Advanced Lab 3

Memory, LC3 State Machine and Pipelining

Andrew Wilder

Topics

- Memory
- LC3 State Machine
  - Summary of State Machine
  - Instruction Overview
- Pipelining
  - Throughput
  - Hazards

Memory

Question: The LC3 register file is composed of an array of 16-bit registers, made of D flip-flops. Would this work for memory?

Memory

Question: The LC3 register file is composed of an array of 16-bit registers, made of D flip-flops. Would this work for memory?

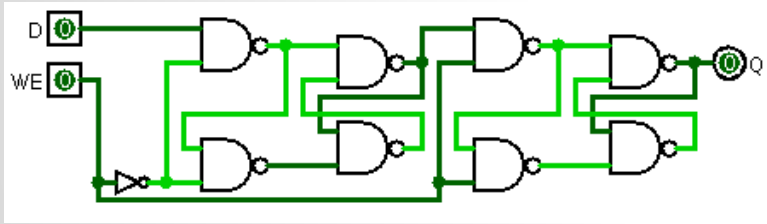
No:

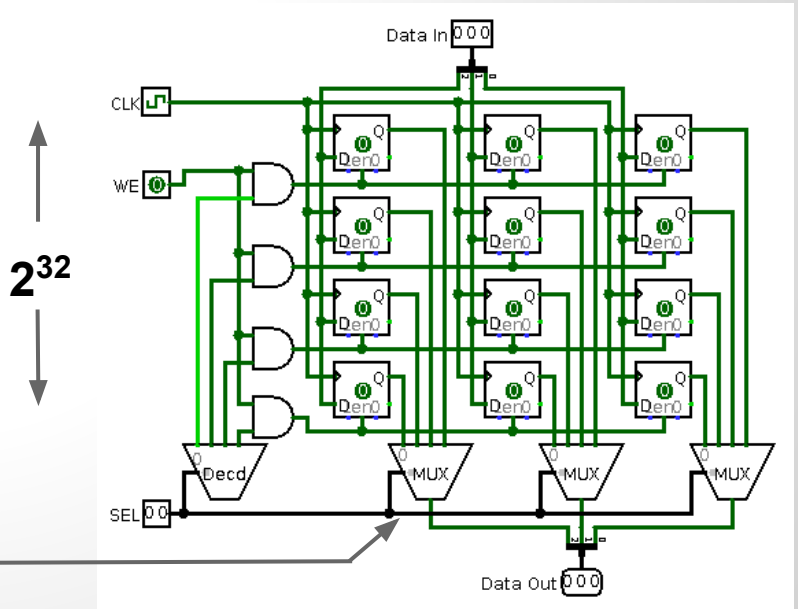
- Too many resources
- Too much power
- Too complex to manufacture

Memory

Consider a 4GB RAM chip. What if this was composed of a D flip-flop array?

34 transistors

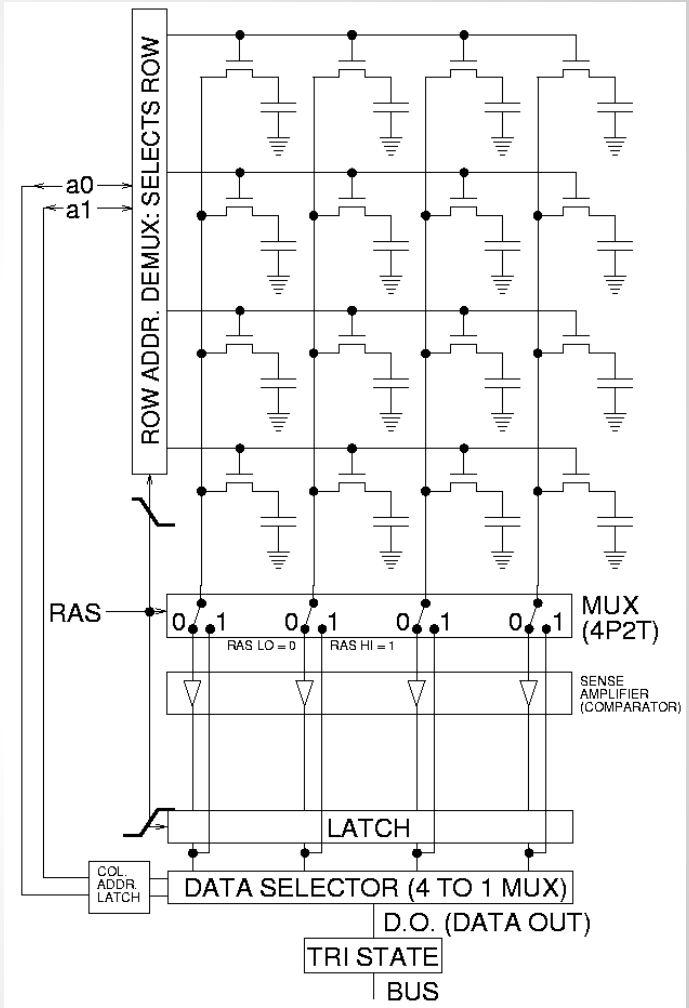




Billions of AND gates and OR gates

Memory

Solution: One transistor and capacitor per bit, constructed in rows and columns. Easier to manufacture, lower power consumption.



Memory

Much like the homework, DRAM is manufactured in smaller pieces on the order of kilobytes, which are combined as abstractions to create larger memories; however, its design takes multiple clock cycles to access.

LC3 State Machine

The LC3 State Machine has very specific standards outlined in appendix C of the textbook. In the following slides, we will cover the execution of each instruction on the datapath.

LC3 State Machine

Inputs:

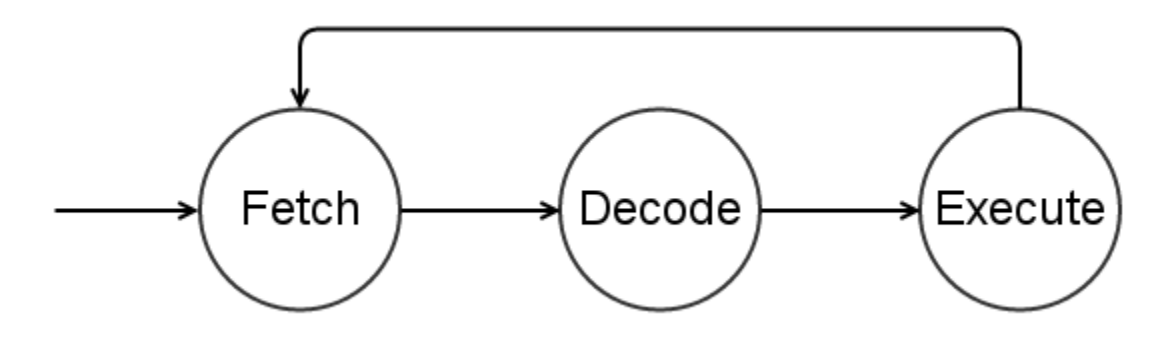
- S0 - S5 (current state bits)
- IR[15:12] (opcode)
- IR[11:9] (hardcoded CC)
- NZP (variable CC)
- IR[5] (imm5 ADD, AND, NOT)
- IR[11] (JSR vs JSRR)

Outputs:

- LD PC,IR,REG,MAR,MDR,CC (6)
- Tri-state buffers (4)
- 5 Multiplexers (7 bits)
- Mem EN/WE (2 bits)
- DR, SR1, SR2 (9 bits)
- ALUK (2 bits)

LC3 State Machine

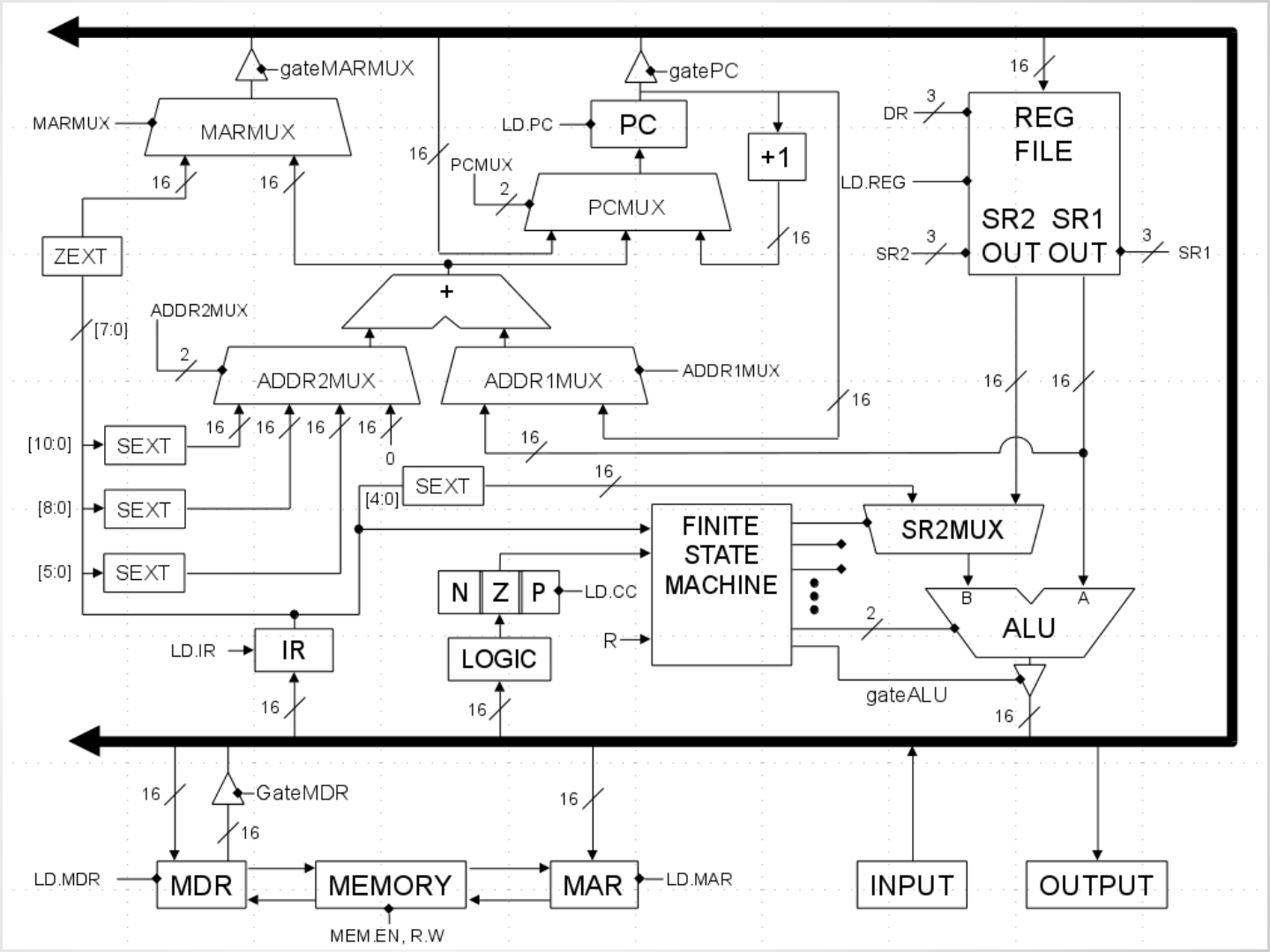
Basic sequence of events: Fetch instruction, decode instruction, execute instruction, repeat.



