTI from the priority 4 service routine and then intiate the priority 3 interrupt? Hint: Four accesses to the system stack are necessary to do this.

On the next page is an improved state machine that will allow, in our scenario, the priority 3 service routine to immediately initiate as part of the execution of RTI for the priority 4 service routine. In fact, by so doing, we will save three of the four accesses required in your answer to Part a.

We include below definitions of new signals we have used in the state machine: You may or may not need to look at them to understand what is going on.

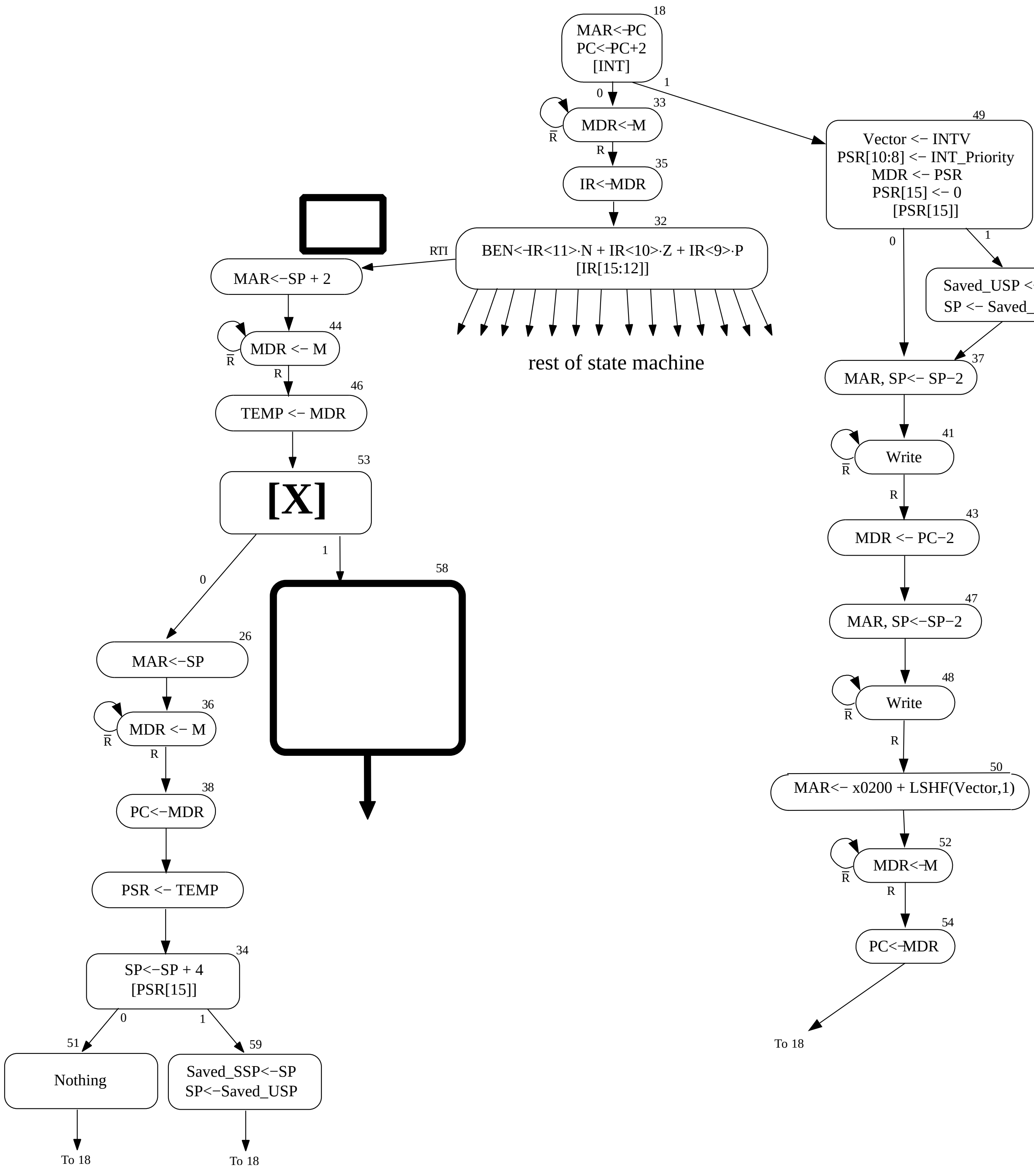
- INT_Priority: the priority of the highest priority interrupt which is currently pending
- INT: a signal signifying a pending interrupt whose priority is higher than the priority of the running process
  - i.e. INT = (INT_Priority $>$ PSR[10:8])
- INTV: the vector of the highest priority interrupt which is currently pending
- Vector: the vector of the interrupt which we will handle next
- Saved_USP: the saved User Stack Pointer
- Saved_SSP: the saved System Stack Pointer
- TEMP: a temporary register

