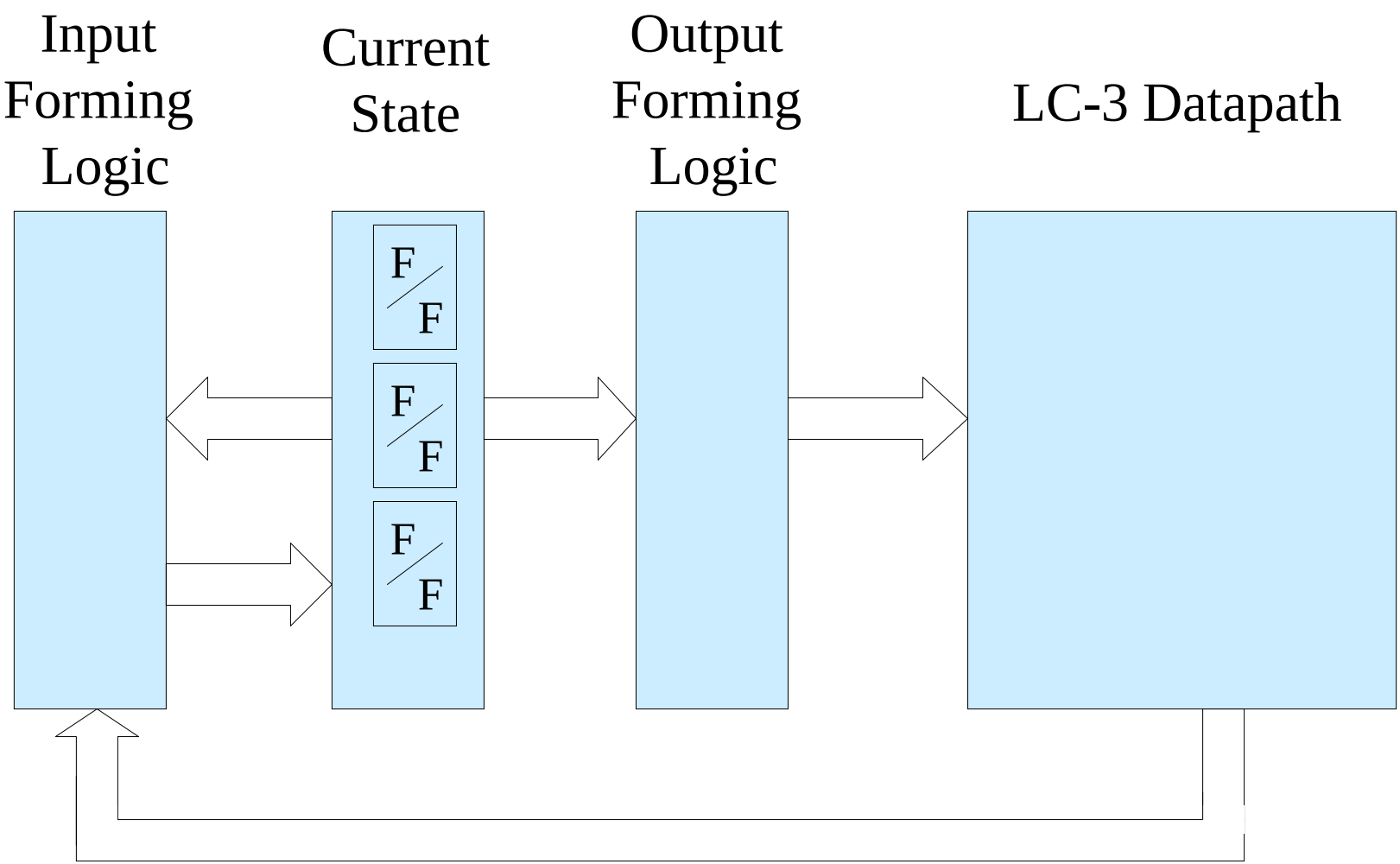
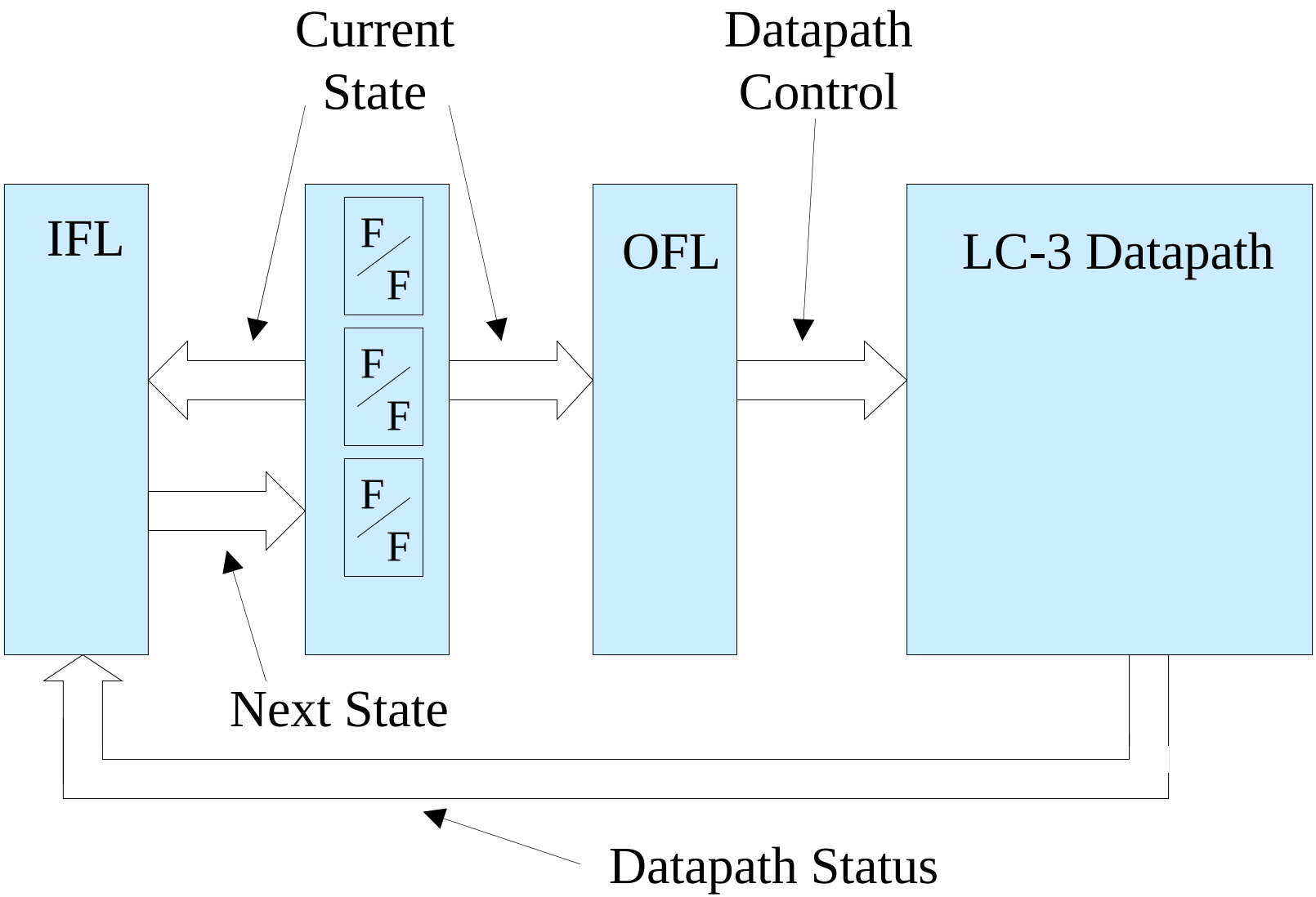
LC-3 Control and FSM Design


Input Forming Logic
Current State
Output Forming Logic
LC-3 Datapath
F
F
F
F
F
F


Current
State
Datapath
Control
IFL
F
F
F
F
F
F
OFL
LC-3 Datapath
Next State
Datapath Status