Fetch

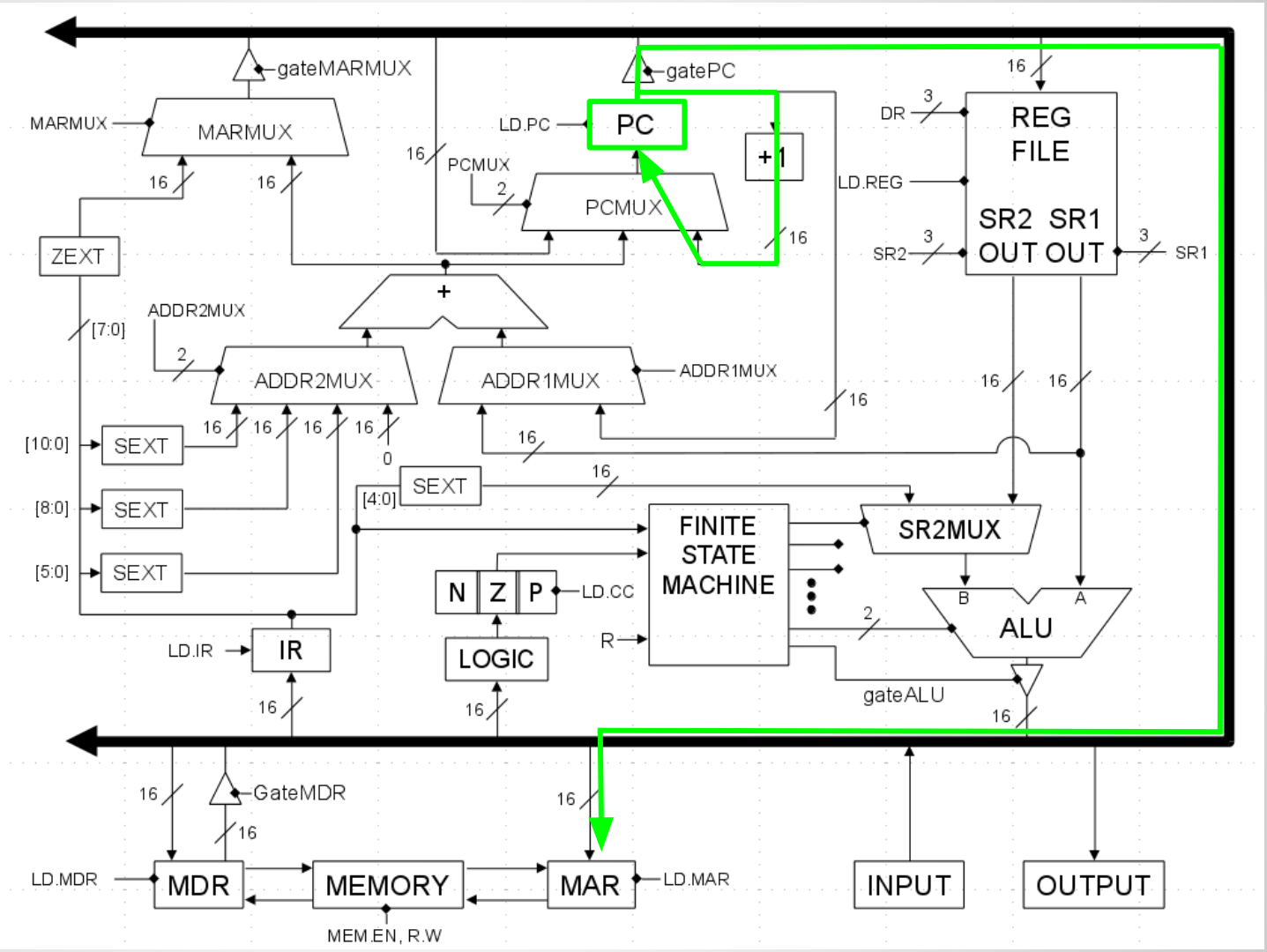
The purpose of fetch is to obtain the next instruction to be executed. This involves going to the memory location pointed to by PC, retrieving the value there, and storing it in IR.

Fetch


gateMARMUX
MARMUX
MARMUX
gatePC
LD.PC
PC
+1
PCMUX
PCMUX
16
2
16
16
16
16
DR
3
REG
FILE
LD.REG
SR2 SR1
OUT OUT
SR2
3
3
SR1
ZEXT
[7:0]
+
ADDR2MUX
2
ADDR2MUX
ADDR1MUX
ADDR1MUX
16
16
16
[10:0]
SEXT
16
16
16
16
0
[8:0]
SEXT
[5:0]
SEXT
SEXT
[4:0]
16
FINITE
STATE
MACHINE
SR2MUX
N
Z
P
LD.CC
B
A
2
ALU
LD.IR
IR
LOGIC
R
gateALU
16
16
16
GateMDR
16
16
16
LD.MDR
MDR
MEMORY
MAR
LD.MAR
INPUT
OUTPUT
MEM.EN, R.W

Fetch

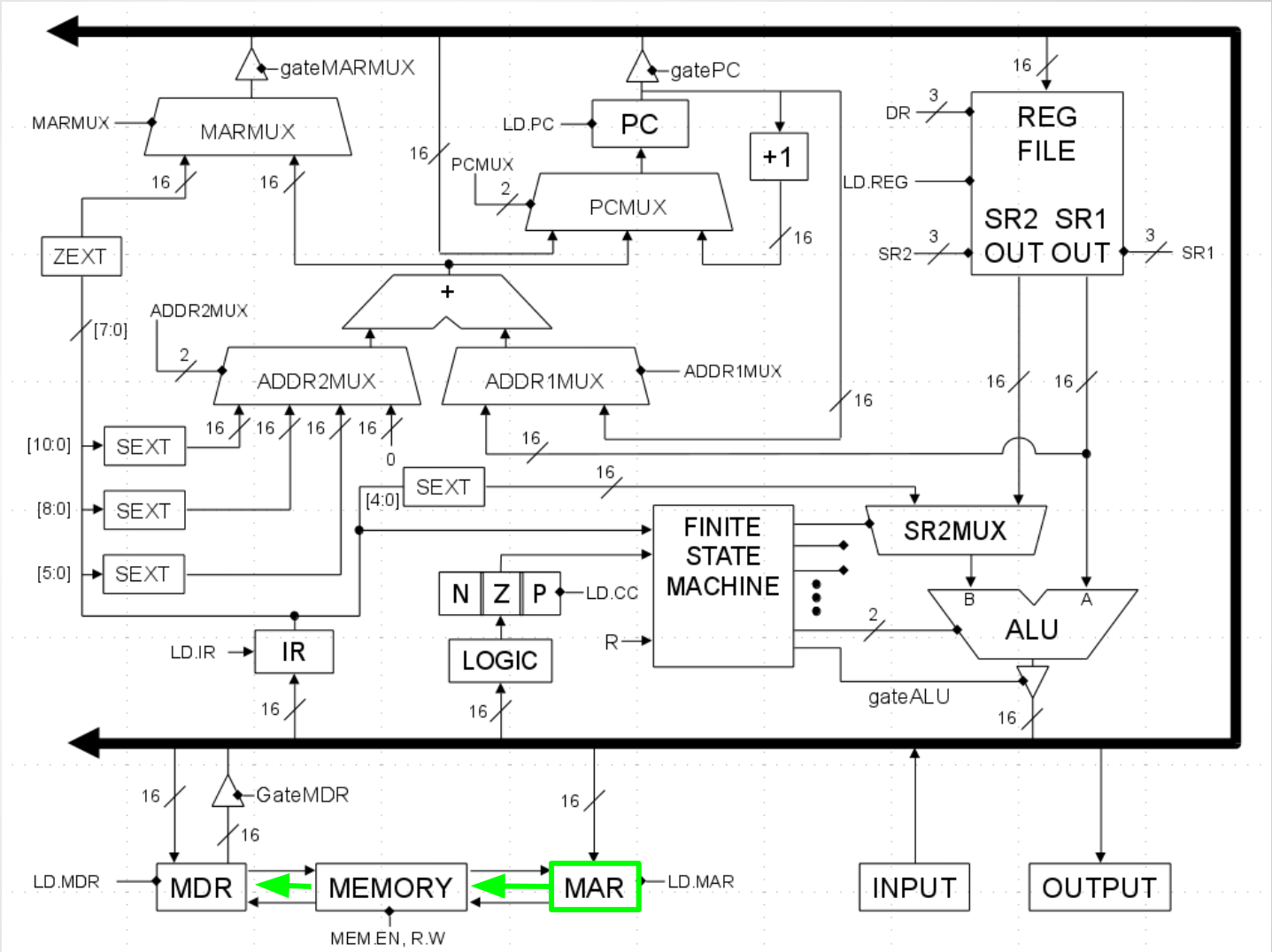
CC1:
gatePC
LD.MAR
PCMUX=PC+1
LD.PC



Fetch

CC1:
gatePC
LD.MAR
PCMUX=PC+1
LD.PC

CC2:
MEM.EN
LD.MDR

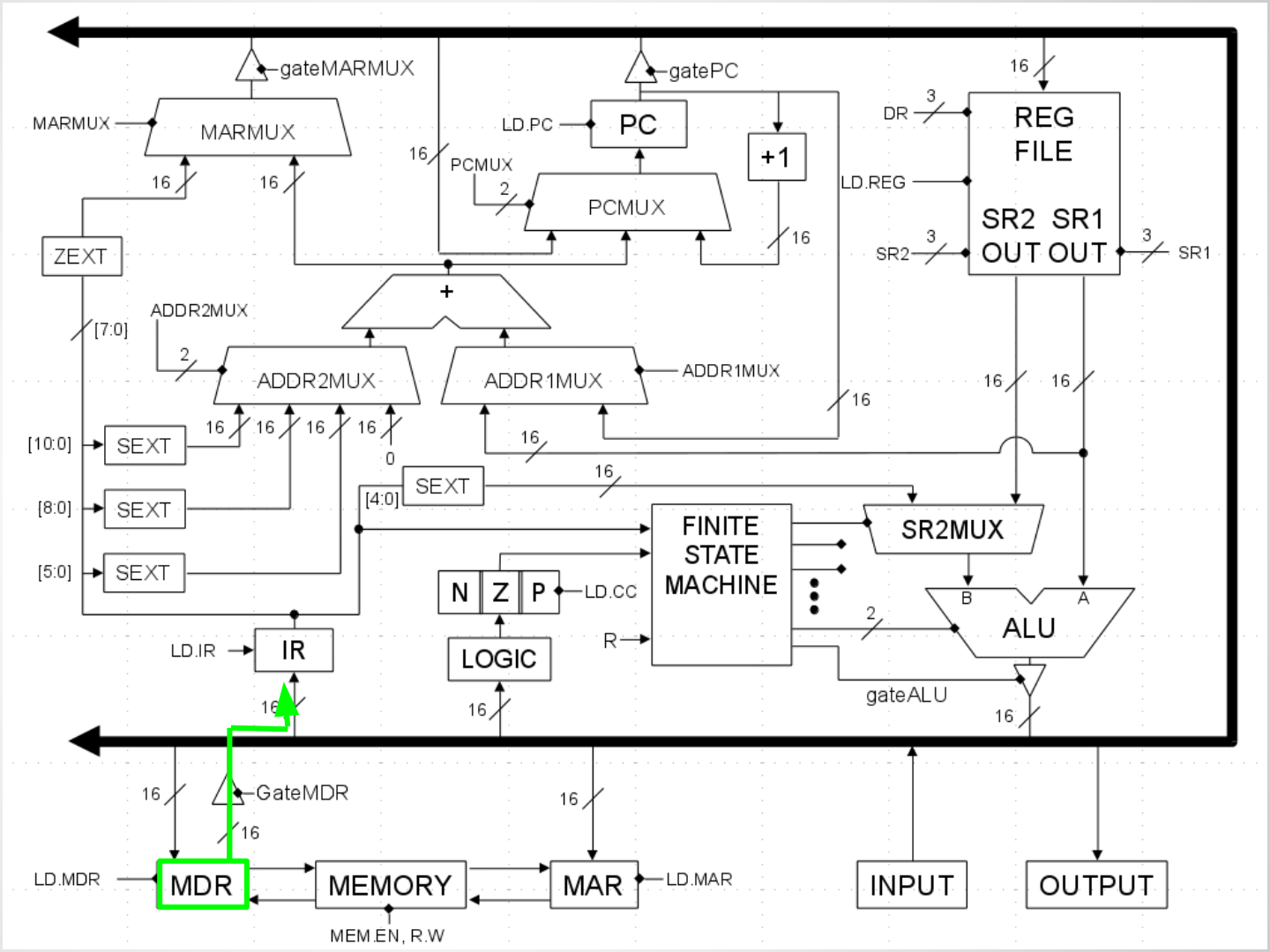


Fetch

CC1:
gatePC
LD.MAR
PCMUX=PC+1
LD.PC

CC2:
MEM.EN
LD.MDR

CC3:
gateMDR
LD.IR

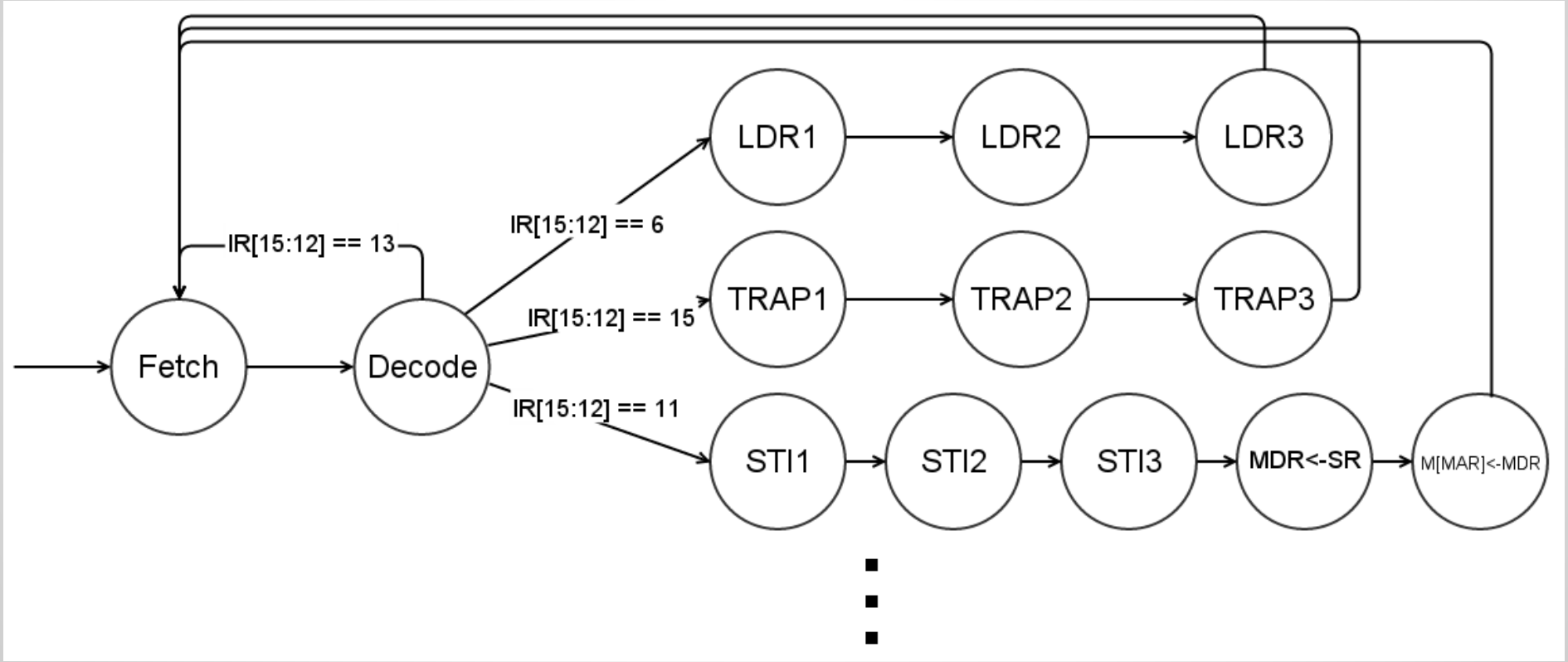


Decode

In the decode stage, the opcode bits are read, and determine the sequence of states to execute next.