Part b (8 points): Identify the appropriate modifications to the execution flow of RTI to accomplish this improvement. That is, identify X, identify the state that RTI takes you to from state 32 (shown in bold), fill in state 58 (shown in bold), and connect the arrow out of state 58 to the state you go to next.

X is

Name:___________________________________

A few hints about the state machine to help you get started: The flow starting at state 49 is the normal interrupt from states 18,19. The flow on the left is the flow for RTI. Normally, X is 0 and the RTI completes as usual along the left path. If a higher priority interrupt is pending, you instead go to state 58 to initiate processing the higher priority interrupt. Finally note that in state 44, you are loading into MDR the next to top element on the stack, NOT the top of the stack.



Name:____________________________________

Problem 4 (25 points)

A Vector Processor is attached to a multiplexed, asynchronous, pending bus. (Recall pending is the opposite of split-transaction.) A non-interleaved memory is also attached to this bus. We wish to perform a vector store, transferring the components of the vector register to memory locations during one very long bus cycle.

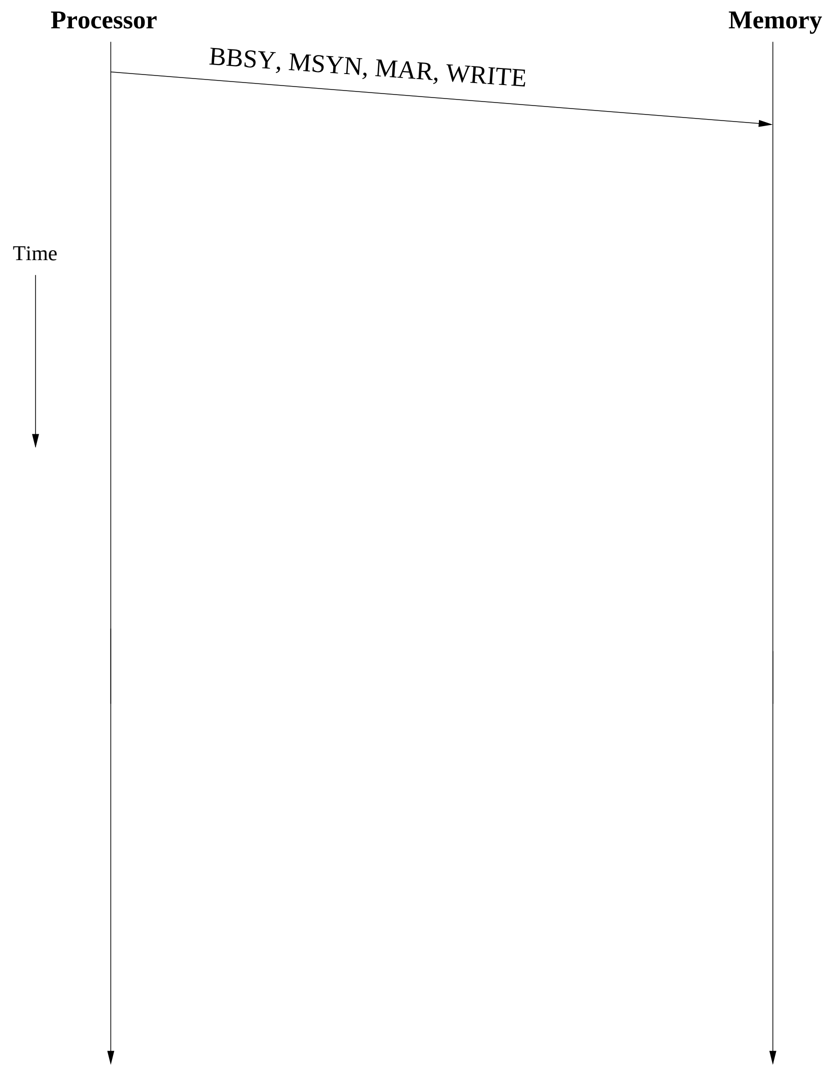
We assume that the controller on the memory side is a typical dumb controller. If it receives an address it latches it to an address register. If it receives data, it stores the data in the location specified by the address register. The controller on the memory side does no arithmetic.