Instruction Fetch

1. Copy PC contents to MAR

   enaPC = 1 & ldMAR= 1

2. Perform memory read

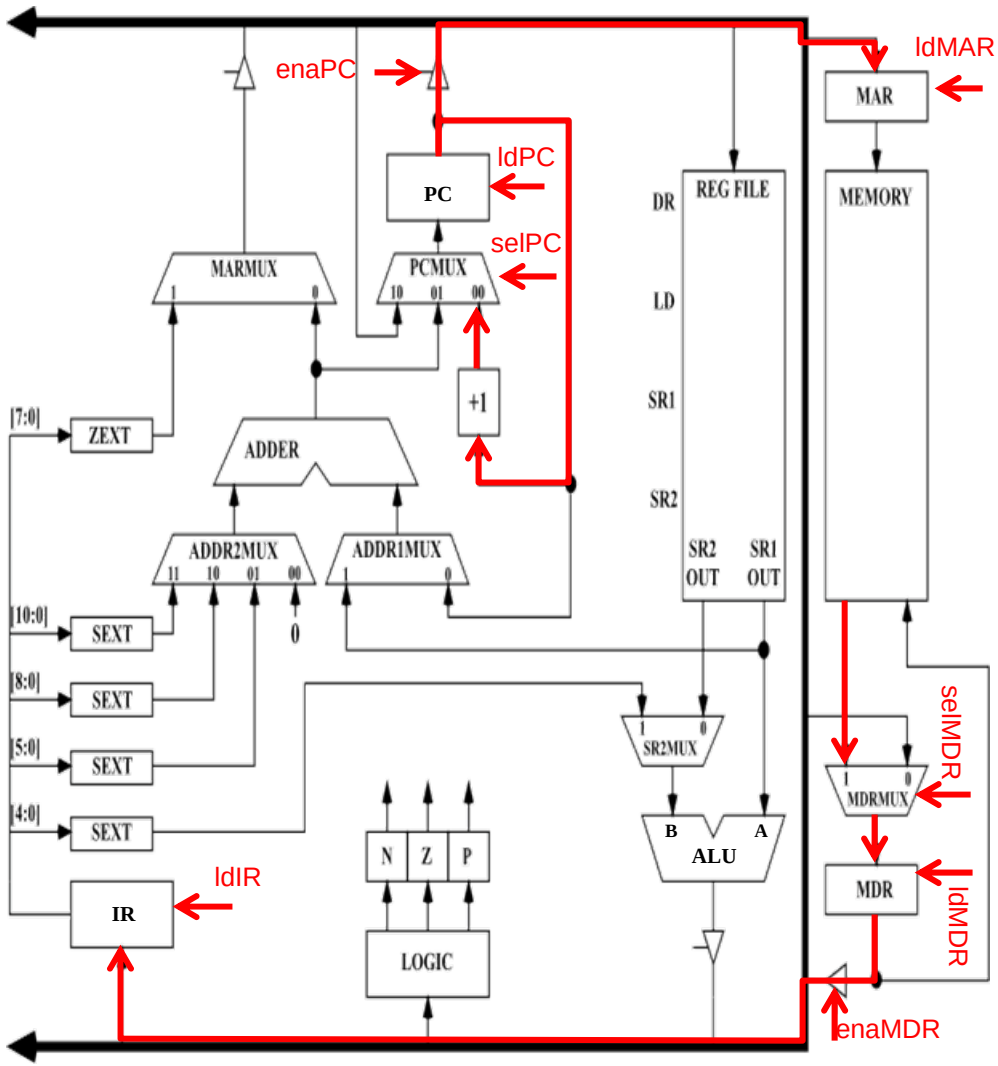
   selMDR=1 & ldMDR=1

   Increment PC

   selPC = 00 & ldPC = 1

3. Copy memory output register contents to IR

   enaMDR = 1 & ldIR = 1



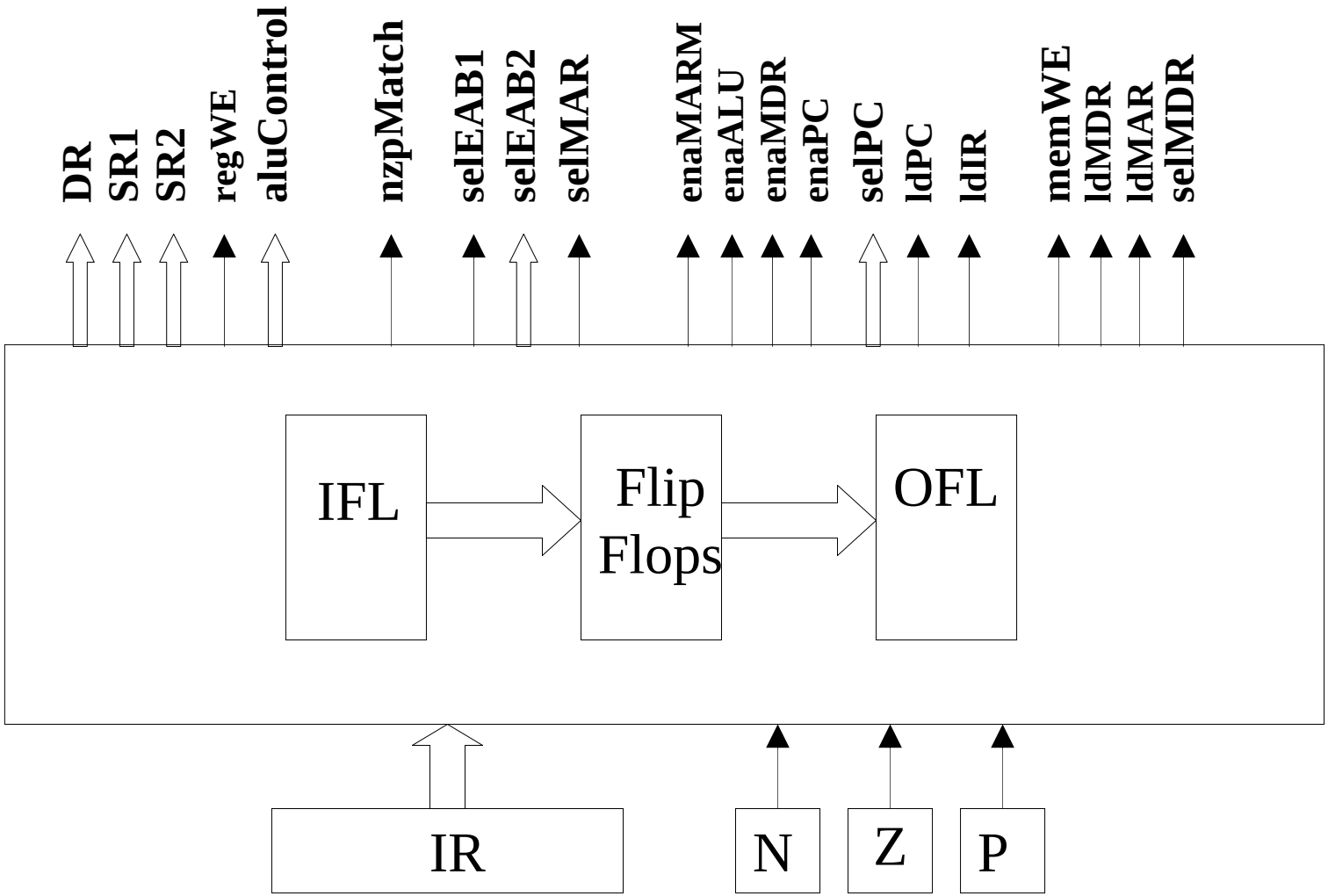
Fetch Control Sequence

Current State		Next State		Outputs							
Q1	Q0	N1	N0	**enaPC**	**ldMAR**	**selPC**	**ldPC**	**selMDR**	**ldMDR**	**enaMDR**	**ldIR**
0	0	0	1	1	1	xx	0	x	0	0	0
0	1	1	0	0	0	00	1	1	1	0	0
1	0	1	1	0	0	xx	0	x	0	1	1
1	1	--	--	?	?	??	?	?	?	?	?

The Control Logic

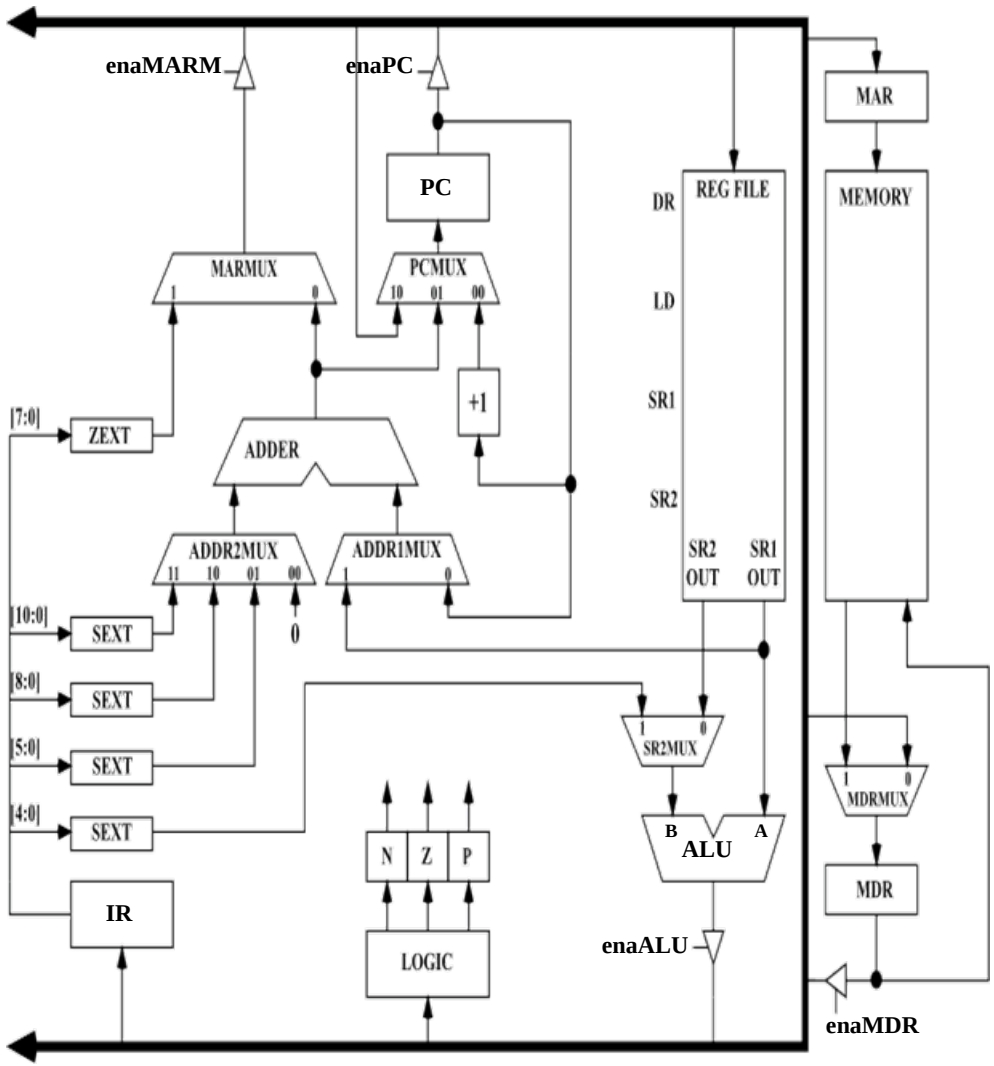
curState	IR[15:12]	IR[11]	nextState	aluControl	SR1	SR2	DR	selPC	selEAB1	selEAB2	enaALU	regWE	enaMARM	selMAR	enaPC	ldPC	ldIR	ldMAR	ldMDR	selMDR	memWE	enaMDR
fetch0	xxxx	x	fetch1	xx	xxx	xxx	xxx	xx	x	xx	0	0	0	x	1	0	0	1	0	x	0	0
fetch1	xxxx	x	fetch2	xx	xxx	xxx	xxx	00	x	xx	0	0	0	x	0	1	0	0	1	1	0	0
fetch2	xxxx	x	decode	xx	xxx	xxx	xxx	xx	x	xx	0	0	0	x	0	0	1	0	0	x	0	1

The Control Logic



LC3-DC

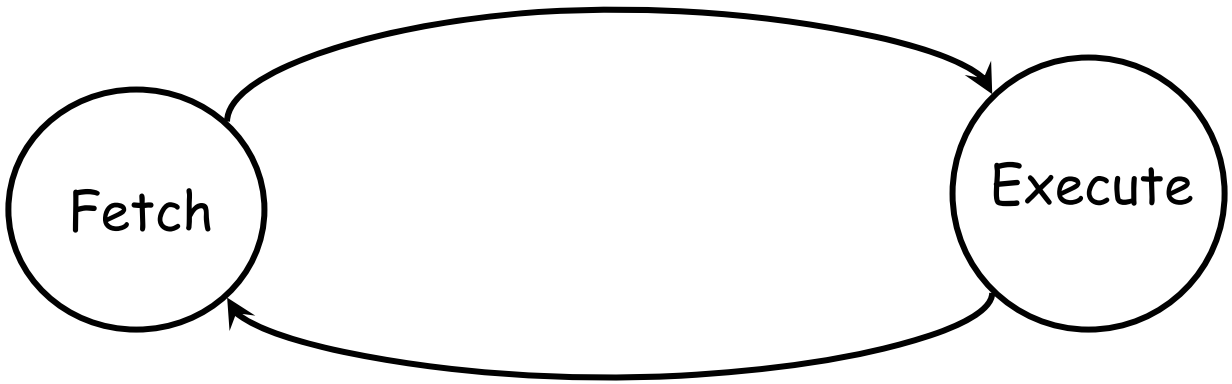
Designing The LC-3 Control



The Von Neumann Model

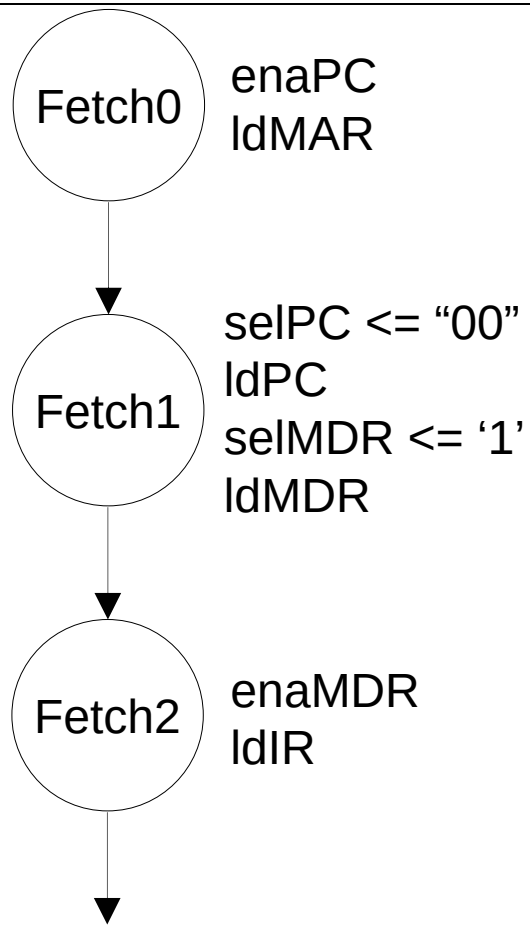
- Fetch an instruction
- Execute it
- Fetch the next instruction

continue ...