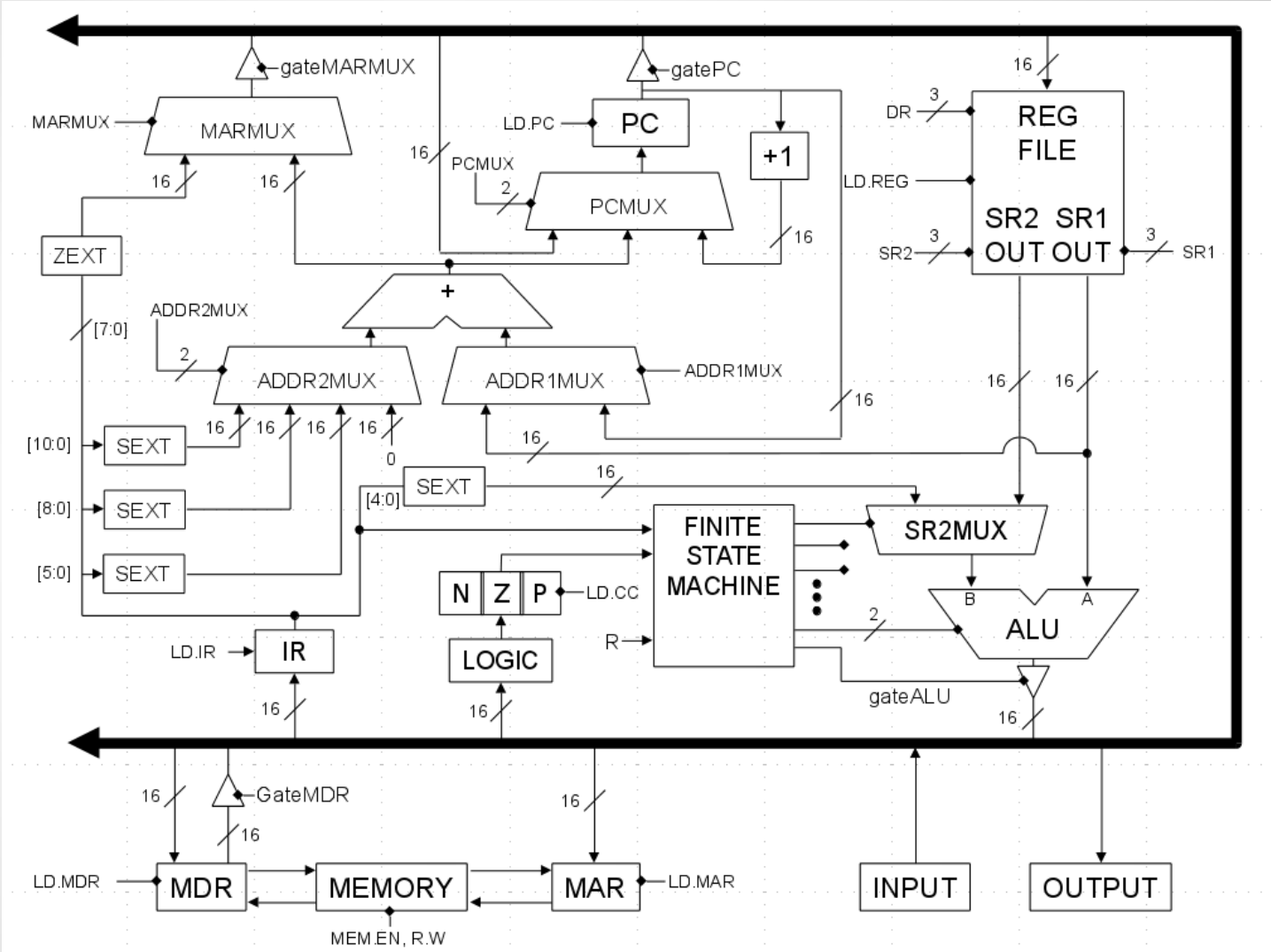
Note: The RTI instruction will be omitted from these slides. Appendix C in the textbook has more information on interrupt support, and interrupts will be covered in more detail in CS 2200.

Decode


Fetch
Decode
IR[15:12] == 13
IR[15:12] == 6
IR[15:12] == 15
IR[15:12] == 11
LDR1
LDR2
LDR3
TRAP1
TRAP2
TRAP3
STI1
STI2
STI3
MDR<-SR
M[MAR]<-MDR

Decode

No signals, because the decode events take place within the state machine.



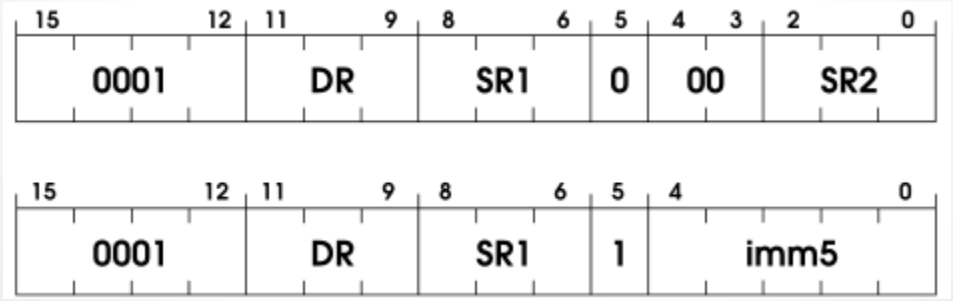
Execute

Each instruction has its own series of states with specific outputs enabled to control the flow of data in the LC3. The 3 general types of instructions are:

- Operations
  - ADD, AND, NOT
- Data transfer
  - LD, LDR, LDI, ST, STR, STI, LEA
- Program control
  - JMP/RET, JSR/JSRR, BR, TRAP

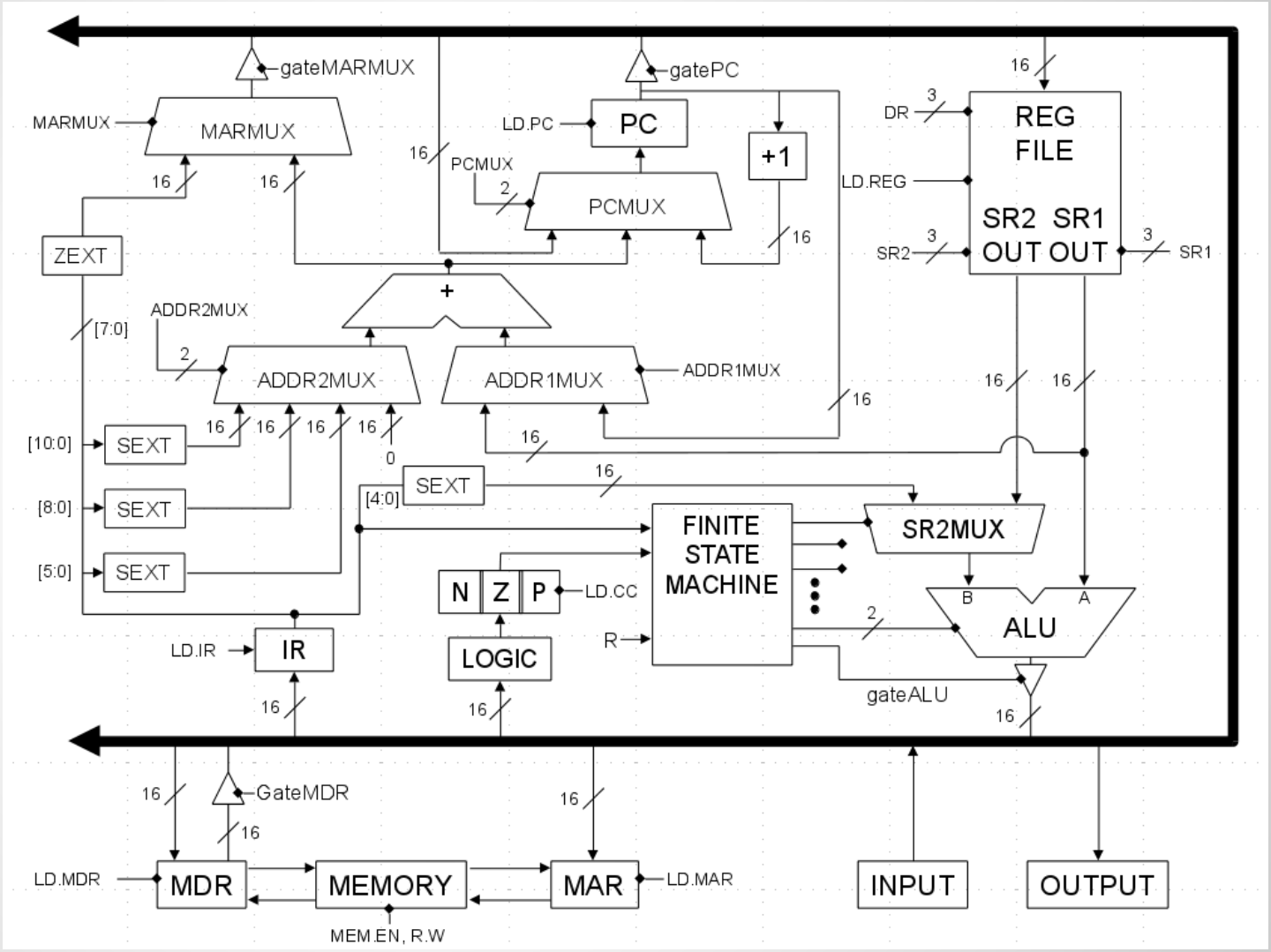
ADD

The ADD instruction adds the contents of a register and a second operand, and stores the result in a register. The second operand may be another register, or an immediate 5-bit value.