The vector processor bus controller, on the other hand, contains all that is needed to make this work. It contains a 64-element BUFFER to hold the vector to be stored, an index register i, so we know which BUFFER[i] we are dealing with at any moment, a VSTRIDE register, a VLEN register, and a Memory Address Register (MAR). We load BUFFER, MAR, VSTRIDE, VLEN, and set i to 0 before the bus controller requests the bus.

Note: BUFFER entries are BUFFER[0] to BUFFER[VLEN-1].

Your job: complete the timing diagram and the state machine for the processor bus controller.

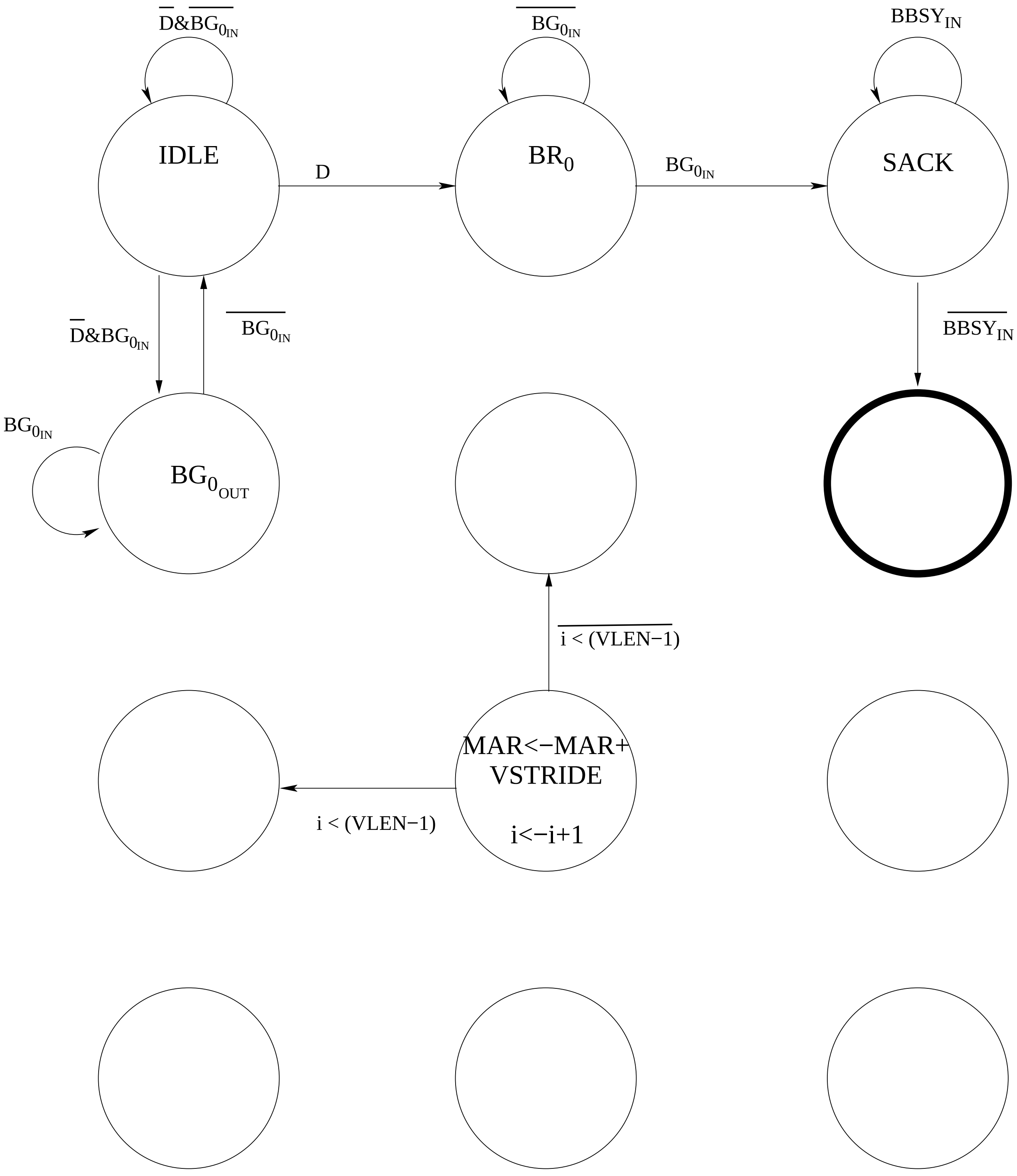
Two new signals are provided for you: M-CNTRL for the processor and S-CNTRL for memory.