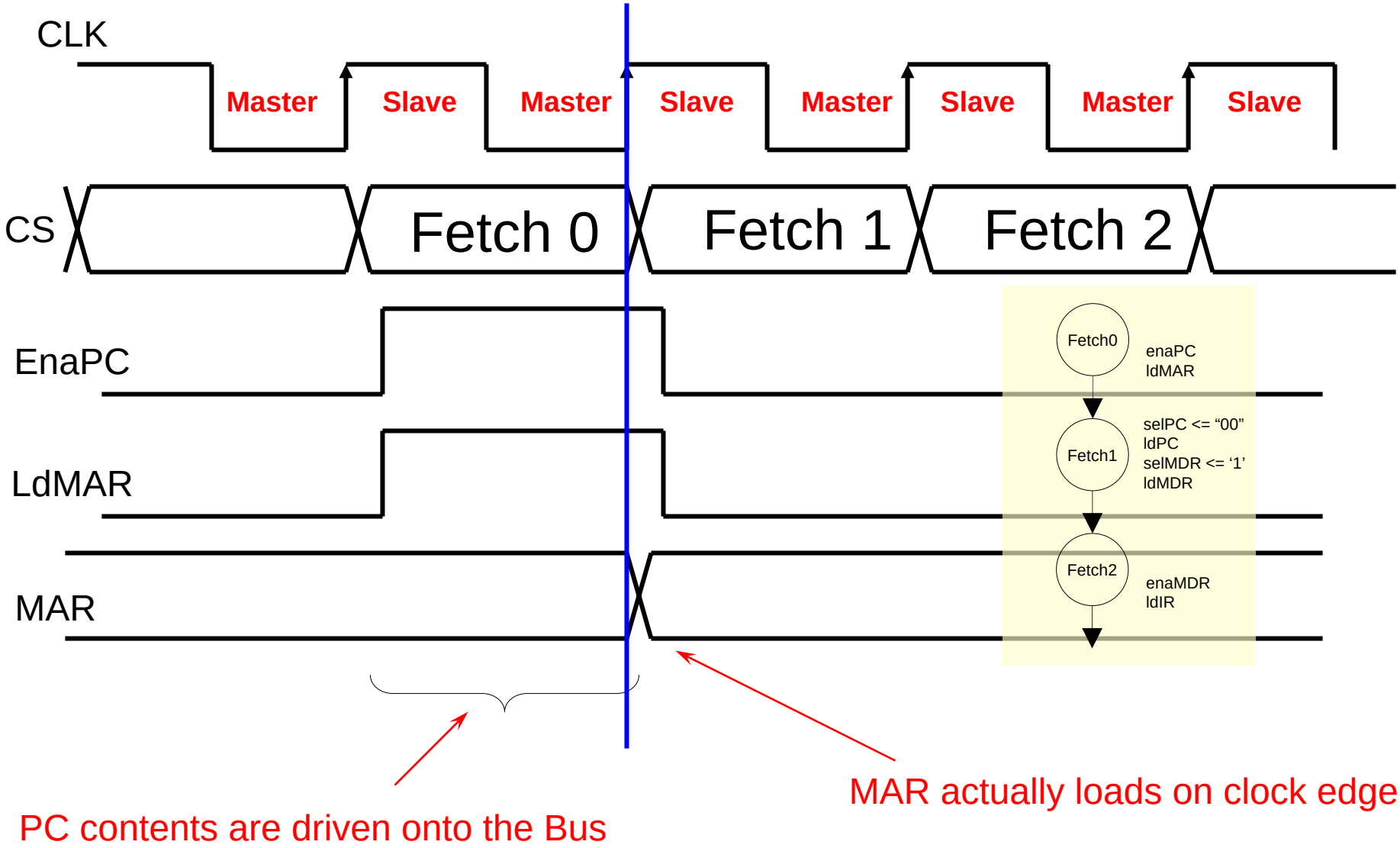
The Fetch Cycle

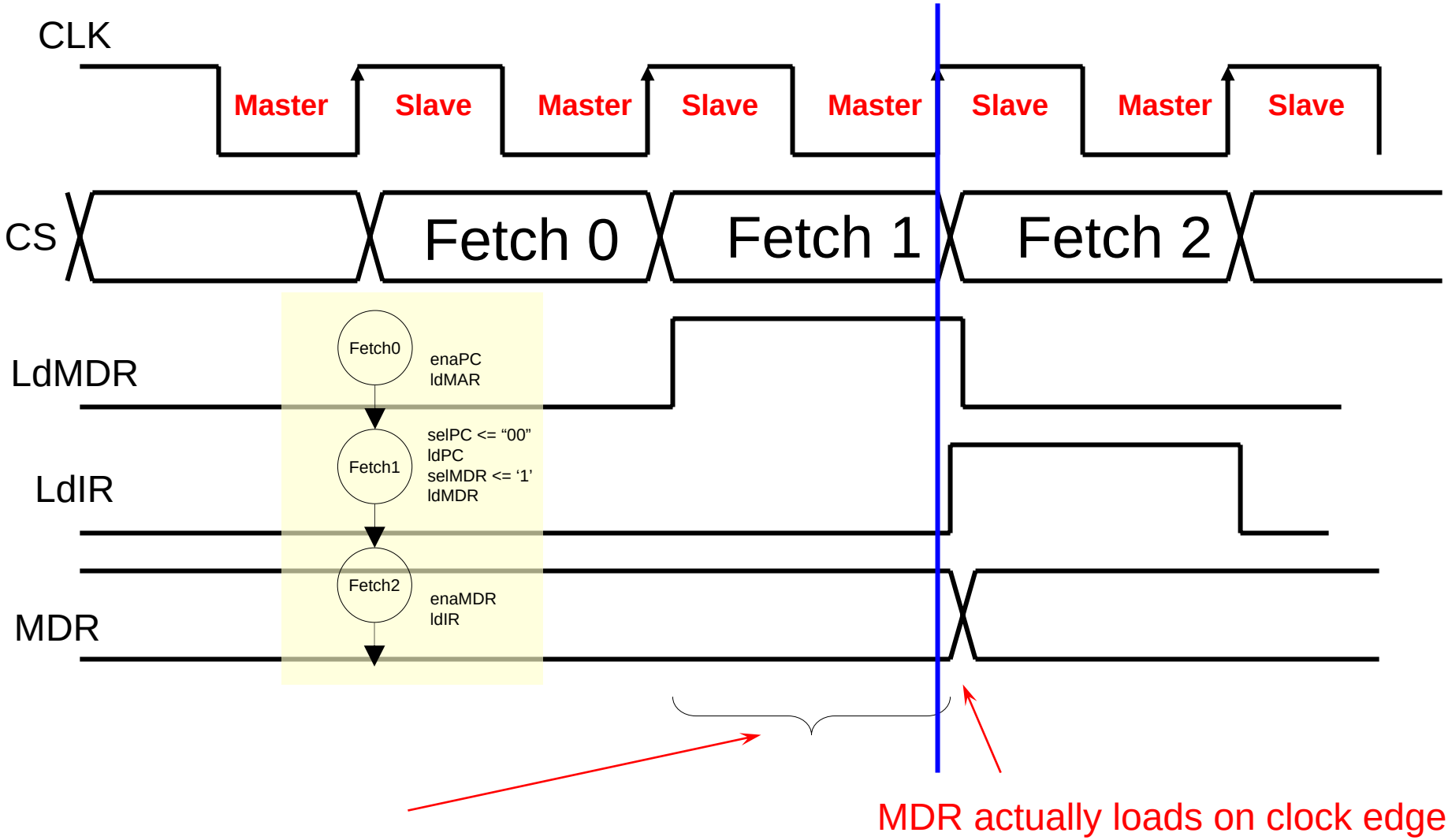
1. PC ➔MAR
2. PC ➔ PC+1,
   Mem[MAR] ➔ MDR
3. MDR ➔ IR

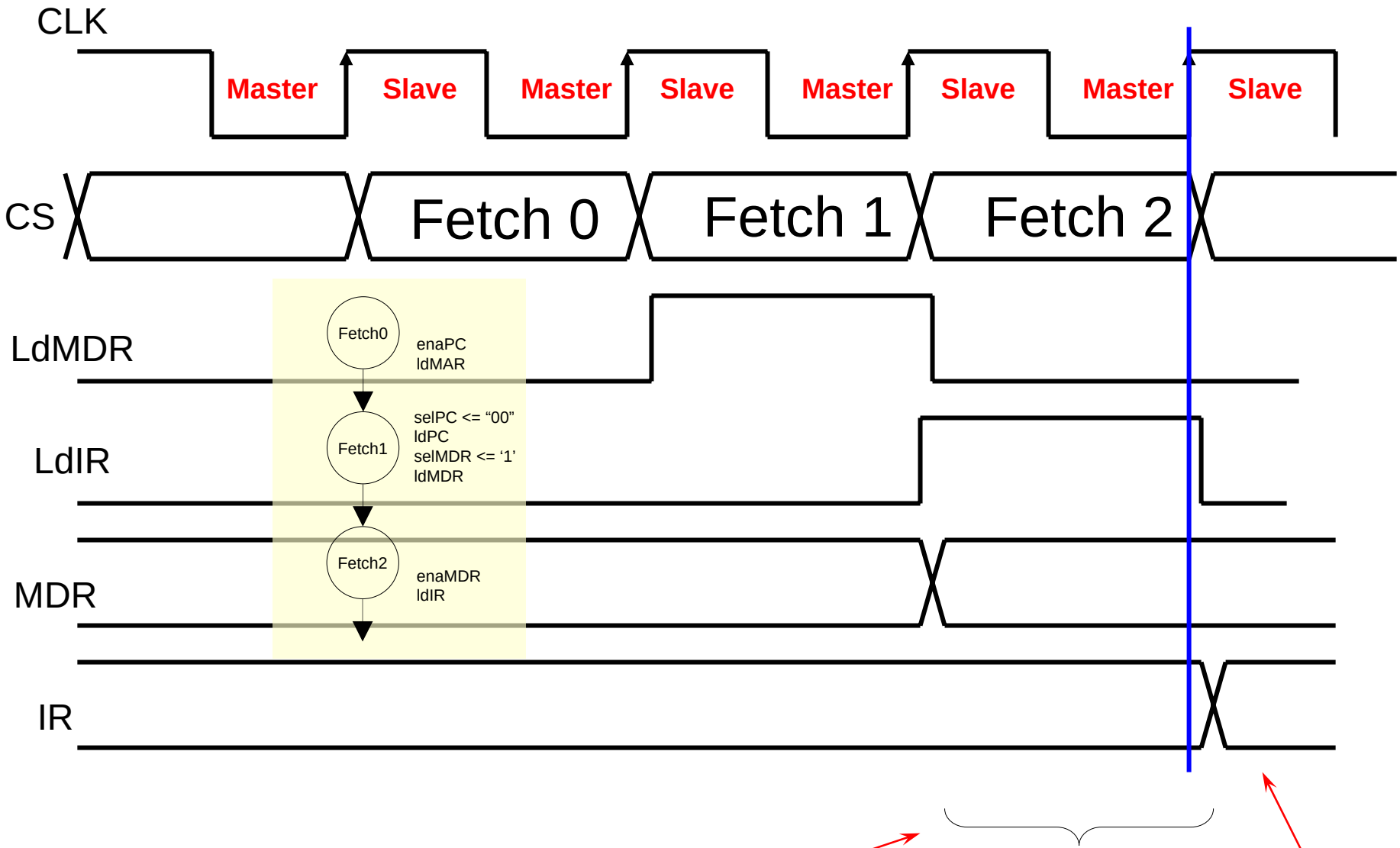


A Note on Timing

- In all cases:
  - Buses are driven and muxes are selected *during* a state
  - Registers and memory inputs are latched on the rising clock edge at the end of the state


CLK
Master
Slave
Master
Slave
Master
Slave
Master
Slave
CS
Fetch 0
Fetch 1
Fetch 2
EnaPC
LdMAR
MAR
Fetch0
enaPC
ldMAR
Fetch1
selPC <= "00"
ldPC
selMDR <= '1'
ldMDR
Fetch2
enaMDR
ldIR
PC contents are driven onto the Bus
MAR actually loads on clock edge