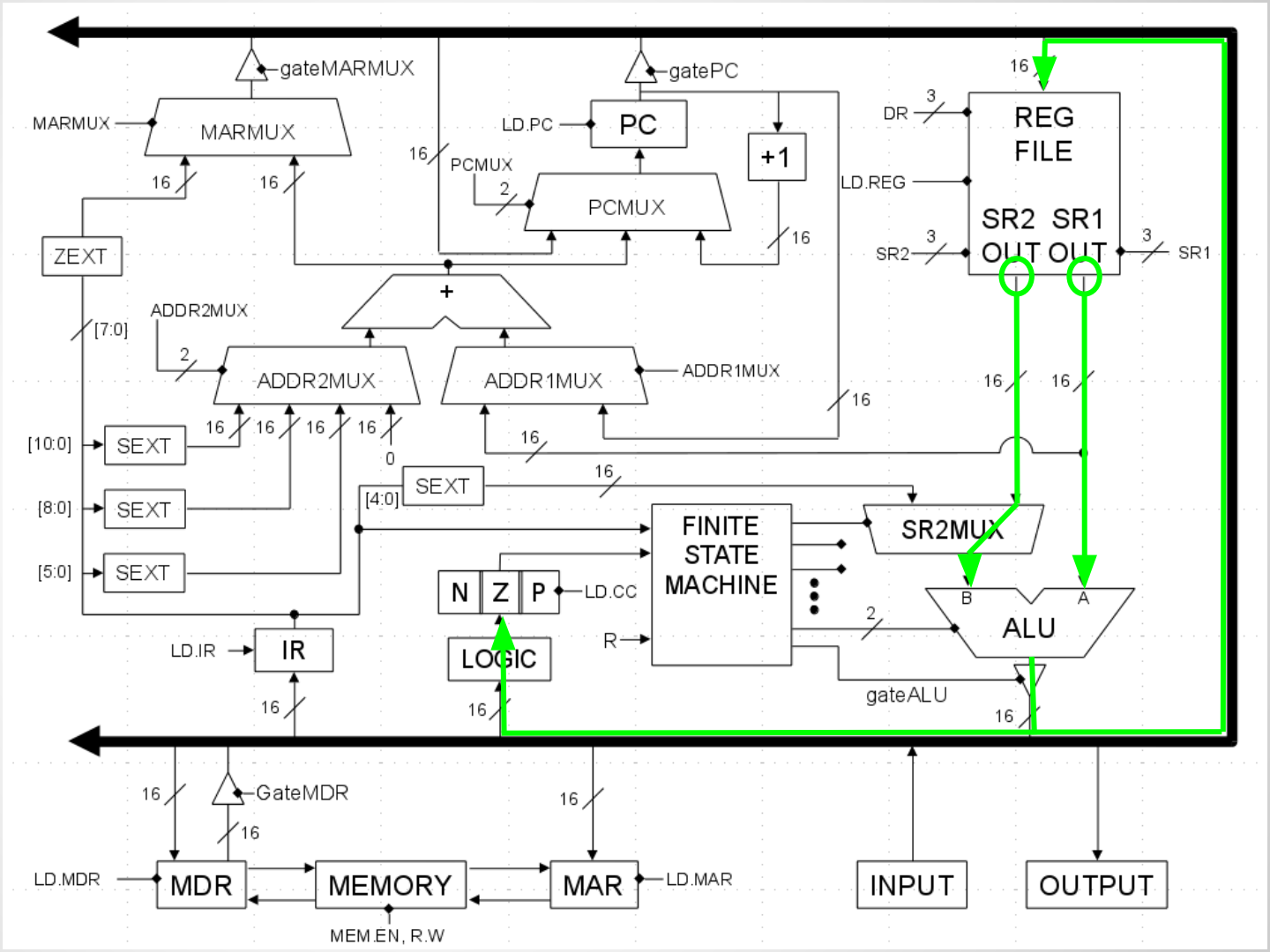
DR ← SR1 + (IR[5] == 0 ? SR2 : imm5)

ADD (SR2)

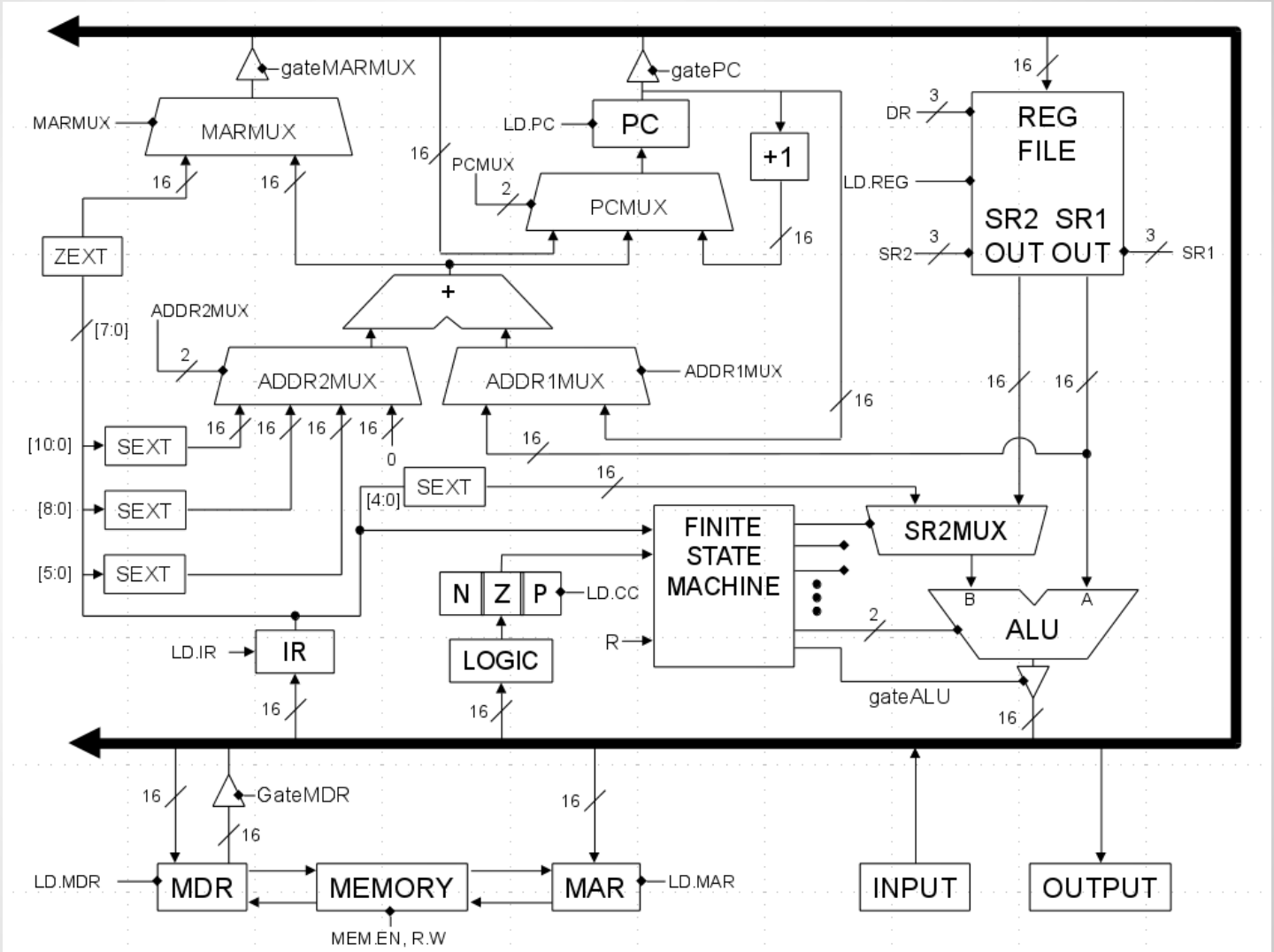

gateMARMUX
gatePC
MARMUX
MARMUX
LD.PC
PC
+1
PCMUX
PCMUX
REG FILE
DR
LD.REG
SR2
SR1
SR2 SR1
OUT OUT
ZEXT
+
ADDR2MUX
ADDR2MUX
ADDR1MUX
ADDR1MUX
[7:0]
[10:0]
[8:0]
[5:0]
[4:0]
SEXT
SEXT
SEXT
SEXT
FINITE STATE MACHINE
SR2MUX
N
Z
P
LD.CC
LOGIC
R
ALU
B
A
gateALU
LD.IR
IR
GateMDR
LD.MDR
MDR
MEMORY
MAR
LD.MAR
MEM.EN, R.W
INPUT
OUTPUT

ADD (SR2)

CC1:
SR1 = IR[8:6]
SR2 = IR[2:0]
SR2MUX = SR2
ALUK = ADD
gateALU
LD.REG
LD.CC
DR = IR[11:9]

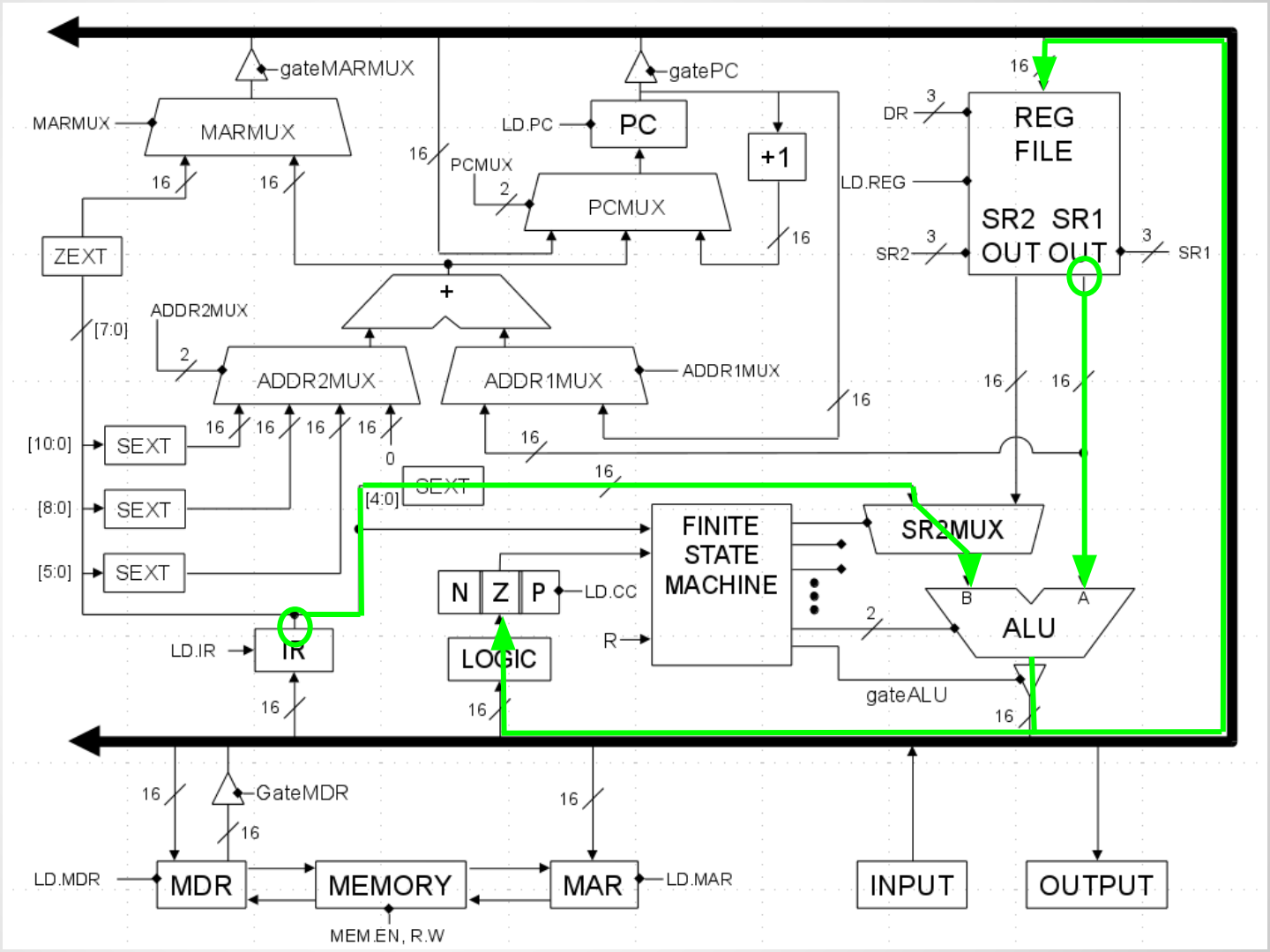


ADD (imm5)


gateMARMUX
gatePC
MARMUX
MARMUX
LD.PC
PC
+1
PCMUX
PCMUX
REG FILE
DR
LD.REG
SR2 SR1 OUT OUT
SR2
SR1
ZEXT
+
ADDR2MUX
ADDR2MUX
ADDR1MUX
ADDR1MUX
[7:0]
[10:0]
[8:0]
[5:0]
[4:0]
SEXT
SEXT
SEXT
SEXT
FINITE STATE MACHINE
SR2MUX
N
Z
P
LD.CC
LOGIC
R
ALU
B
A
LD.IR
IR
gateALU
GateMDR
LD.MDR
MDR
MEMORY
MAR
LD.MAR
MEM.EN, R.W
INPUT
OUTPUT

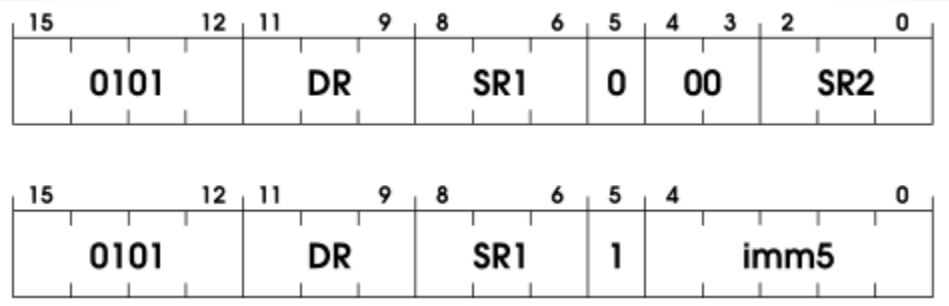
ADD (imm5)

CC1:
SR1 = IR[8:6]
SR2MUX = SEXT
ALUK = ADD
gateALU
LD.REG
LD.CC
DR = IR[11:9]