Name:__________________________________

Note: Your job is to complete the Moore-model (not Mealy-model) state machine, starting with the state shown in boldface. You can assume VLEN is non-zero. Any signal not written in a state we will assume is not asserted while in that state.

Note that we have provided one state for you. This is the state in which MAR is updated, i is updated, and i is tested. Use this state as you find useful in your state machine.



Name:______________________________________

Problem 5 (25 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, some number of Execution cycles (2 for ADD, 5 for MUL), and a final cycle to store the result into a register and/or one or more reservation station entries that are waiting for the result. A result is available to subsequent instructions after it is stored in a register or reservation station entry.

A program fragment, consisting of four instructions, is executed on this machine. The first instruction is fetched in cycle 1. Your job: Complete the table below, i.e., the complete specification of the four instructions:

Instruction	Opcode	DR	SR1	SR2
1				
2				
3				R1
4				

Information on the next page will help you identify the four instructions executed.

Name:______________________________

The following information is provided to help you specify completely the four instructions executed. The machine has one adder and one multiplier. Neither is pipelined. Each has three reservation stations supplying it with instructions to execute.

The adder and multiplier are in use only during the cycles containing an E in the table below:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Adder			E	E								E	E				
Multiplier						E	E	E	E	E	E	E	E	E	E		

The Register Alias Table before cycle 1 and after cycle 9 are shown below. Note that some entries are missing. Part of your job is to fill them in.

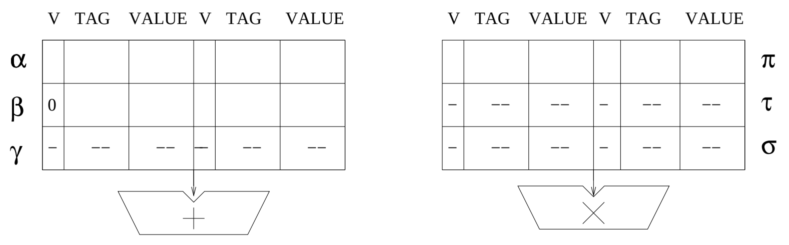
	V	TAG	VALUE
R0	1	–	3
R1	1	–	4
R2	1	–	0
R3	1	–	8

Registers Before Cycle 1

	V	TAG	VALUE
R0	1	–	3
R1	0		–
R2	1	–	7
R3	0		–

Registers After Cycle 9

The contents of the Reservation stations after cycle 4. Note that reservation stations are assigned from the top down, and that the topmost reservation station with both data entries valid is the next to be processed. Each instruction remains in its reservation station until its result is stored. Note also that some entries are not filled in. It is also part of your job to fill them in.