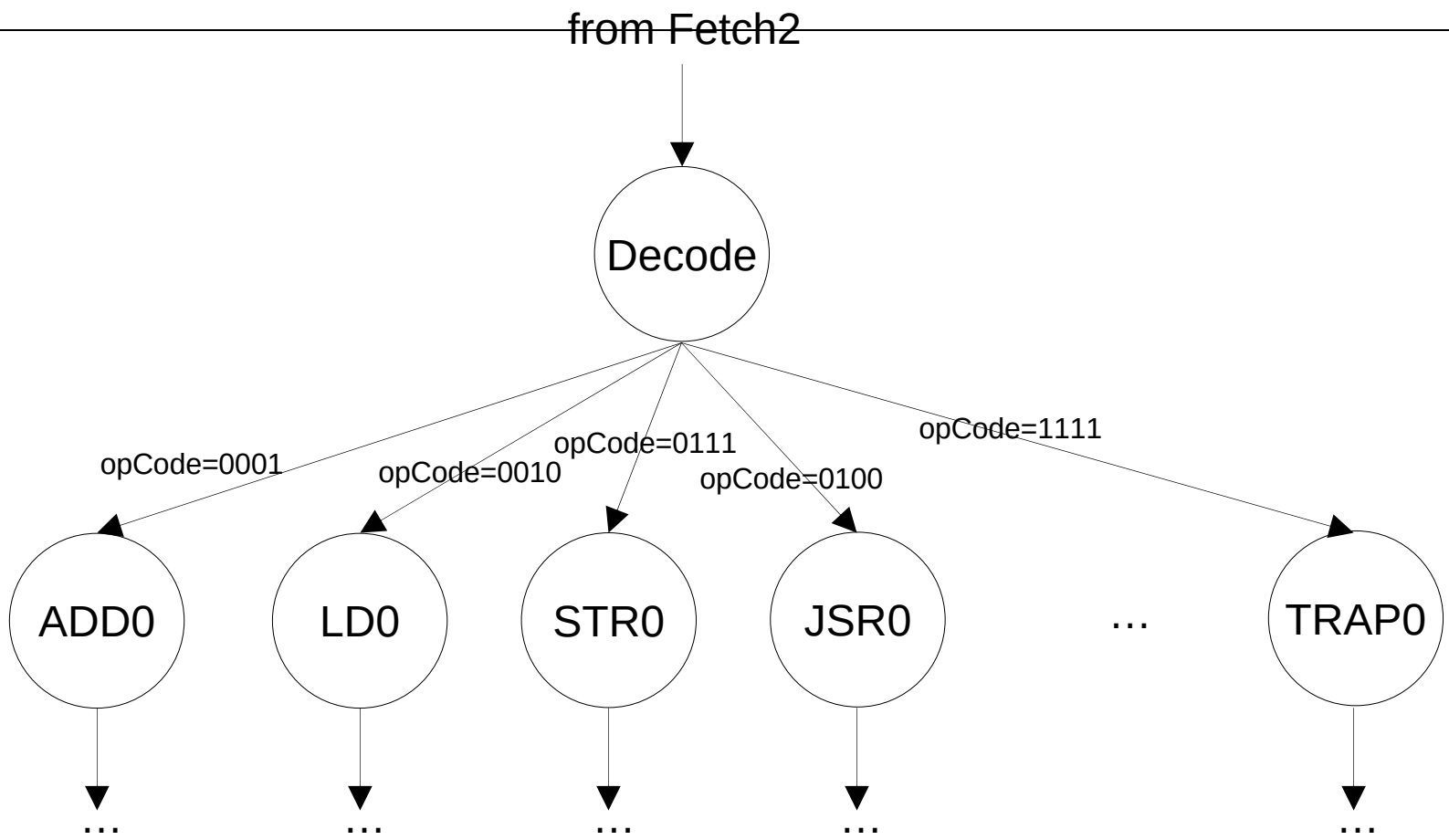

CLK
Master
Slave
Master
Slave
Master
Slave
Master
Slave
CS
Fetch 0
Fetch 1
Fetch 2
LdMDR
LdIR
MDR
Fetch0
enaPC
ldMAR
Fetch1
selPC <= "00"
ldPC
selMDR <= '1'
ldMDR
Fetch2
enaMDR
ldIR
Memory is being accessed
and generates its output
MDR actually loads on clock edge


CLK
Master
Slave
Master
Slave
Master
Slave
Master
Slave
CS
Fetch 0
Fetch 1
Fetch 2
LdMDR
LdIR
MDR
IR
Fetch0
enaPC
ldMAR
Fetch1
selPC <= "00"
ldPC
selMDR <= '1'
ldMDR
Fetch2
enaMDR
ldIR
MDR output is driven onto bus
IR actually loads on clock edge

A Note on PC Incrementing

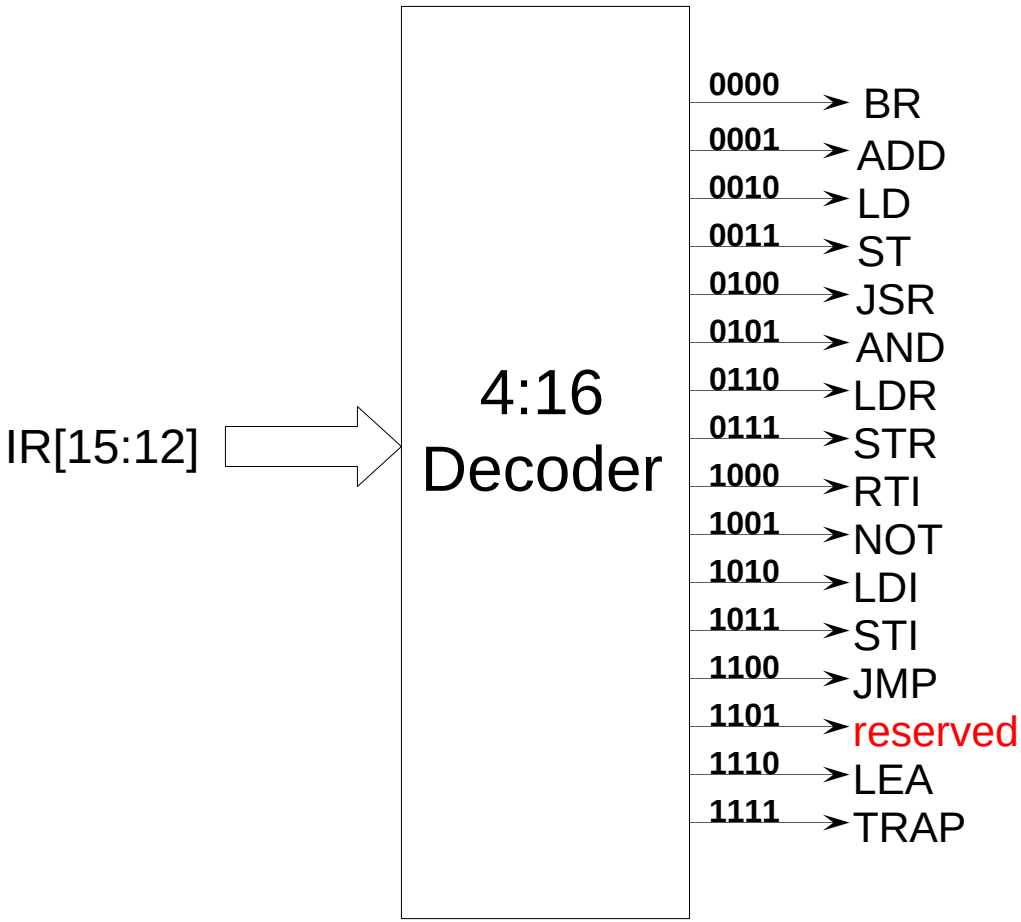
- The approach taken is to *always* increment the PC during the fetch process
  - This assumes that the next instruction to be executed will always be at location PC+1
- In the case of a branch or jump, the PC will get loaded with a different value during the execution phase of the instruction.

The Decode State

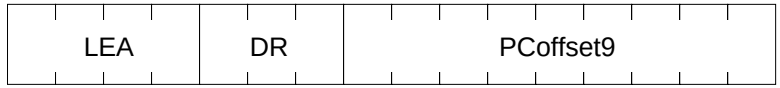


The only state with a conditional branch in the machine…

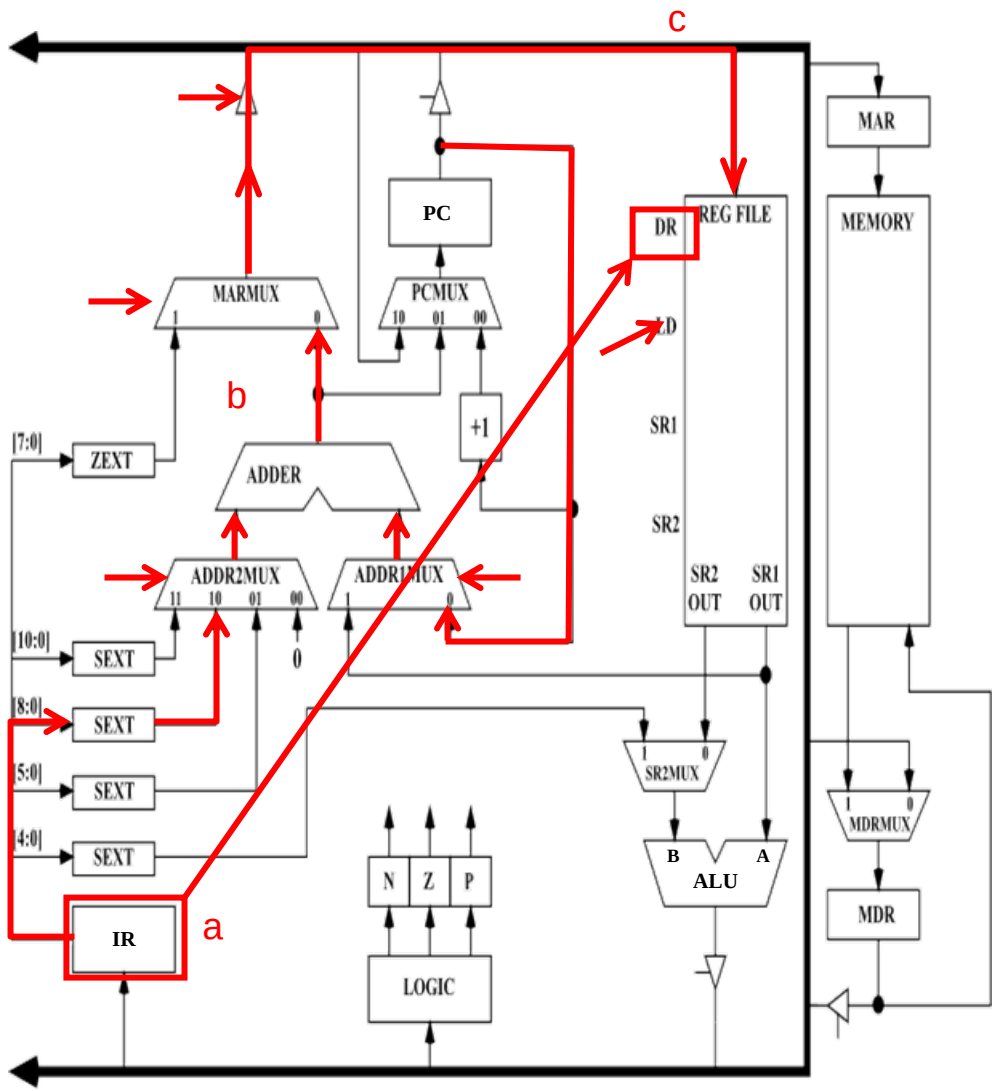
The Decoder


IR[15:12]
4:16
Decoder
0000
BR
0001
ADD
0010
LD
0011
ST
0100
JSR
0101
AND
0110
LDR
0111
STR
1000
RTI
1001
NOT
1010
LDI
1011
STI
1100
JMP
1101
reserved
1110
LEA
1111
TRAP

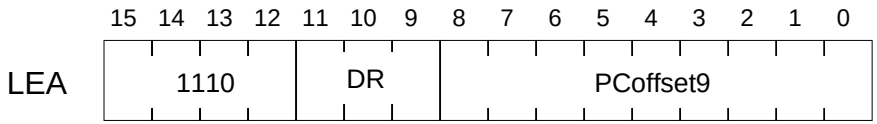
LEA Instruction



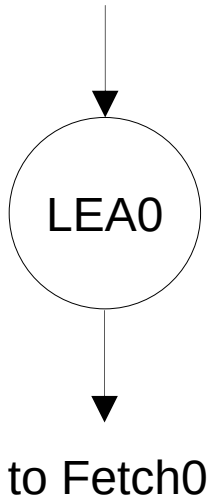
- Send DR field from IR as address to the register file (a)
- Add the contents of the PC to the zero extended PCoffset9 from the IR to form the effective address (b)
- Store the generated address into the DR (c)



The LEA Instruction

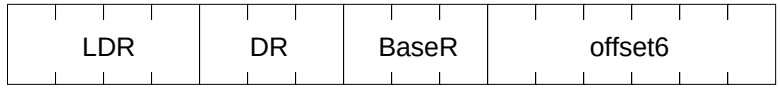


- R[DR] ← PC + IR[8:0]
- Note that the PC Offset is always a 2's complement (signed) value