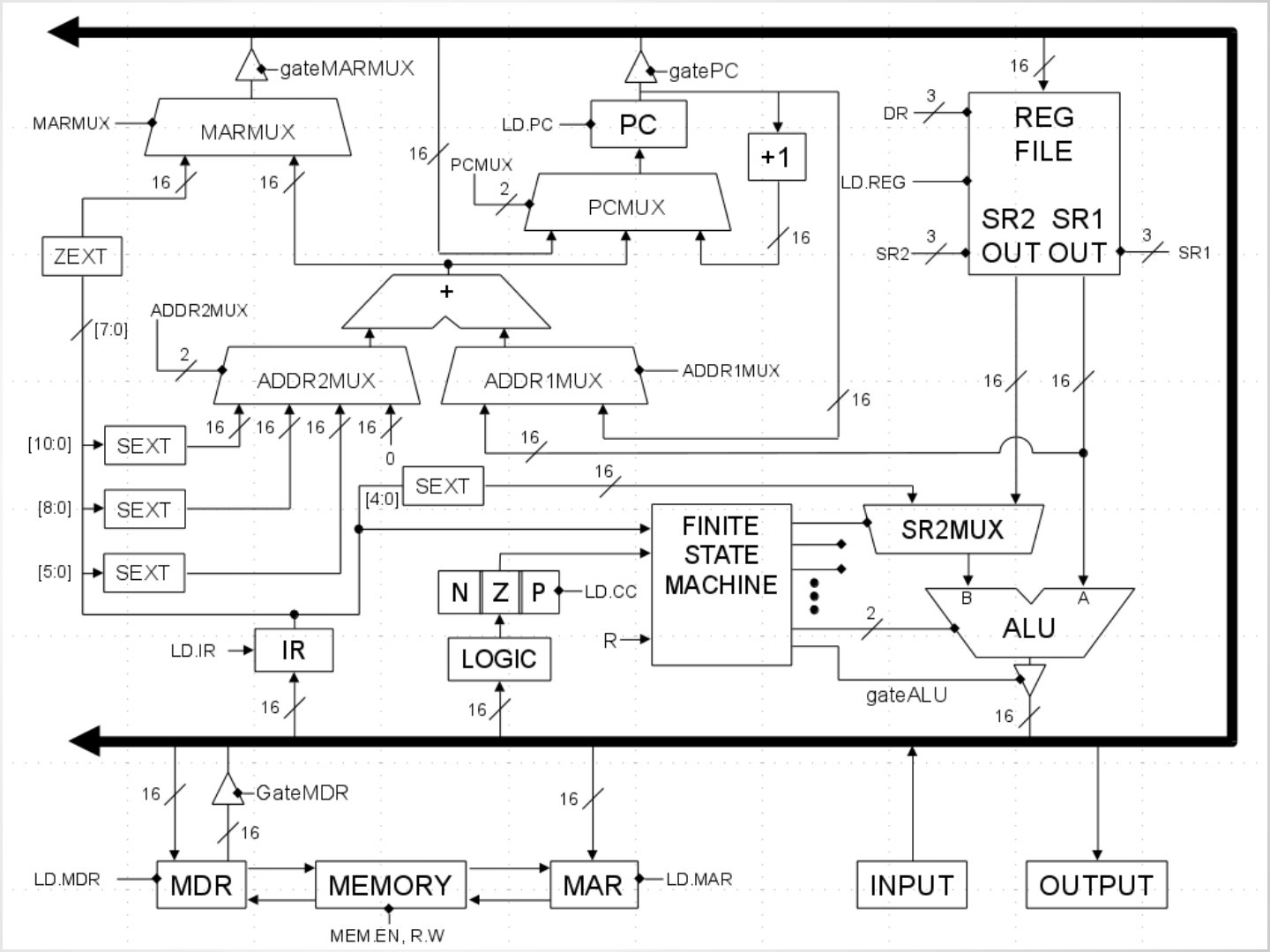
AND

The AND instruction performs a bitwise AND of a register and second operand, and stores the result in a register. AND also has an option for secondary register or immediate value.



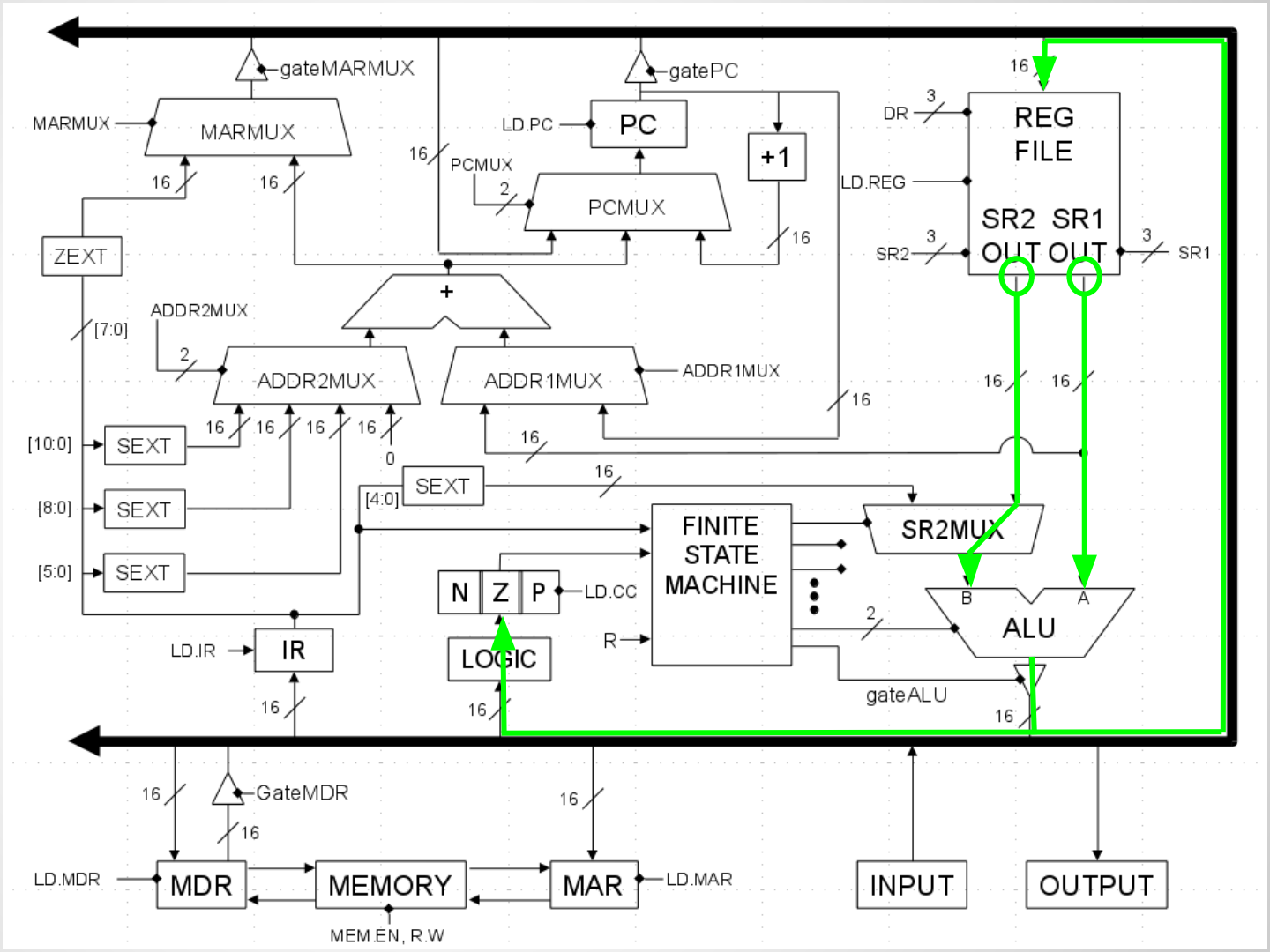
DR ← SR1 & (IR[5] == 0 ? SR2 : imm5)

AND (SR2)

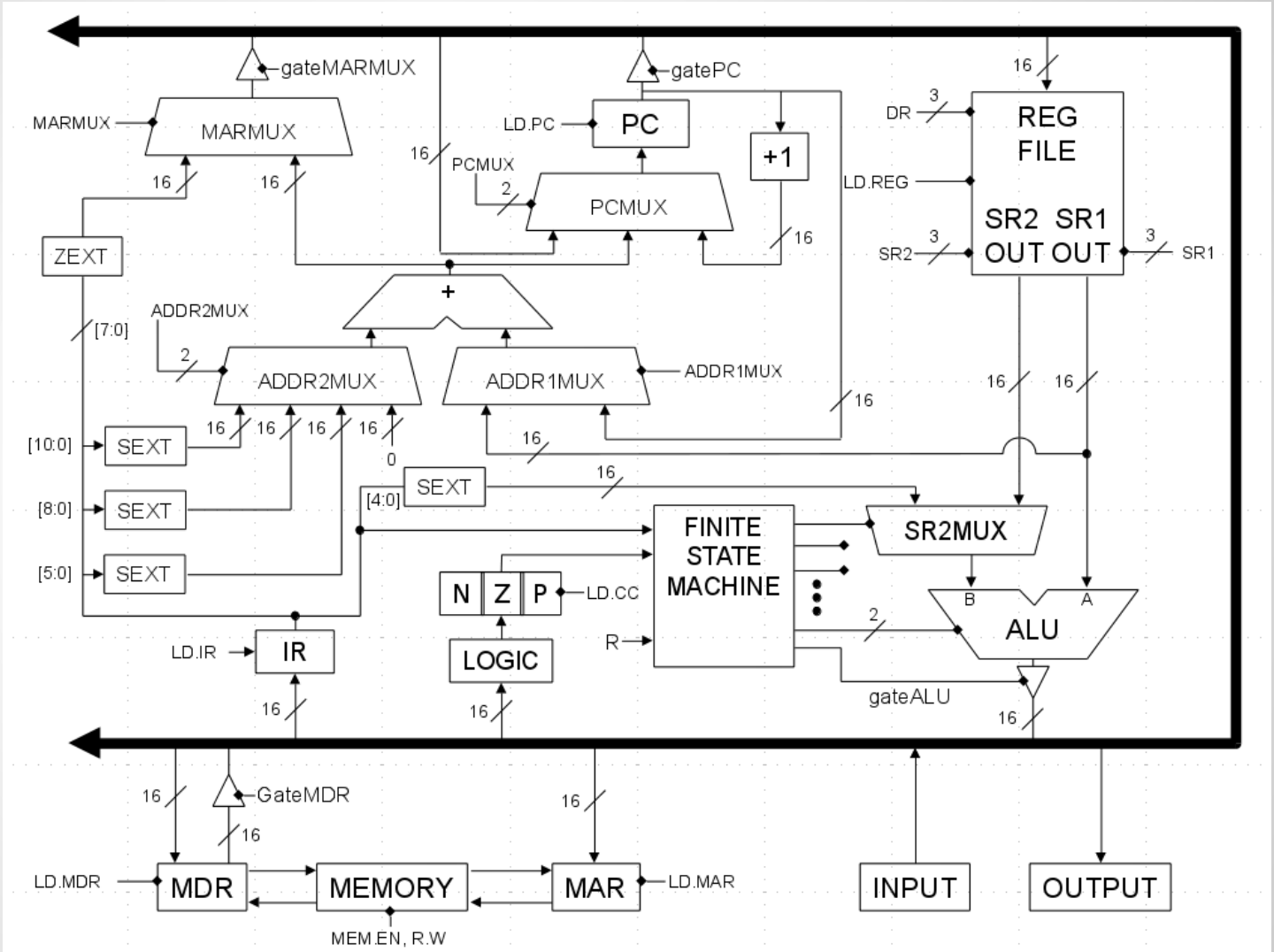

gateMARMUX
gatePC
MARMUX
MARMUX
LD.PC
PC
+1
PCMUX
PCMUX
REG FILE
DR
LD.REG
SR2 SR1 OUT OUT
SR2
SR1
ZEXT
+
ADDR2MUX
ADDR2MUX
ADDR1MUX
ADDR1MUX
[7:0]
[10:0]
[8:0]
[5:0]
[4:0]
SEXT
SEXT
SEXT
SEXT
FINITE STATE MACHINE
SR2MUX
N
Z
P
LD.CC
LOGIC
R
ALU
B
A
LD.IR
IR
gateALU
GateMDR
LD.MDR
MDR
MEMORY
MAR
LD.MAR
MEM.EN, R.W
INPUT
OUTPUT

AND (SR2)

CC1:
SR1 = IR[8:6]
SR2 = IR[2:0]
SR2MUX = SR2
ALUK = AND
gateALU
LD.REG
LD.CC
DR = IR[11:9]