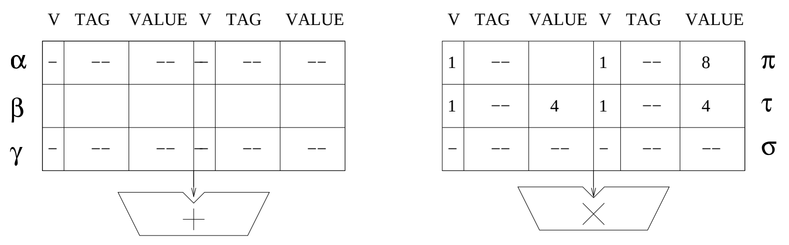


	V	TAG	VALUE	V	TAG	VALUE
α						
β	0					
γ	–	––	––	–	––	––

V	TAG	VALUE	V	TAG	VALUE	
						π
–	––	––	–	––	––	τ
–	––	––	–	––	––	σ

Reservation Stations after Cycle 4

The contents of the Reservation stations after cycle 9. Note that some entries are not filled in. It is also part of your job to fill them in.



	V	TAG	VALUE	V	TAG	VALUE
α	–	––	––	–	––	––
β						
γ	–	––	––	–	––	––

V	TAG	VALUE	V	TAG	VALUE	
1	––		1	––	8	π
1	––	4	1	––	4	τ
–	––	––	–	––	––	σ

Reservation Stations after Cycle 9

Part b (5 points): The design of this machine was not particularly well suited to out-of-order execution. What simple change would you make to it that would yield a lot more performance processing instructions out of program order (20 words or fewer).

	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 1: LC-3b Instruction Encodings

Table 1: 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 2: 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	