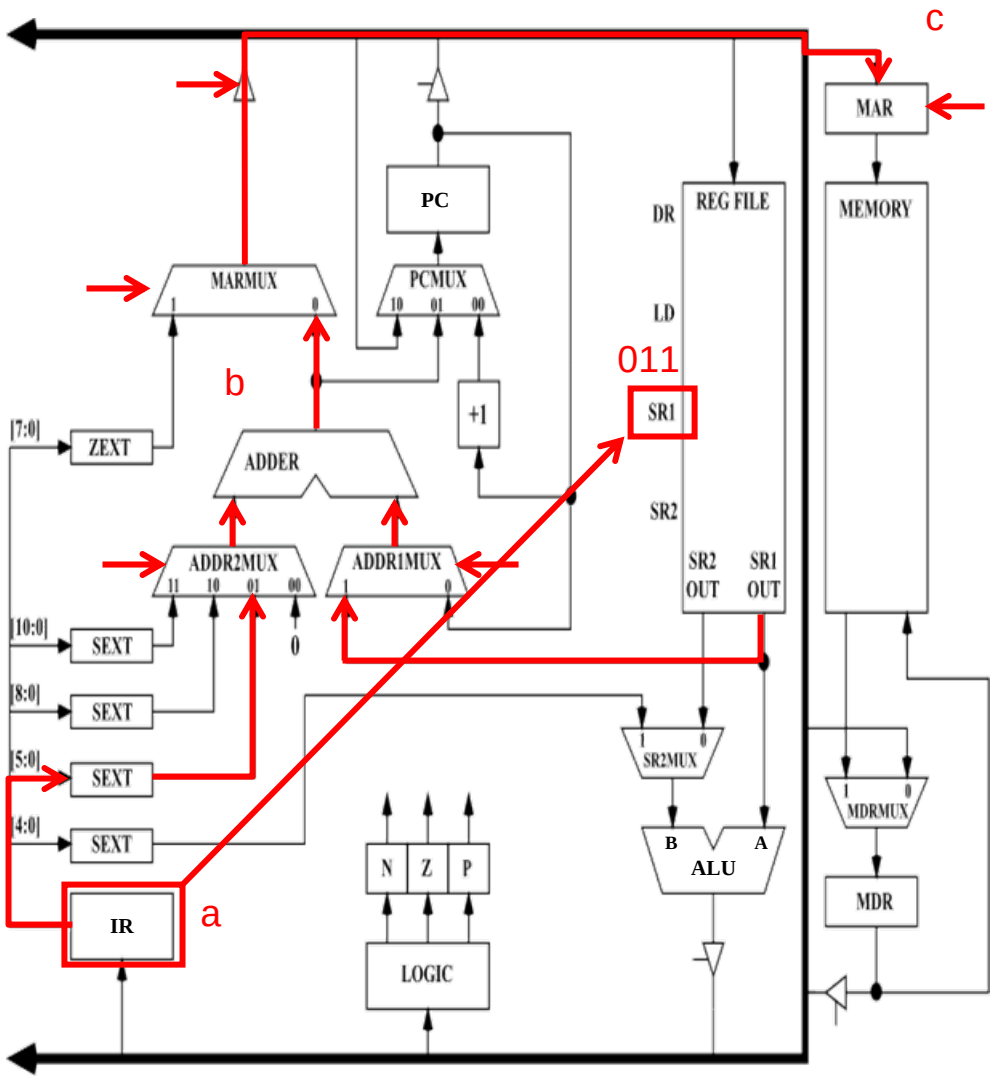


selEAB1 <= '0'
selEAB2 <= "10"
selMAR <= '0'
enaMARM
DR <= IR[11:9]
regWE

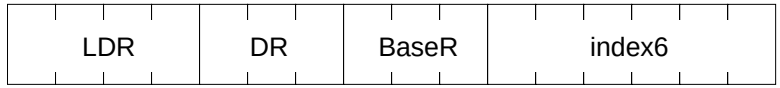
LDR Instruction



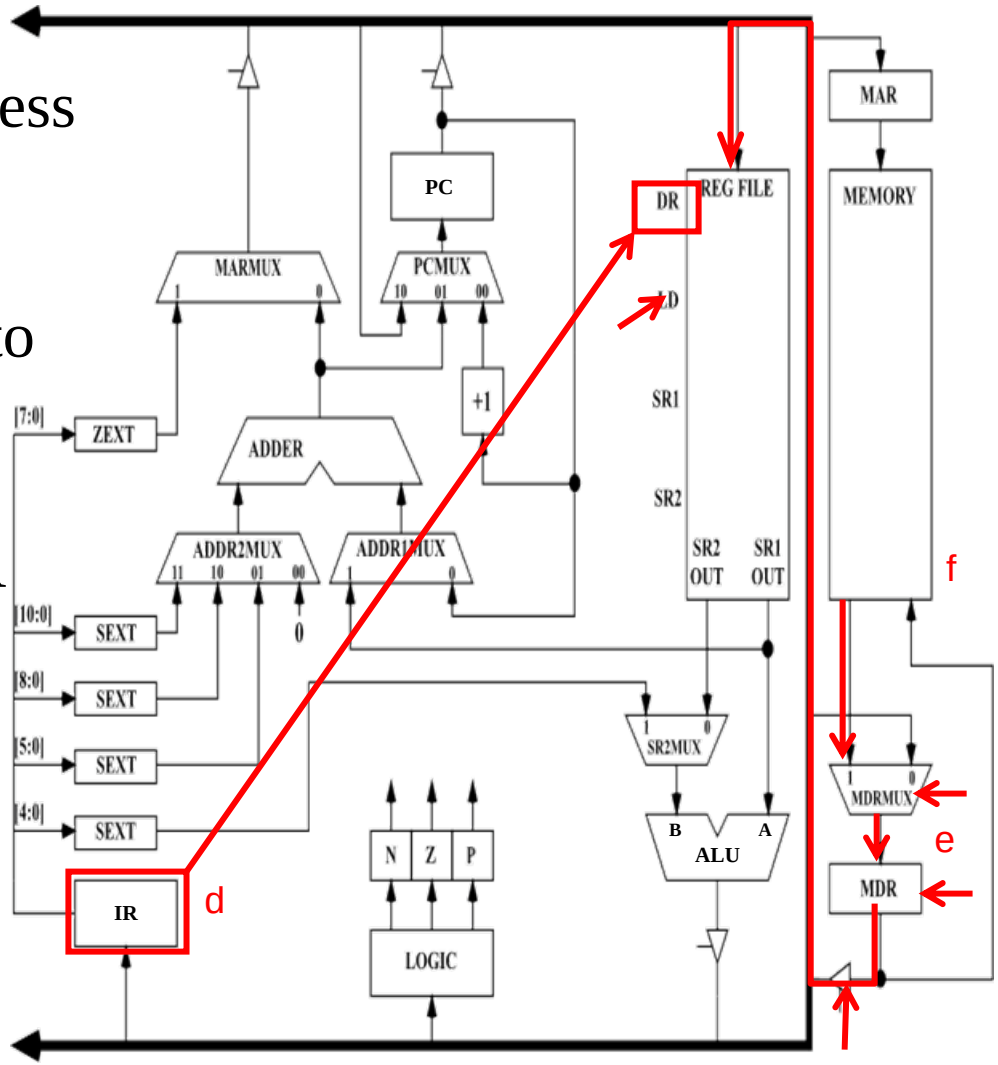
- Send BaseR field from IR as address to the register file (a)
- Add the contents of BaseR to the zero extended offset6 from the IR to form the destination memory address for the STR (b)
- Store the generated address into the MAR (c)



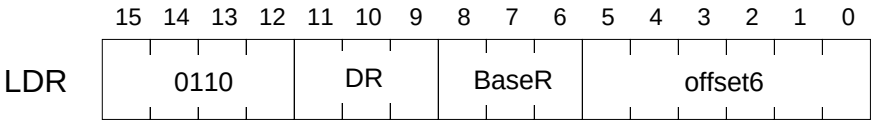
LDR Instruction



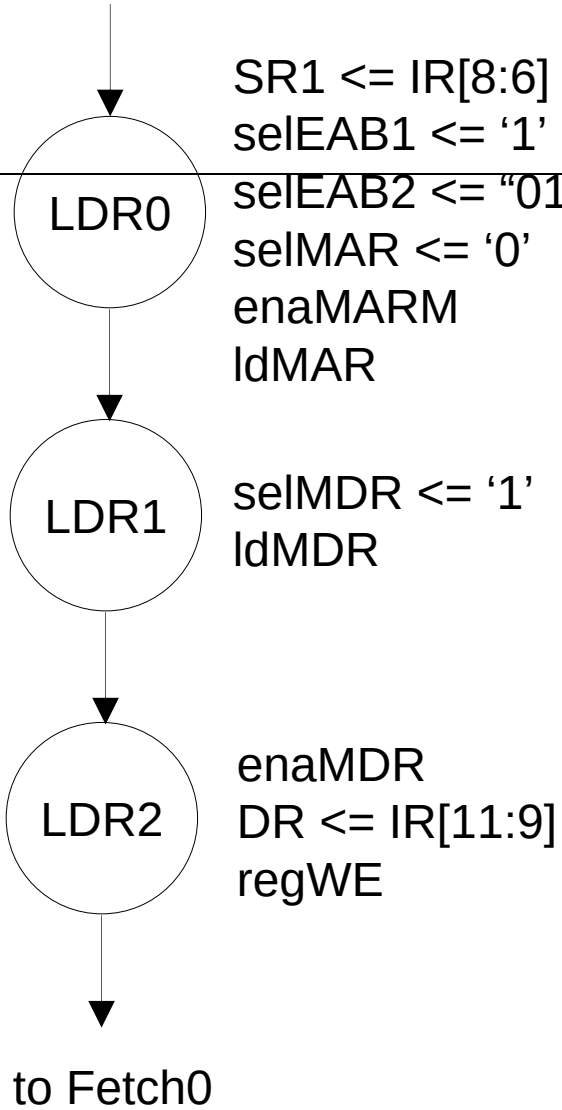
- Send DR field from IR as address to the register file (d)
- Store the contents of memory to the MDR (e)
- Write the contents of the MDR into the DR (f)



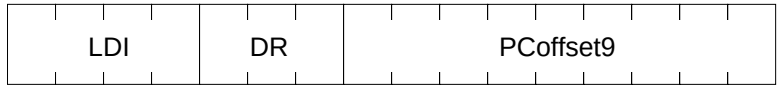
The LDR Instruction



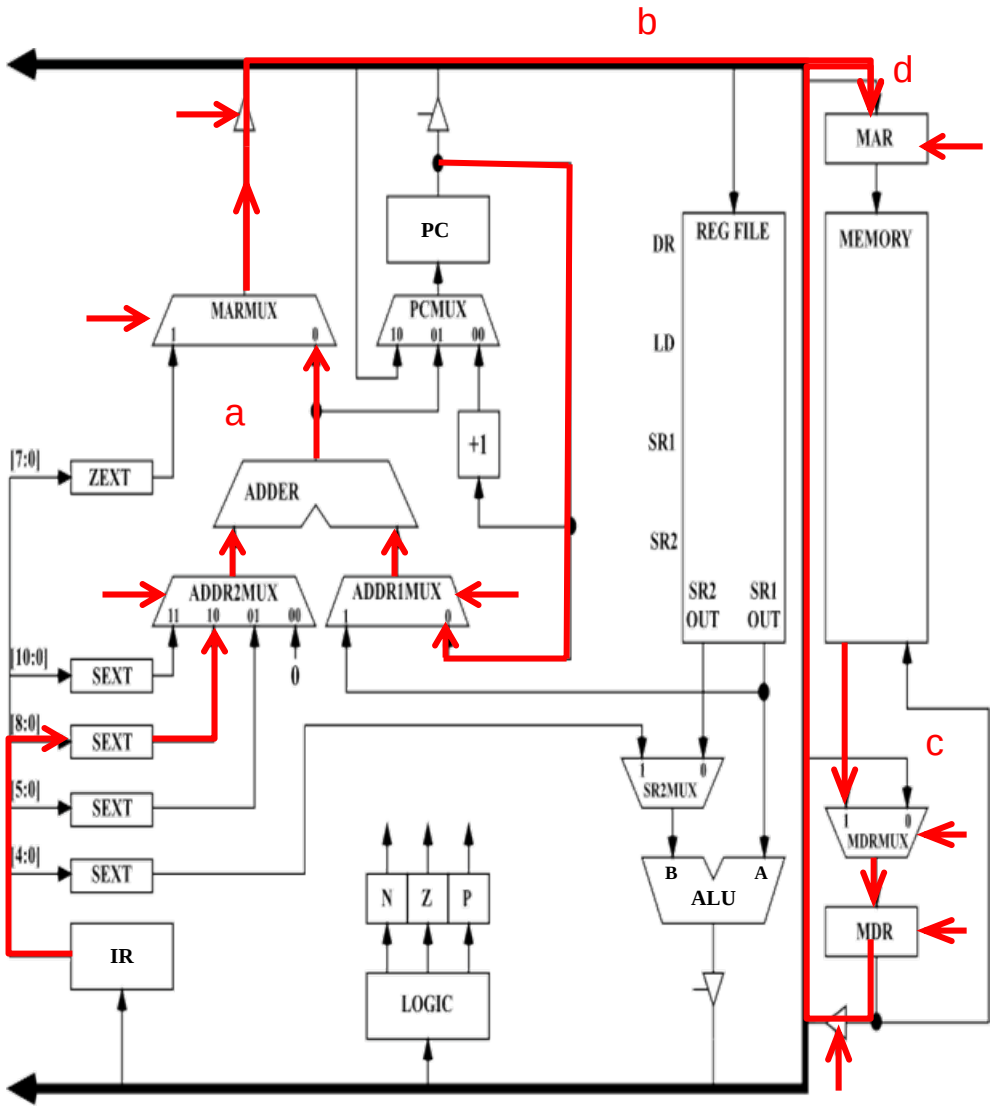
- MAR ← R[BaseR]+offset6
- MDR ← Mem[MAR]
- R[DR] ← MDR