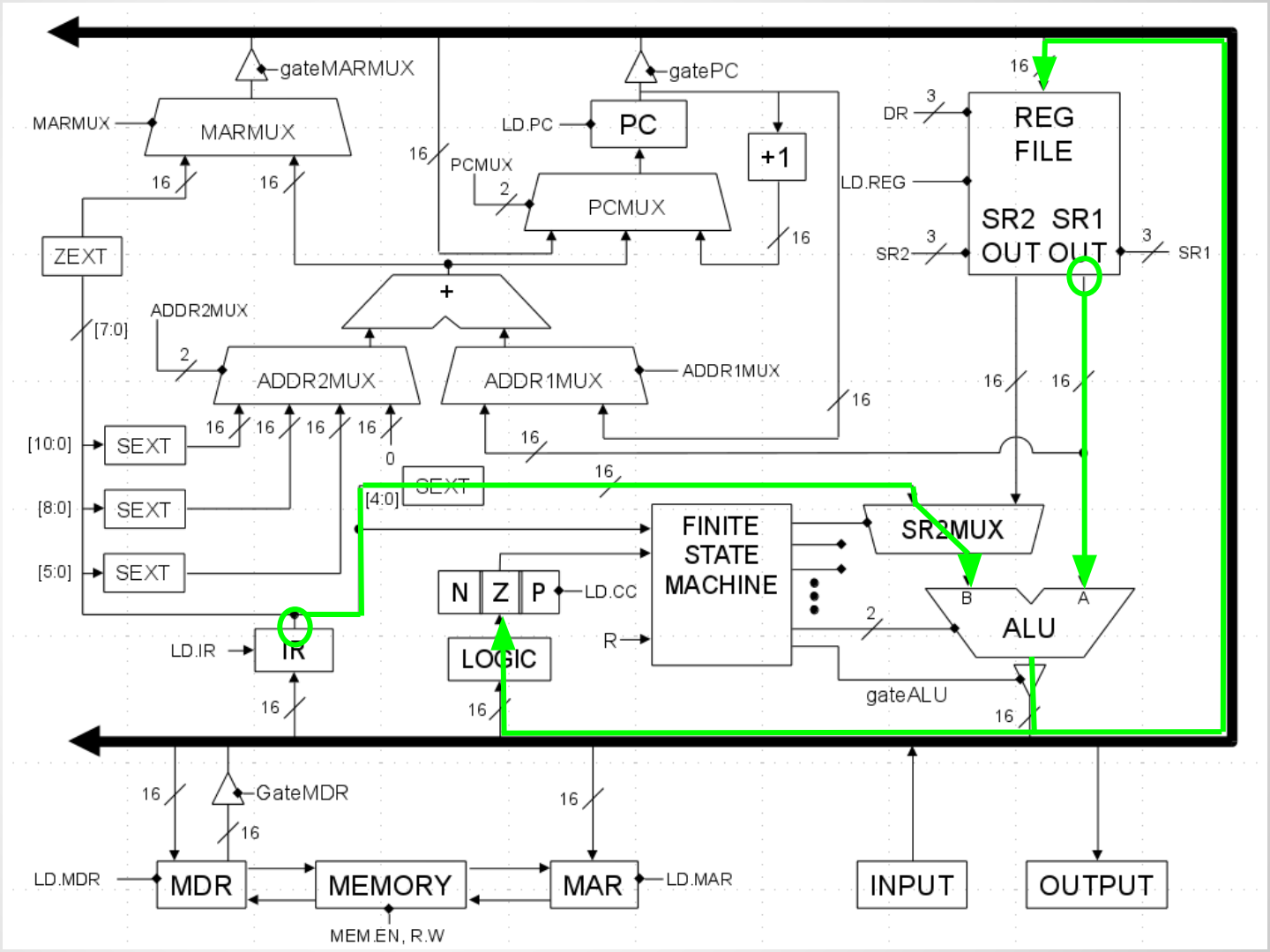


AND (imm5)


gateMARMUX
MARMUX
MARMUX
16
16
ZEXT
[7:0]
ADDR2MUX
2
ADDR2MUX
ADDR1MUX
ADDR1MUX
+
16
16
16
16
0
[10:0]
SEXT
[8:0]
SEXT
[5:0]
SEXT
[4:0]
SEXT
16
16
LD.IR
IR
16
N
Z
P
LD.CC
LOGIC
16
R
FINITE
STATE
MACHINE
SR2MUX
B
A
ALU
2
gateALU
16
gatePC
LD.PC
PC
+1
16
PCMUX
2
PCMUX
16
16
DR
3
REG
FILE
LD.REG
SR2
SR1
OUT
OUT
SR2
3
3
SR1
16
16
16
16
16
GateMDR
16
16
16
LD.MDR
MDR
MEMORY
MAR
LD.MAR
MEM.EN, R.W
INPUT
OUTPUT

AND (imm5)

CC1:
SR1 = IR[8:6]
SR2MUX = SEXT
ALUK = AND
gateALU
LD.REG
LD.CC
DR = IR[11:9]

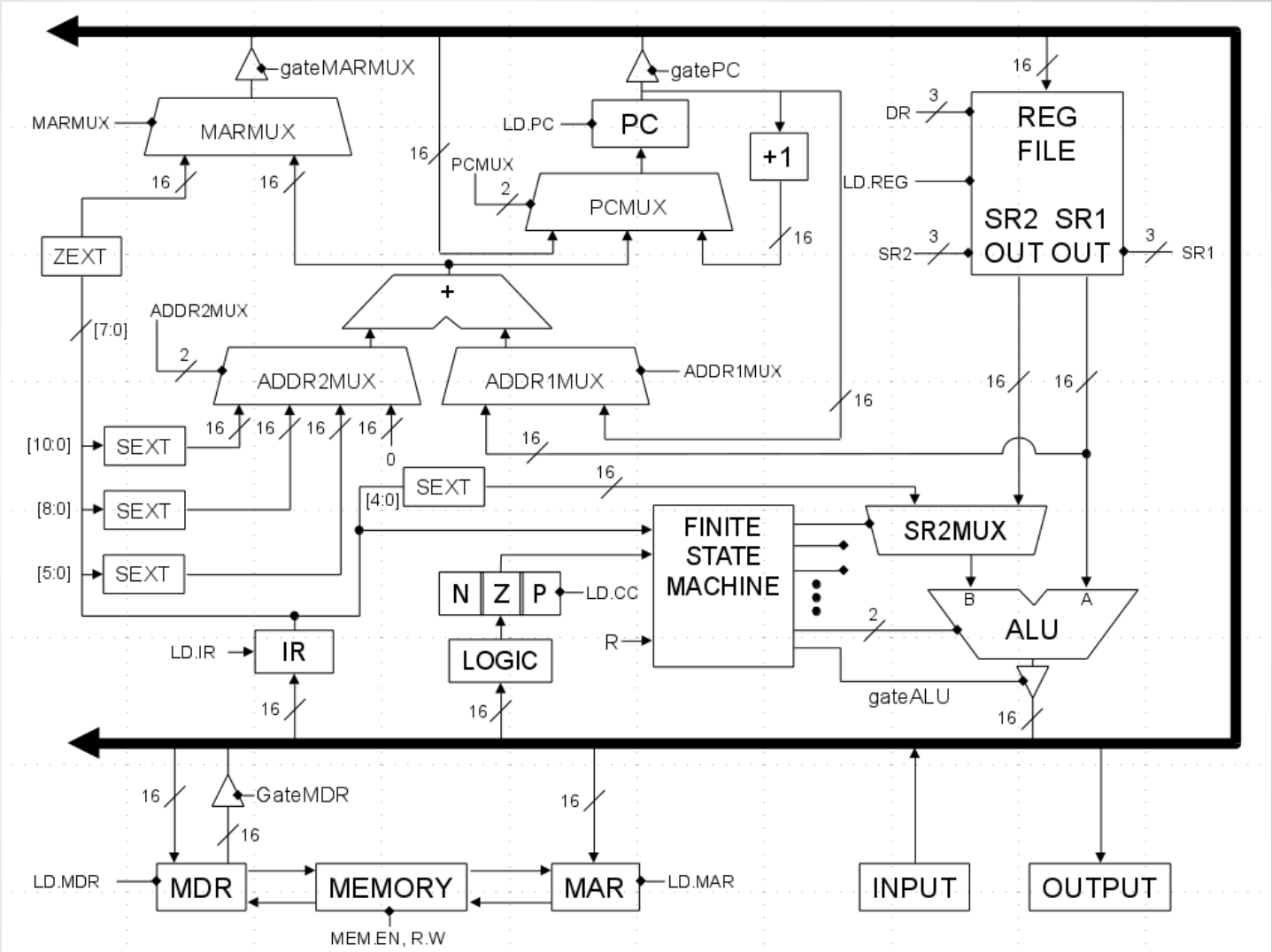


NOT

The NOT instruction performs a bitwise NOT of a register, and stores it in another register.

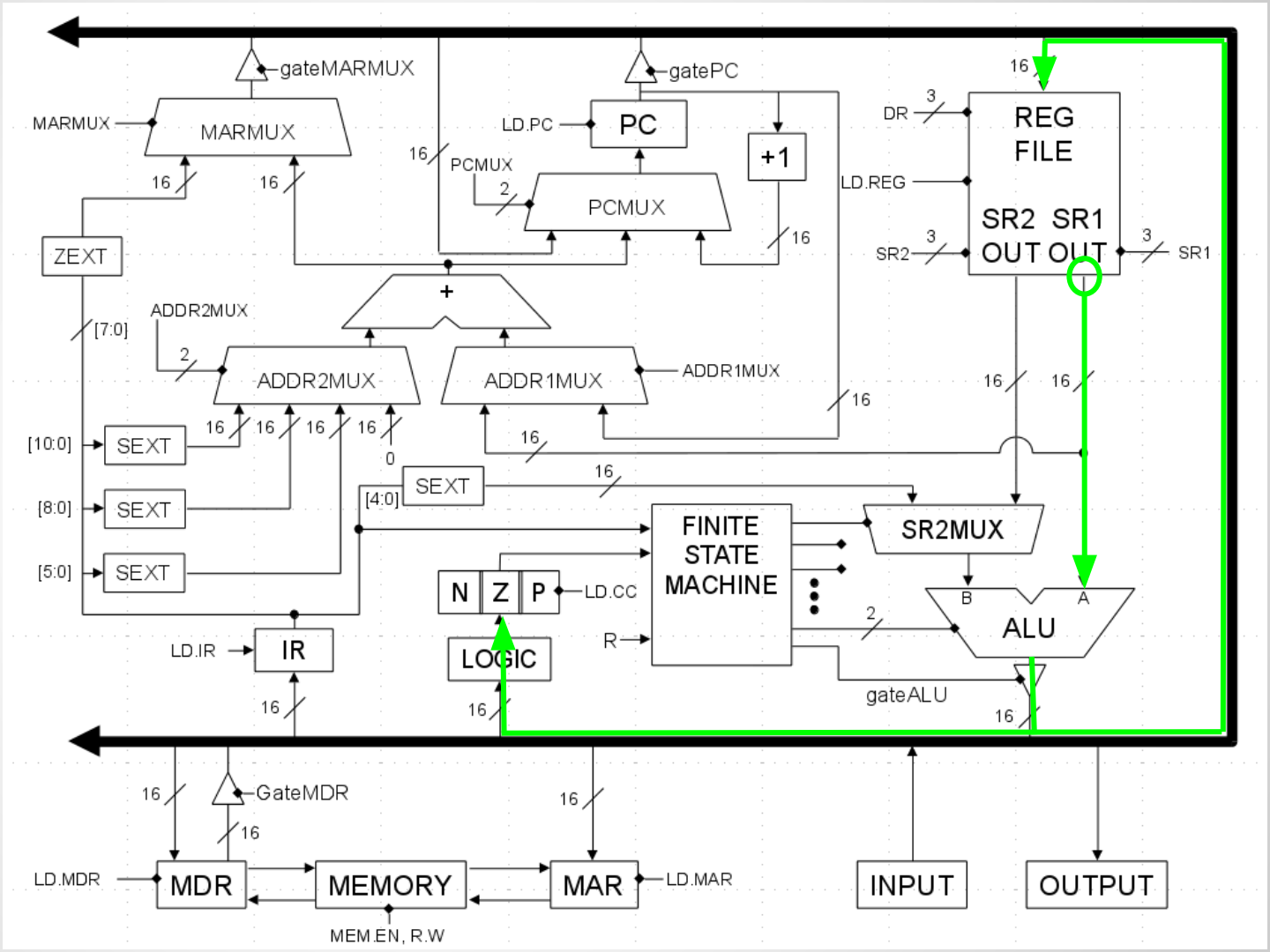
DR ← ~SR

NOT


gateMARMUX
gatePC
MARMUX
MARMUX
LD.PC
PC
+1
PCMUX
PCMUX
REG FILE
DR
LD.REG
SR2 SR1 OUT OUT
SR2
SR1
ZEXT
+
ADDR2MUX
ADDR2MUX
ADDR1MUX
ADDR1MUX
[7:0]
[10:0]
[8:0]
[5:0]
SEXT
SEXT
SEXT
[4:0]
SEXT
FINITE STATE MACHINE
SR2MUX
N
Z
P
LD.CC
LOGIC
R
ALU
B
A
LD.IR
IR
gateALU
GateMDR
LD.MDR
MDR
MEMORY
MAR
LD.MAR
MEM.EN, R.W
INPUT
OUTPUT