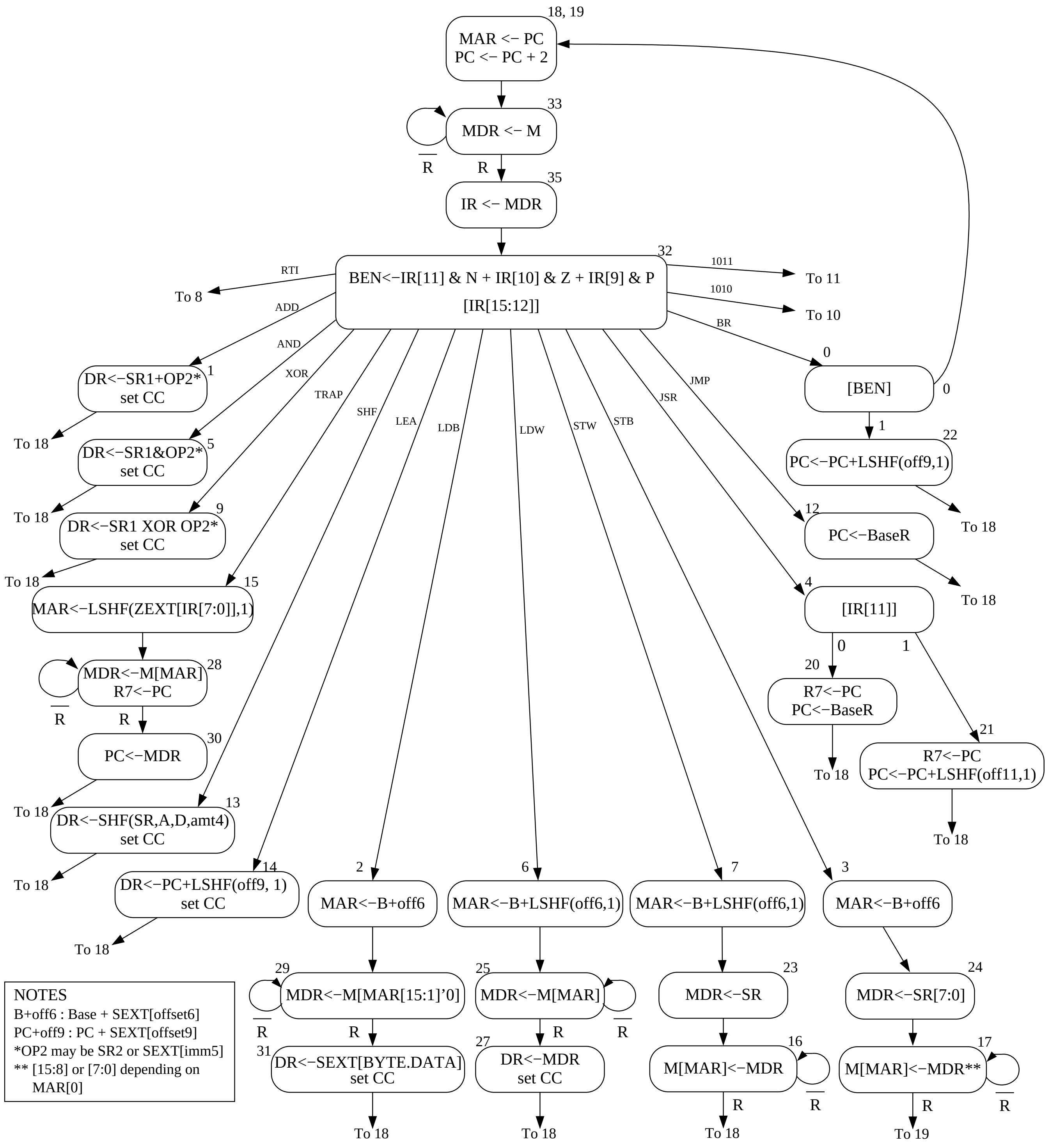


Figure 2: A state machine for the LC-3b

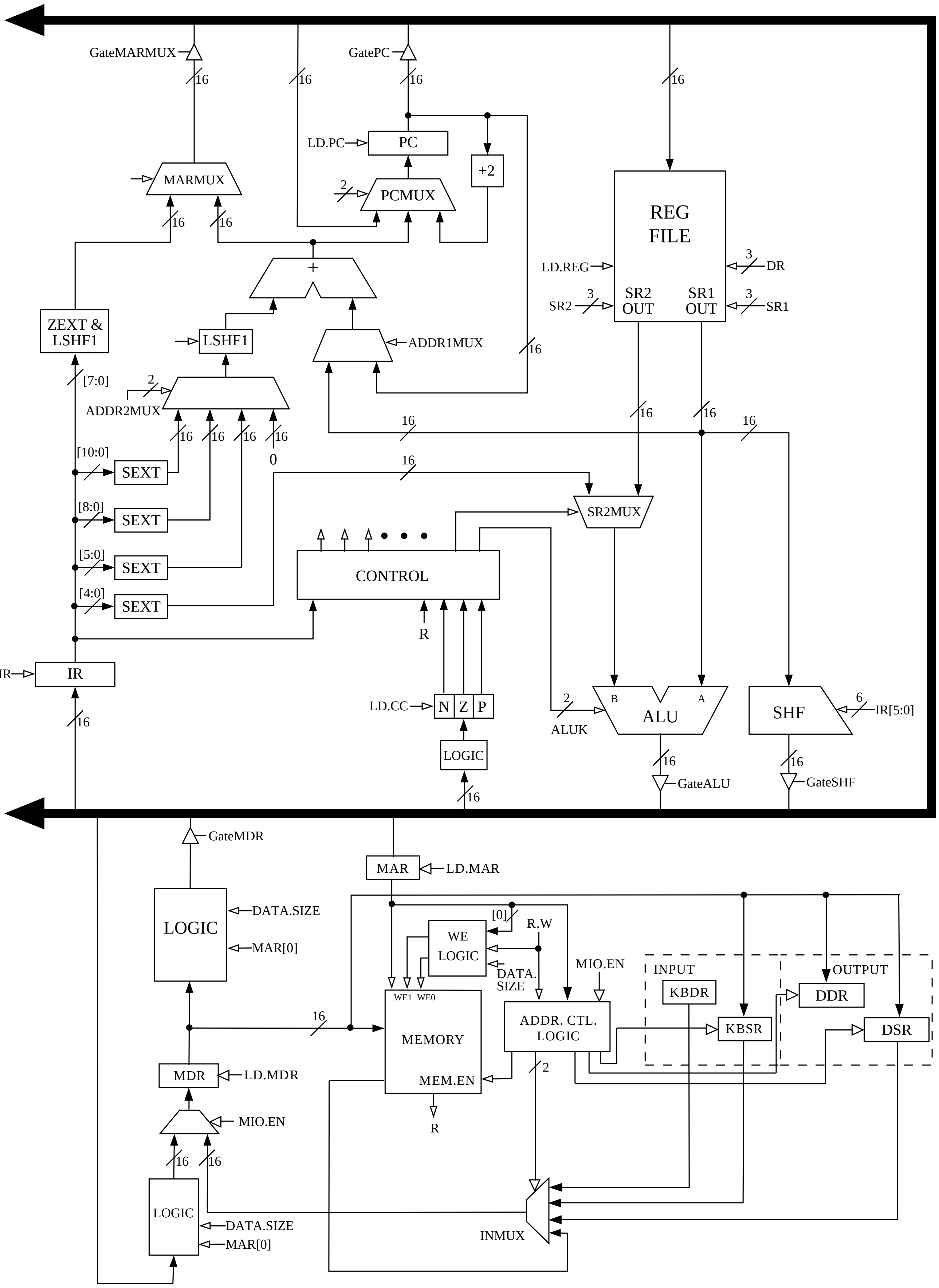