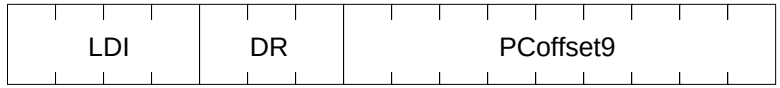
LDI Instruction



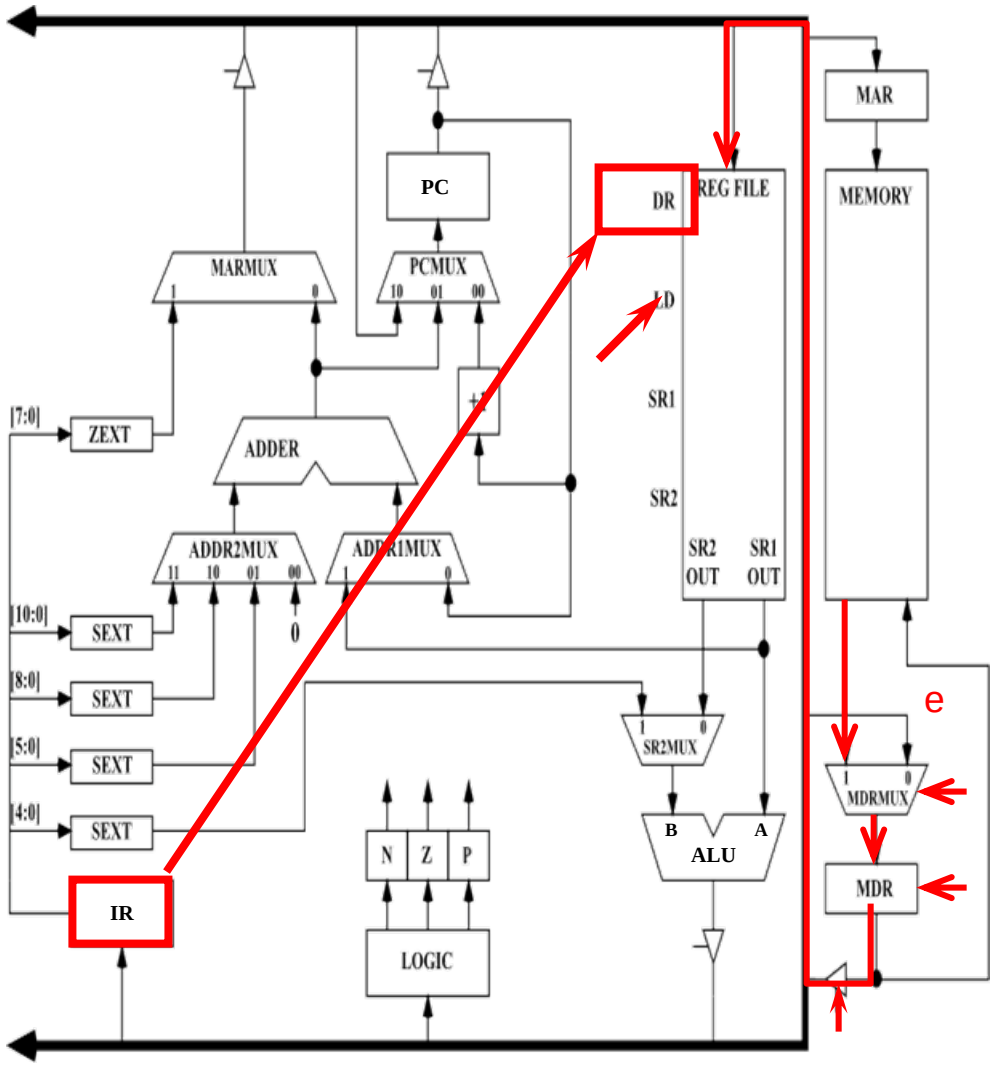
- Add the contents of the PC to the sign extended PCoffset9 from the IR to form the source memory address for the LDI (a)
- Store the generated address into the MAR (b)
- Load the memory contents into the MDR (c)
- Load the contents of the MDR into the MAR (d)



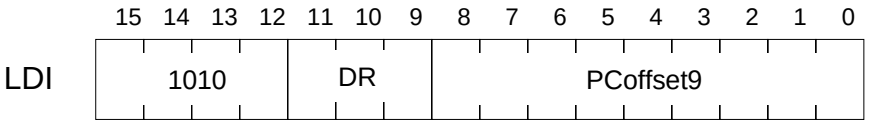
LDI Instruction



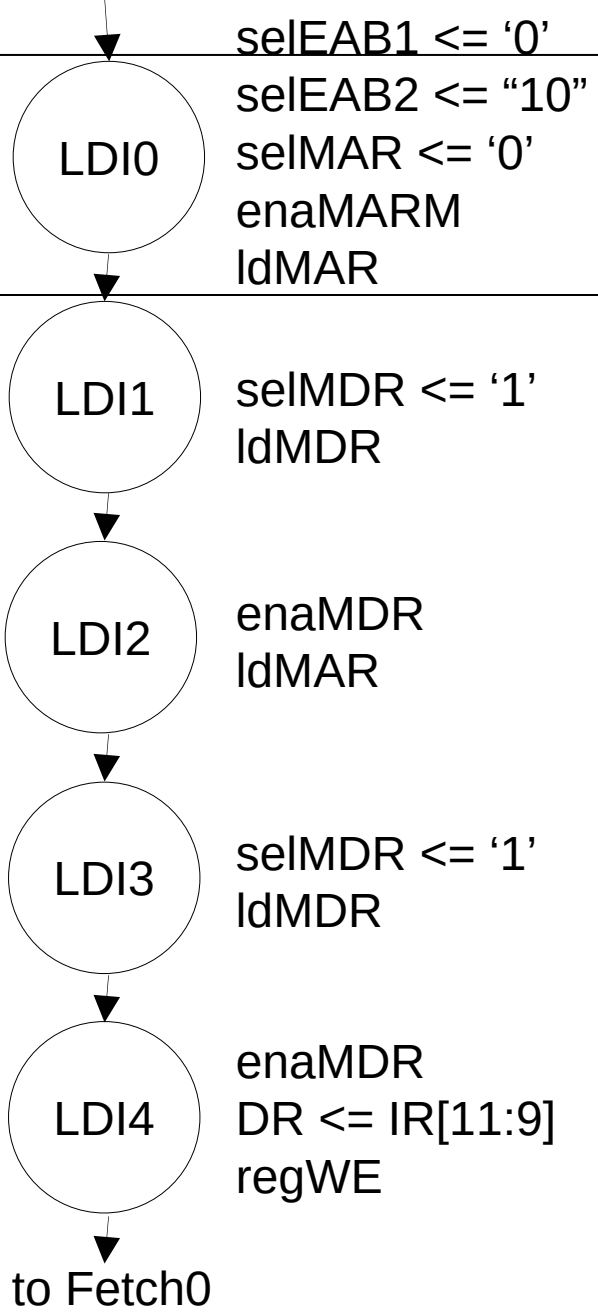
- Load the memory contents into the MDR (e)
- Load the contents of the MDR into the DR (f)



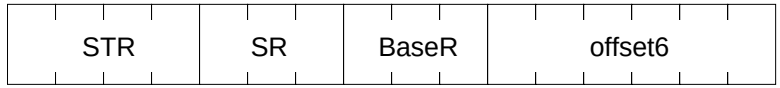
The LDI Instruction



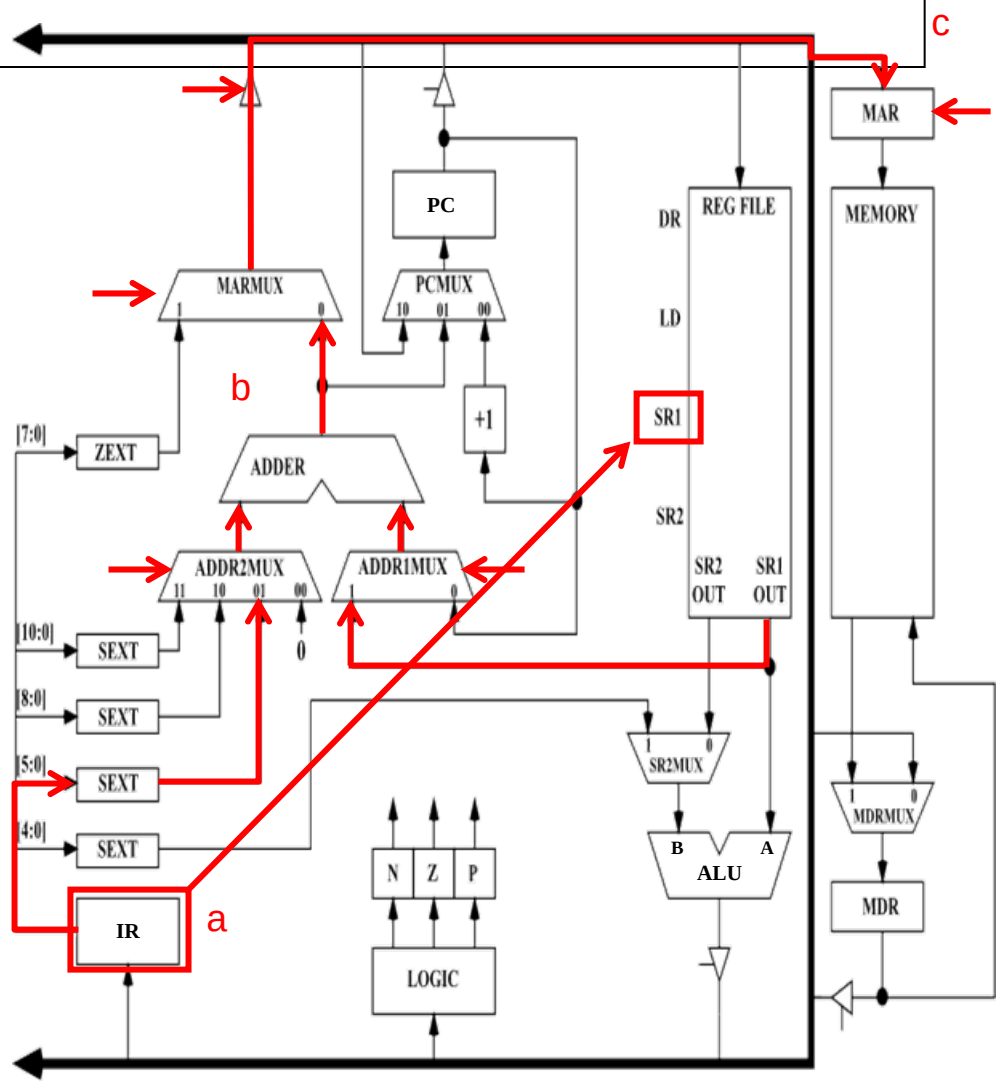
- MAR ← PC + IR[8:0]
- MDR ← Mem[MAR]
- MAR ← MDR
- MDR ← Mem[MAR]
- R[DR] ← MDR