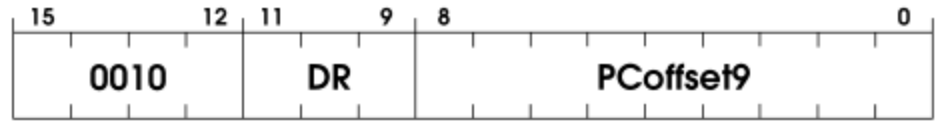
NOT

CC1:
SR1 = IR[8:6]
ALUK = NOT
gateALU
LD.REG
LD.CC
DR = IR[11:9]



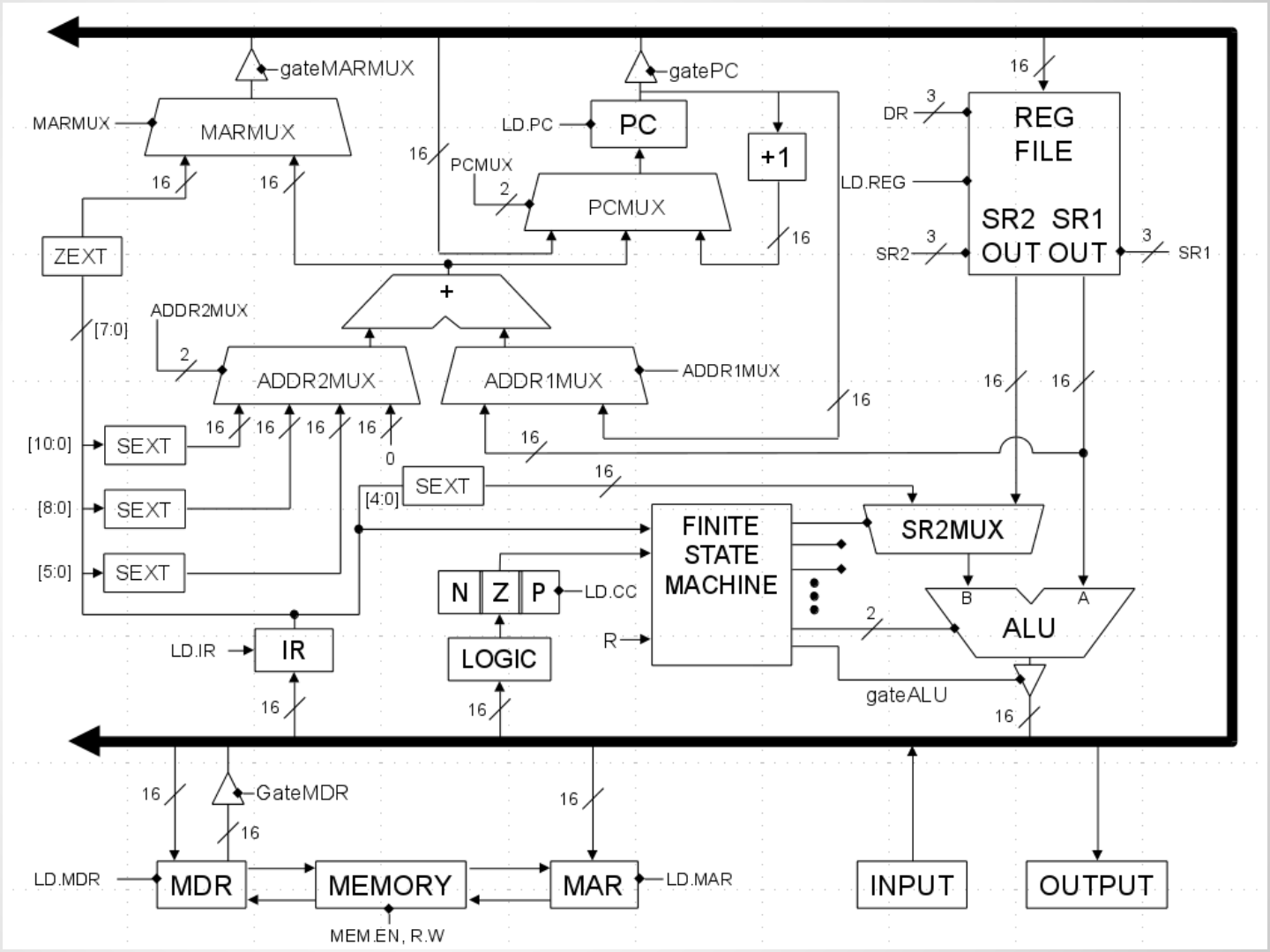
LD

The LD instruction loads a value from memory, nearby the instruction itself, into a register.



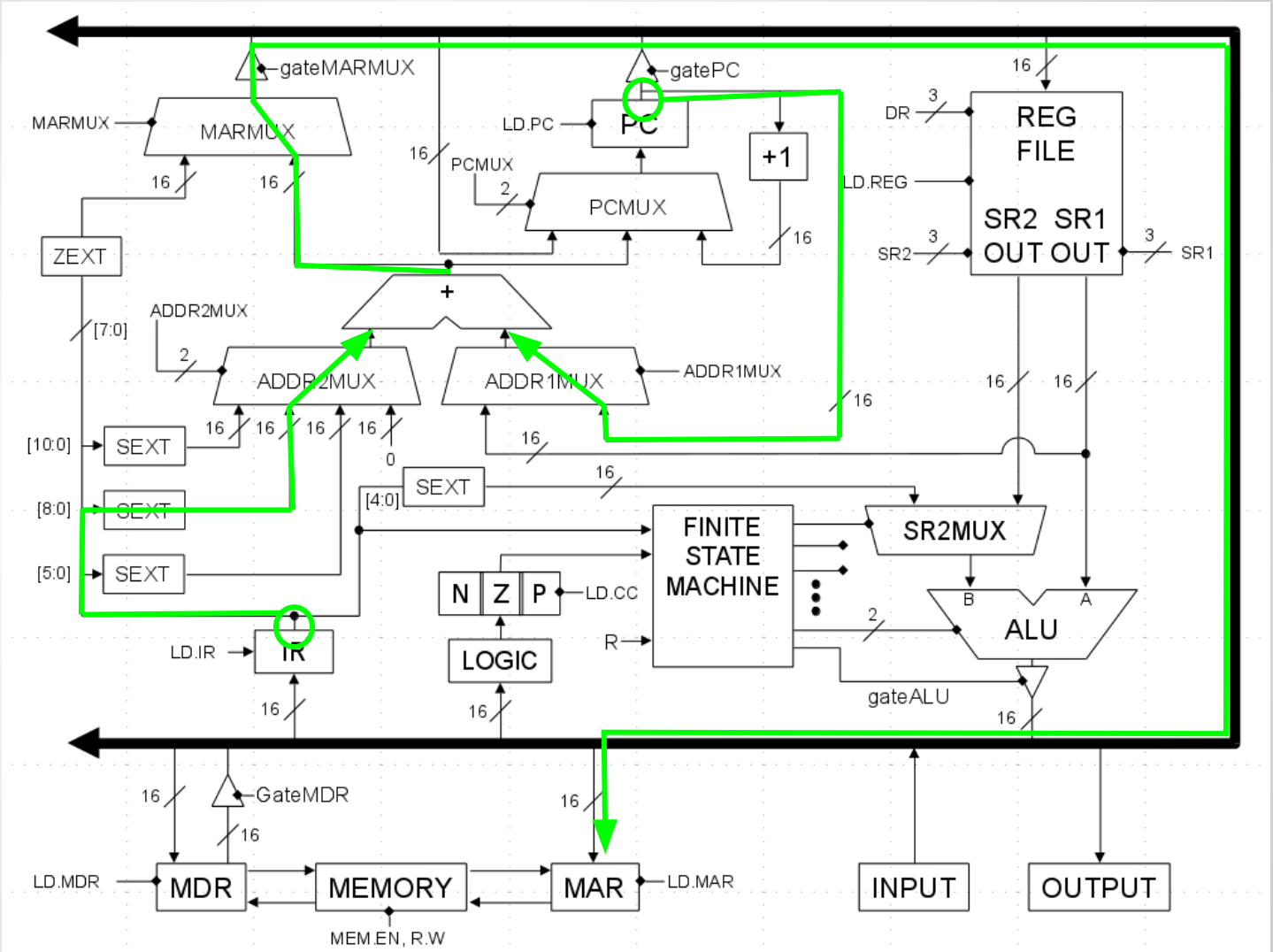
DR ← mem[PC + PCoffset9]

LD


gateMARMUX
MARMUX
MARMUX
gatePC
LD.PC
PC
+1
PCMUX
PCMUX
2
16
16
16
16
16
DR
3
REG
FILE
LD.REG
SR2 SR1
OUT OUT
SR2
3
3
SR1
ZEXT
[7:0]
+
ADDR2MUX
2
ADDR2MUX
ADDR1MUX
ADDR1MUX
16
16
16
[10:0]
SEXT
16
16
16
16
0
[8:0]
SEXT
[5:0]
SEXT
[4:0]
SEXT
16
16
16
FINITE
STATE
MACHINE
SR2MUX
N
Z
P
LD.CC
R
2
B
A
ALU
LD.IR
IR
LOGIC
gateALU
16
16
16
16
GateMDR
16
16
16
LD.MDR
MDR
MEMORY
MAR
LD.MAR
INPUT
OUTPUT
MEM.EN, R.W

LD

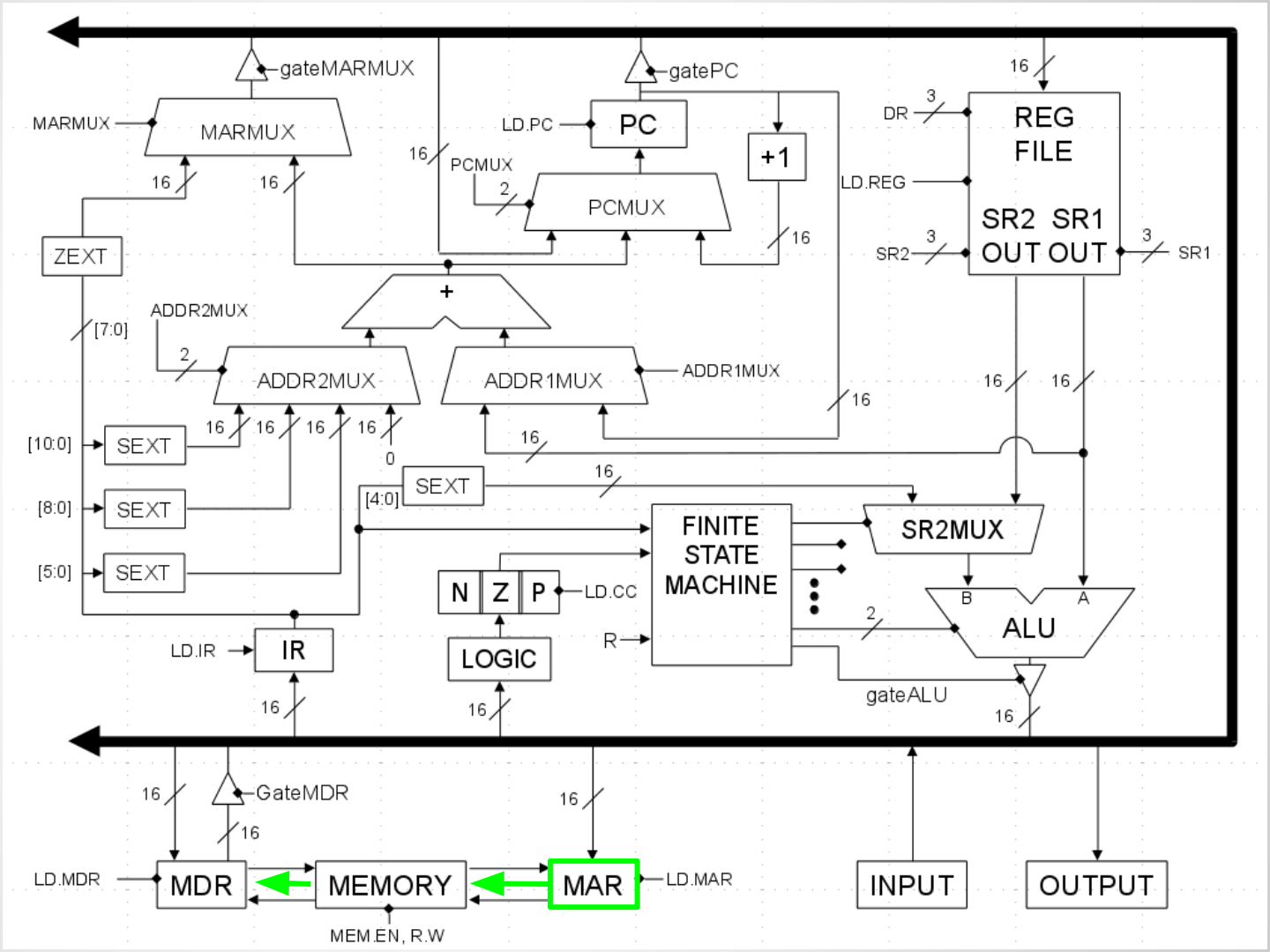
CC1:
ADDR1MUX = PC
ADDR2MUX =
SEXT[8:0]
MARMUX = ADDER
gateMARMUX
LD.MAR