Figure 3: The LC-3b data path

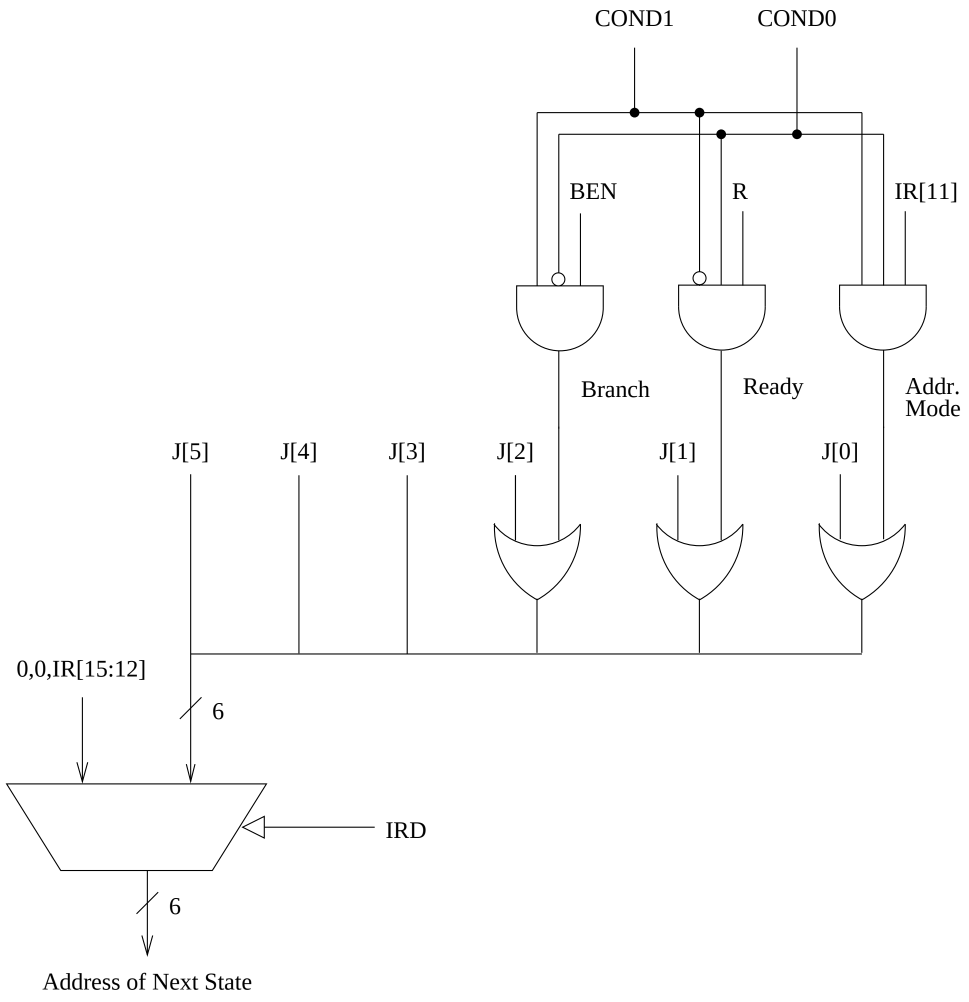


Figure 4: The microsequencer of the LC-3b base machine