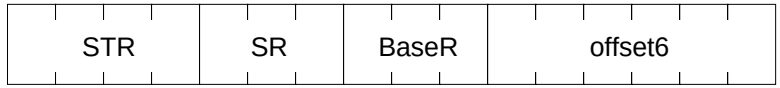
STR Instruction



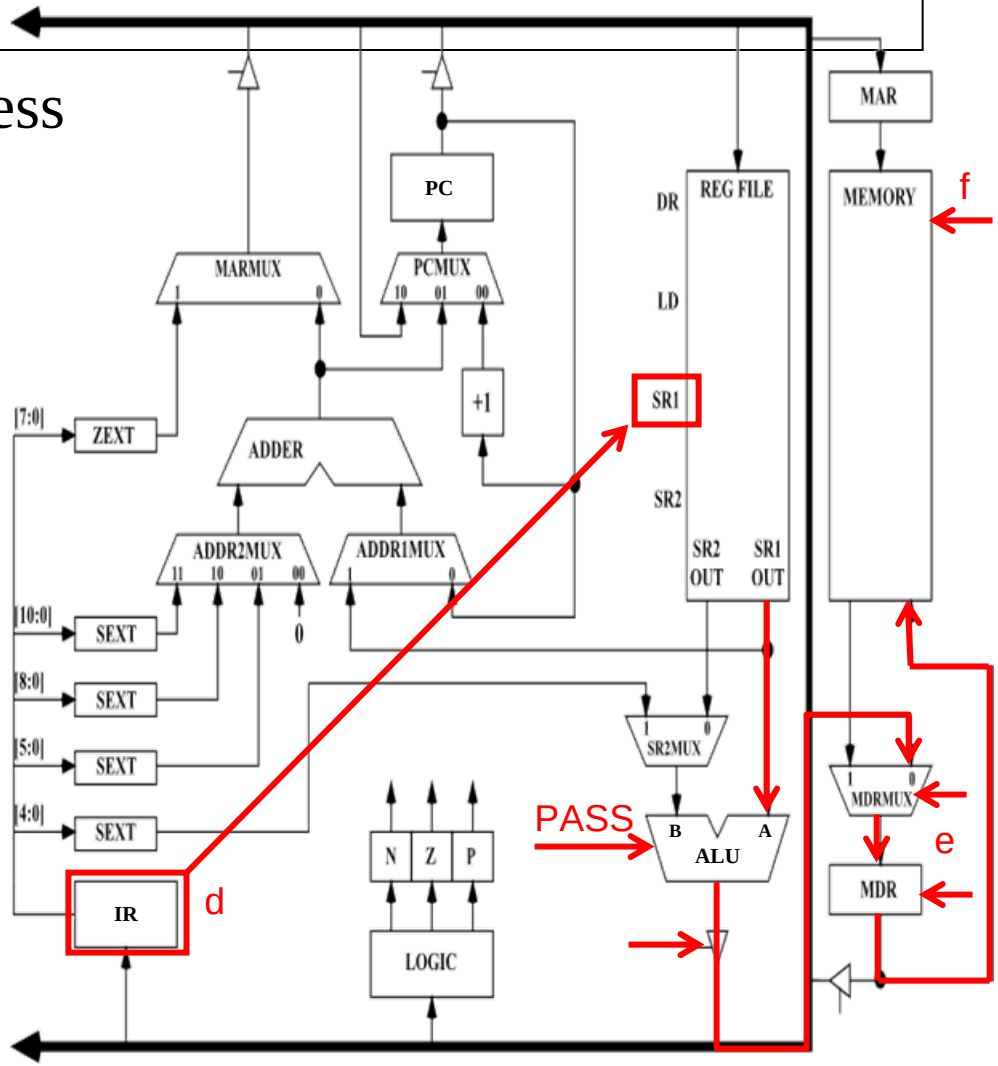
- Send BaseR field from IR as address to the register file (a)
- Add the contents of BaseR to the zero extended offset6 from the IR to form the destination memory address for the STR (b)
- Store the generated address into the MAR (c)



STR Instruction

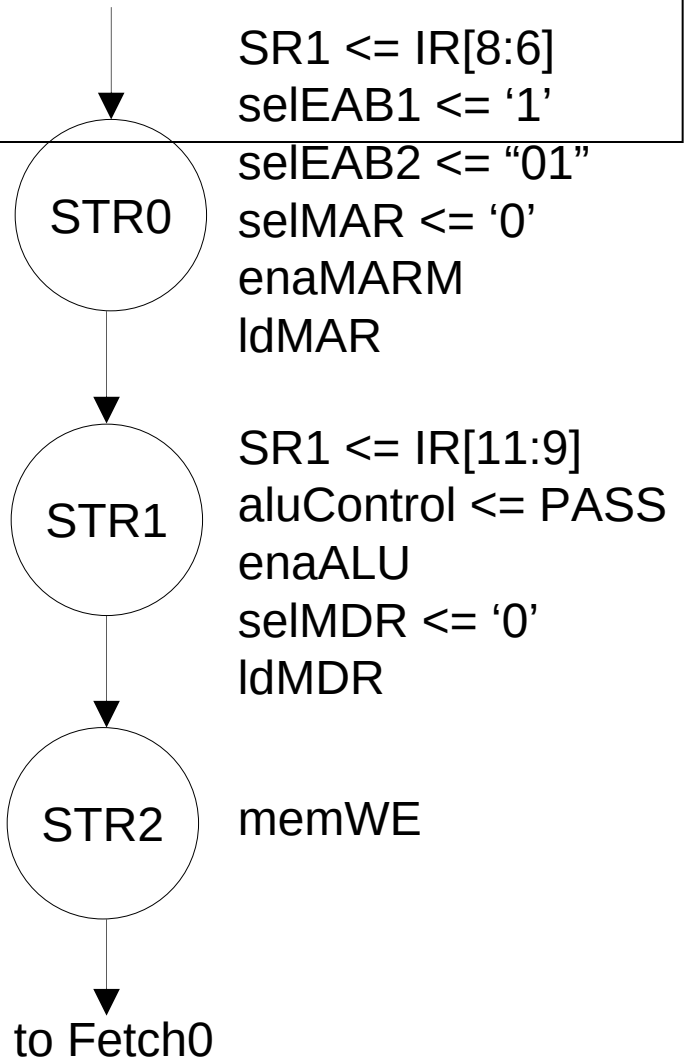


- Send SR field from IR as address to the register file (d)
- Store the contents of SR to the MDR (e)
- Perform the memory write (f)

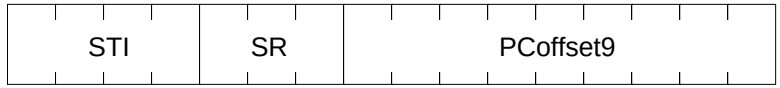


The STR Instruction

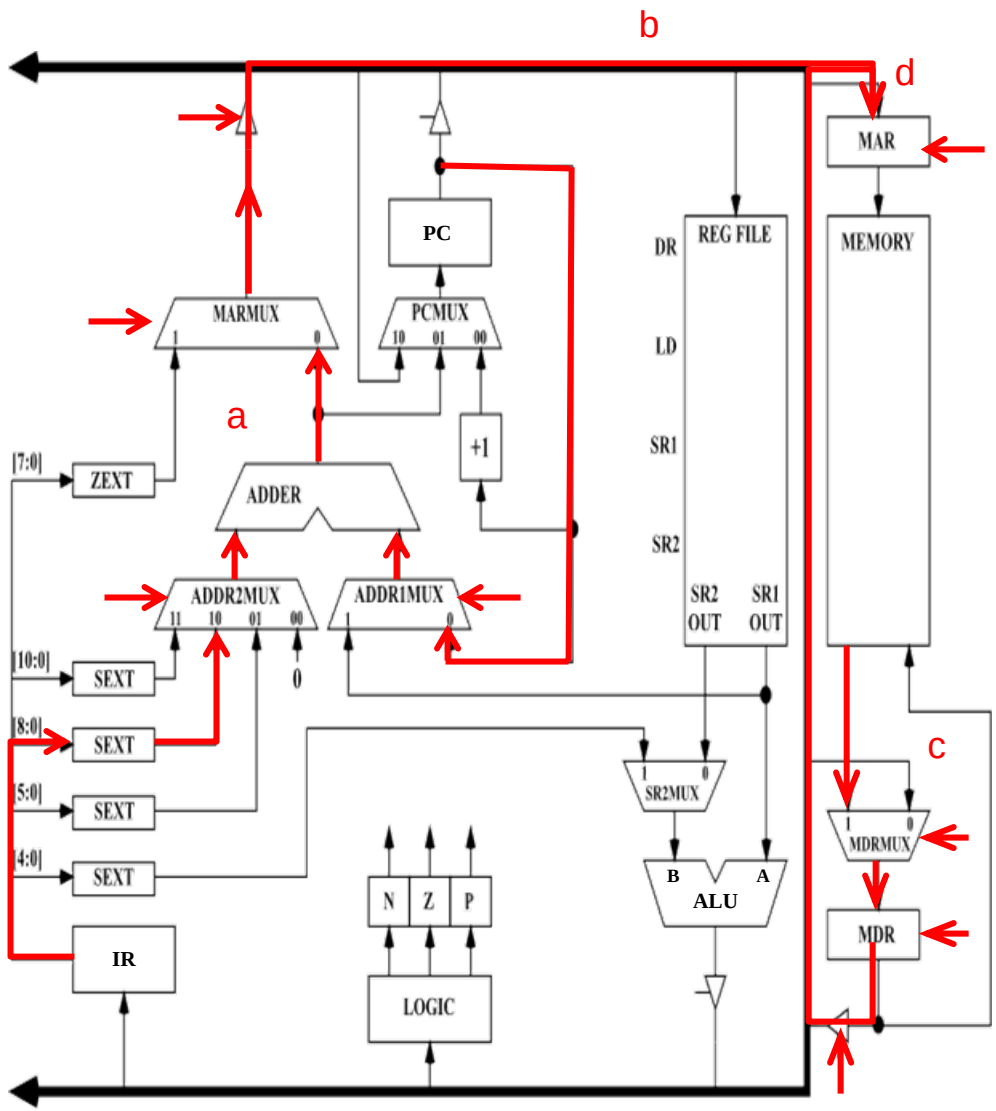
- MAR ← R[BaseR]+offset
- MDR ← R[SR]
- Write memory



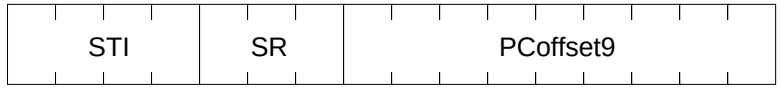
STI Instruction



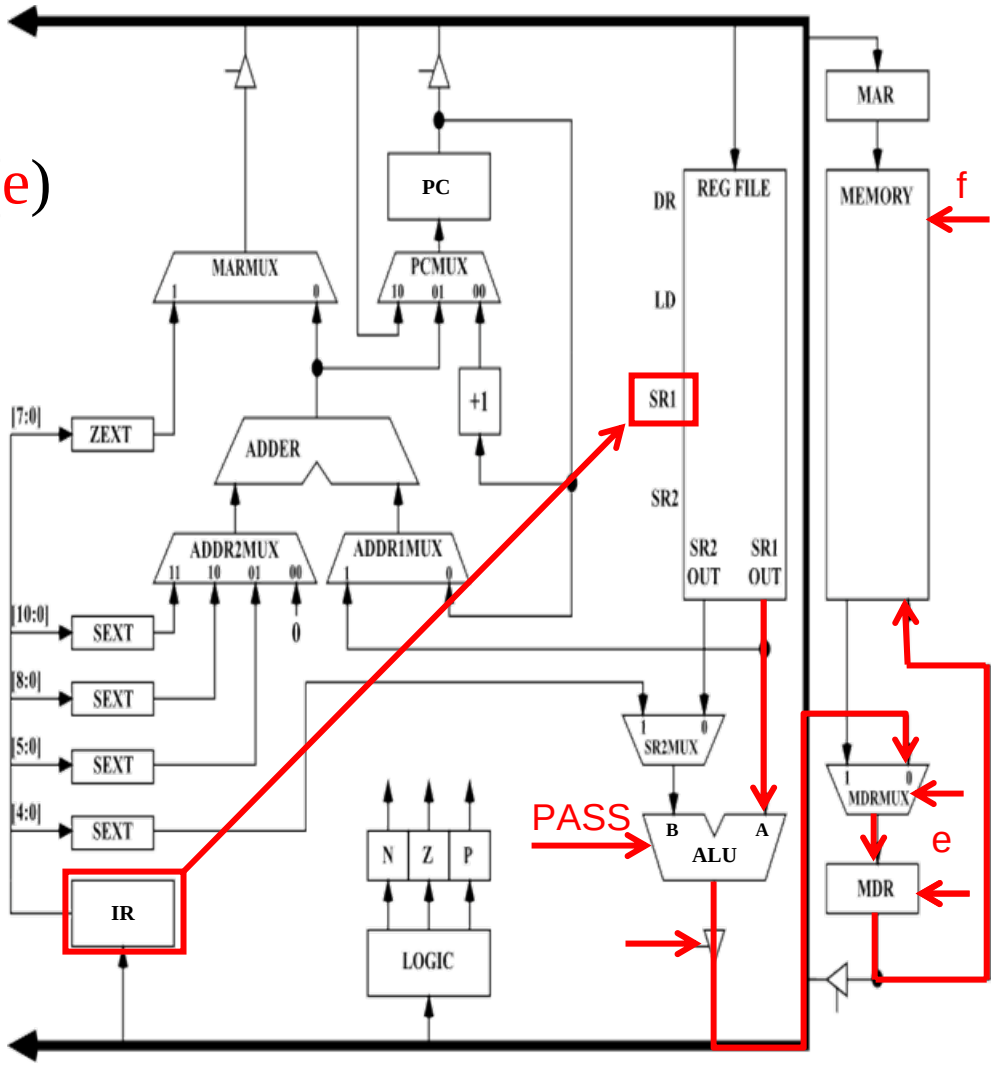
- Add the contents of the PC to the sign extended PCoffset9 from the IR to form the source memory address for the STI (a)
- Store the generated address into the MAR (b)
- Load the memory contents into the MDR (c)
- Load the contents of the MDR into the MAR (d)



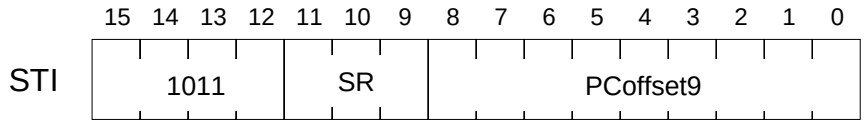
STI Instruction



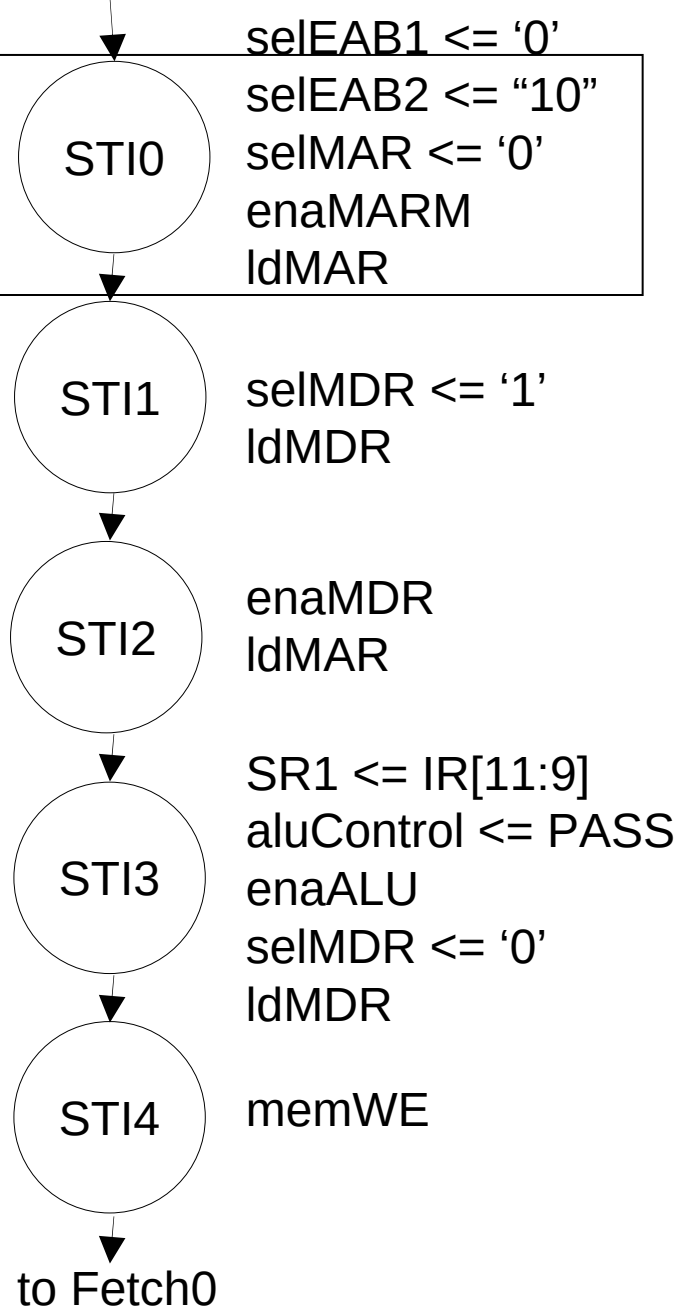
- Load the contents of the source register into the MDR (e)
- Load the contents of the MDR into the DR (f)



The STI Instruction



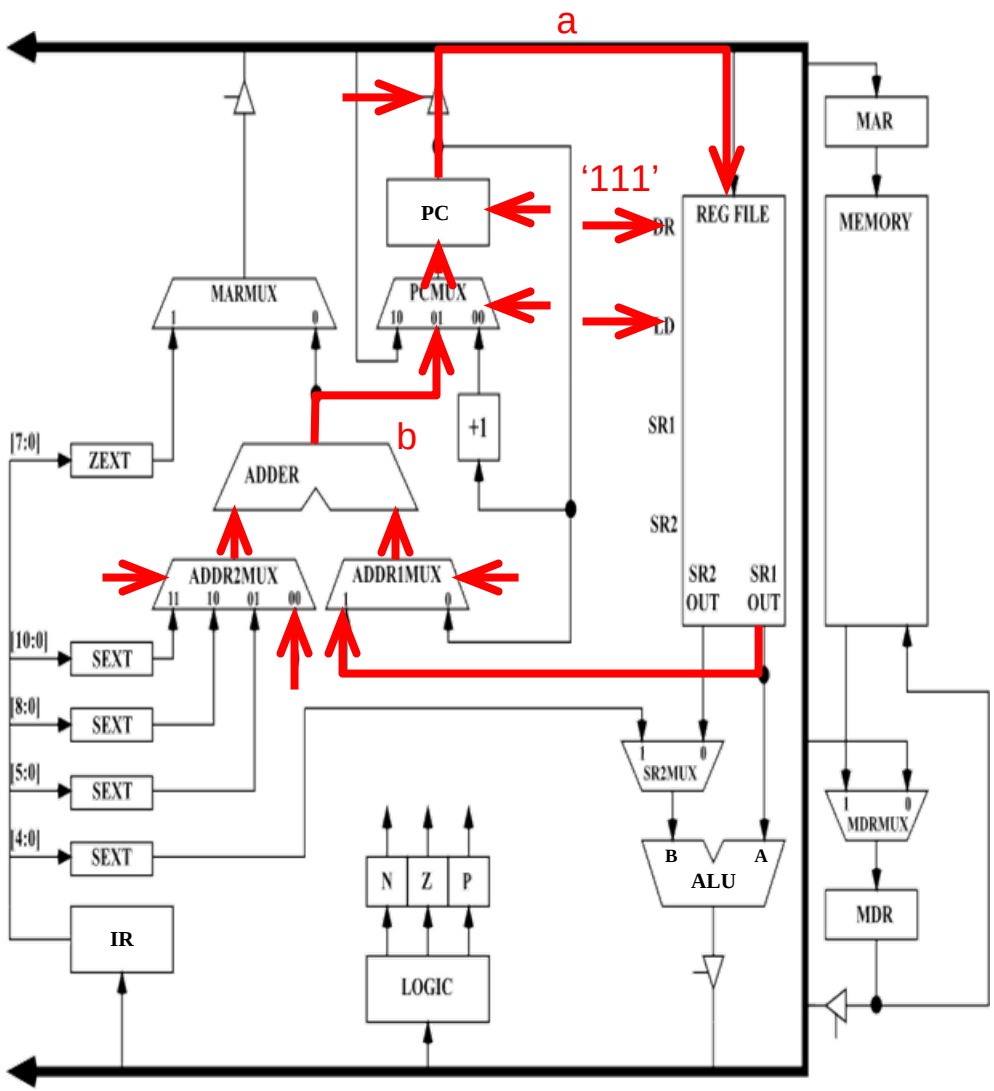
- MAR ← PC + IR[8:0]
- MDR ← M[MAR]
- MAR ← MDR
- MDR ← R[SR]
- Write memory



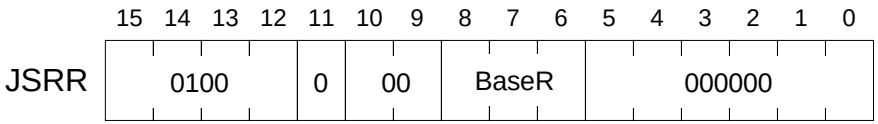
JSRR Instruction

JSRR	0	00	BaseR	000000

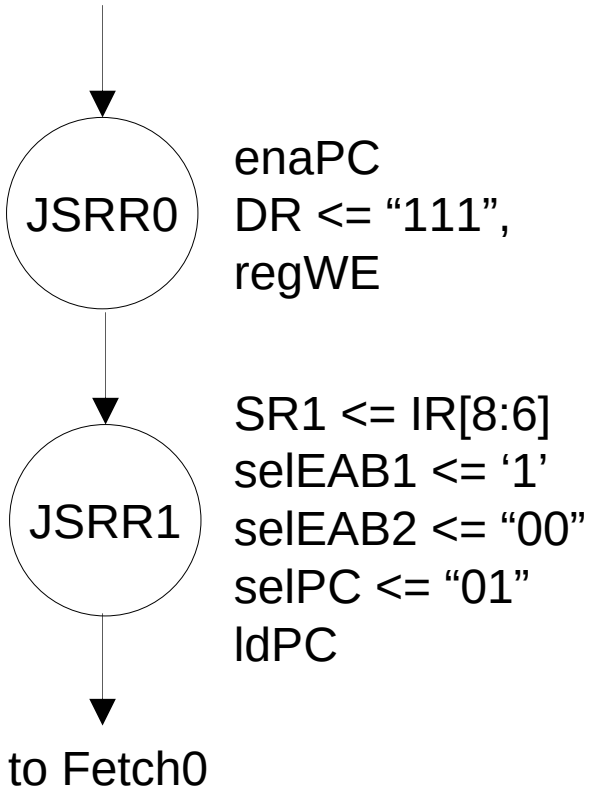
- Load the contents of the PC into R7 (a)
- Load the contents of the base address register into the PC (b)



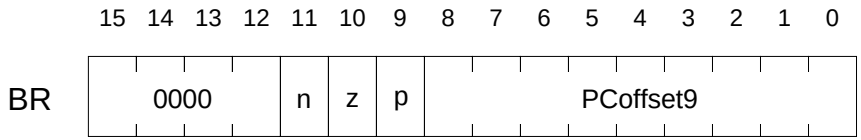
The JSRR Instruction



- Note: Same opcode as JSR! (determined by IR bit 11)
- R7 ← PC
- PC ← R[BaseR]



The BR Instruction



- Simply check NZP flags with nzp from instruction to decide whether to ldPC or not
- Method 1: Could bring nzp flags into FSM as inputs and put comparison into state table… Bad idea
- Method 2: Do comparison with some external gates and bring single-bit input into FSM….Good idea

The BR Instruction