LD

CC1:
ADDR1MUX = PC
ADDR2MUX =
SEXT[8:0]
MARMUX = ADDER
gateMARMUX
LD.MAR

CC2:
MEM.EN
LD.MDR

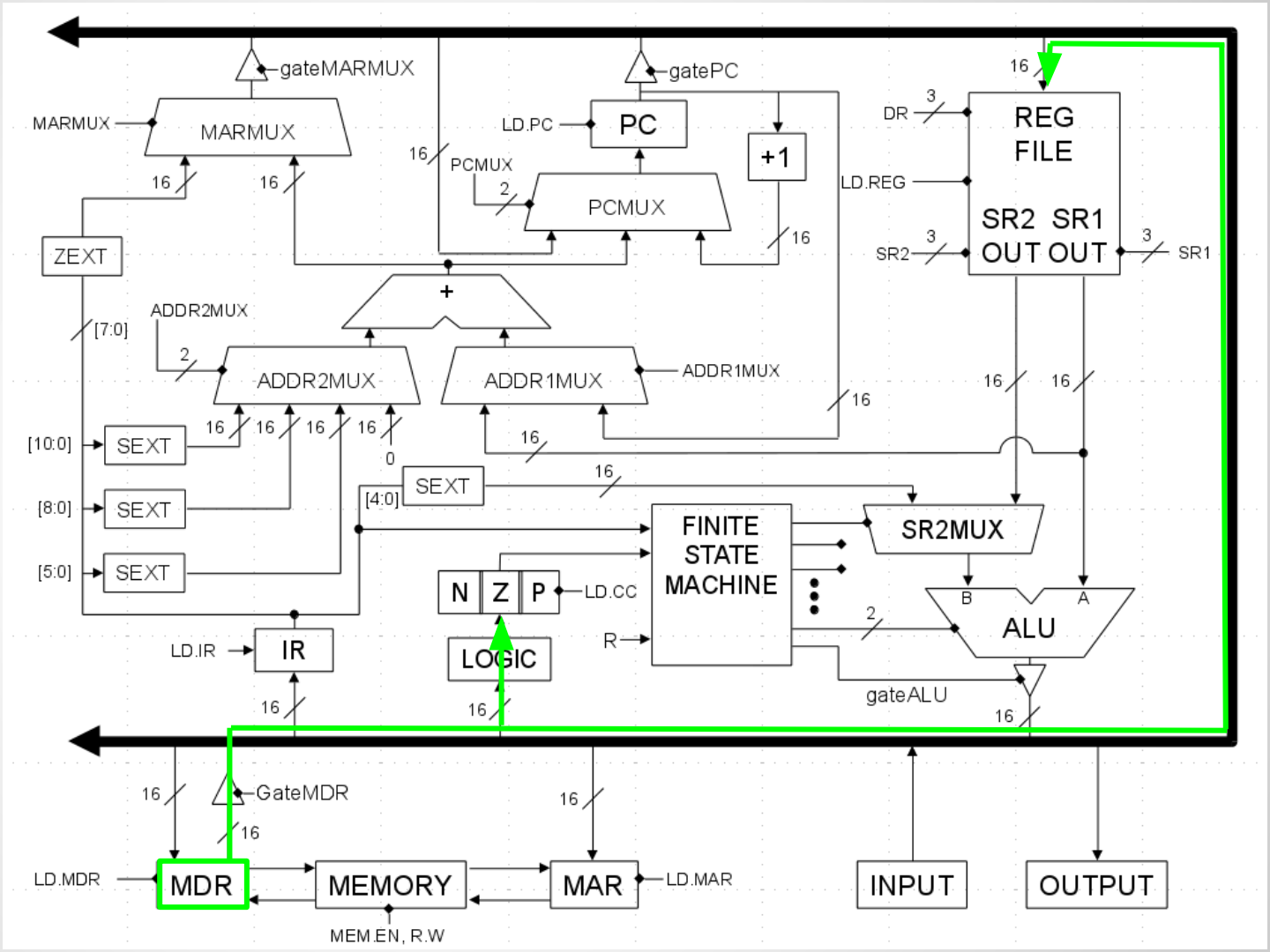


LD

CC1:
ADDR1MUX = PC
ADDR2MUX =
SEXT[8:0]
MARMUX = ADDER
gateMARMUX
LD.MAR

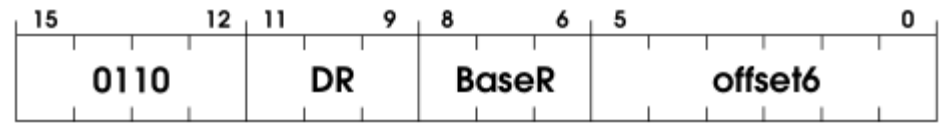
CC2:
MEM.EN
LD.MDR

CC3:
gateMDR
DR = IR[11:9]
LD.REG
LD.CC



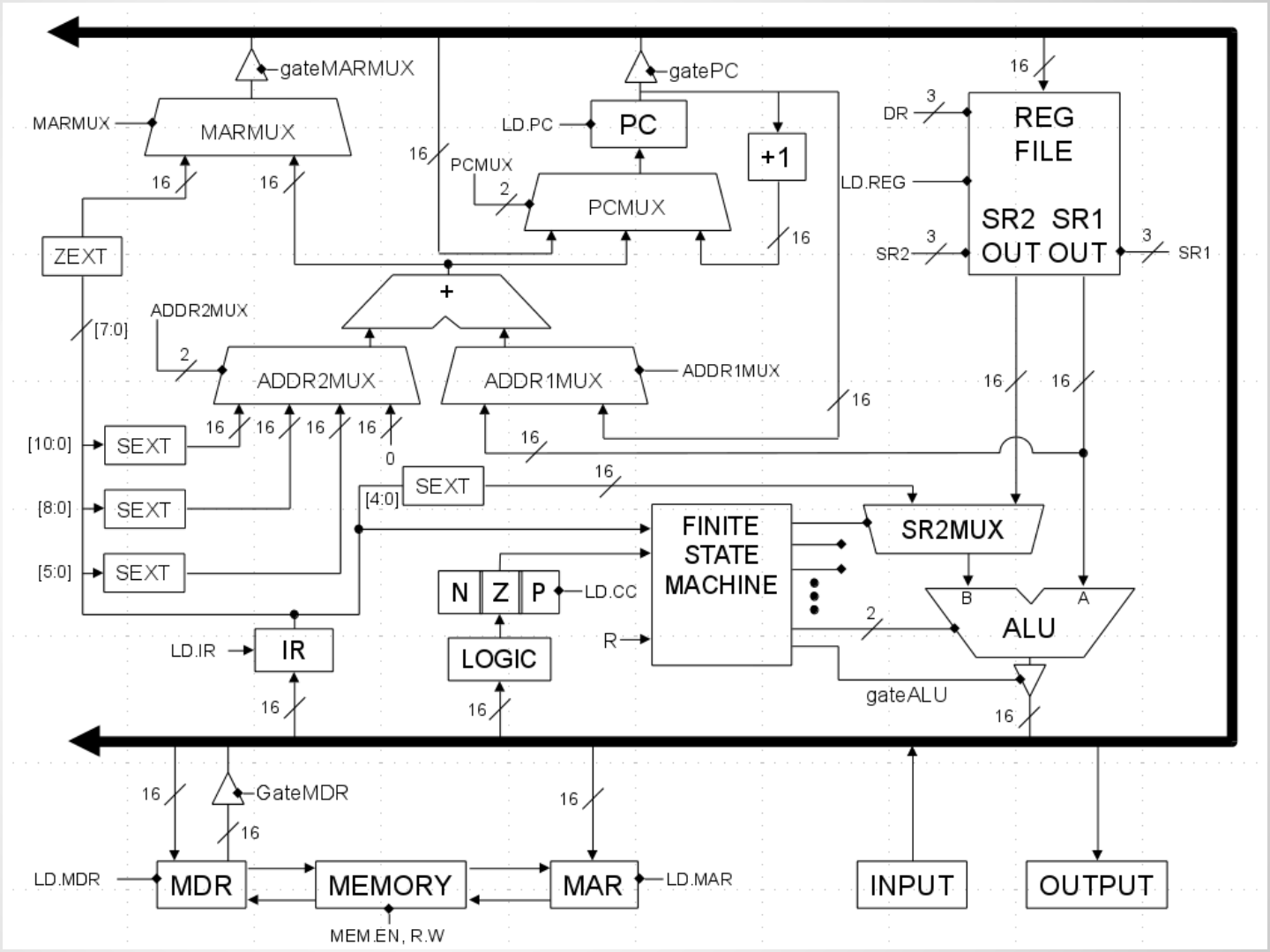
LDR

The LDR instruction loads a value from memory, pointed to by a base register and offset from that location, into a register.



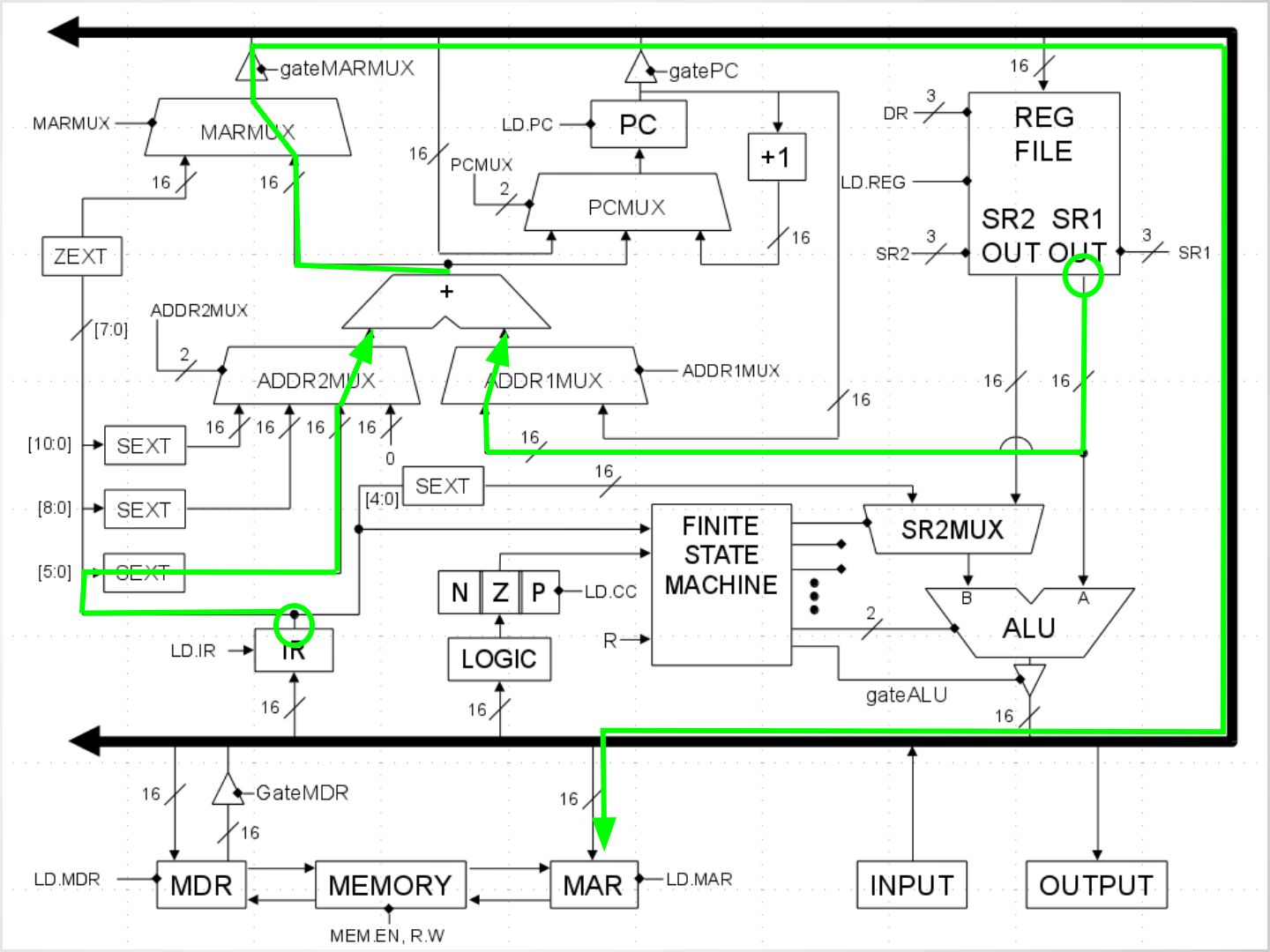
DR ← mem[BaseR + offset6]

LDR


gateMARMUX
gatePC
MARMUX
MARMUX
LD.PC
PC
+1
PCMUX
PCMUX
REG FILE
DR
LD.REG
SR2 SR1 OUT OUT
SR2
SR1
ZEXT
[7:0]
ADDR2MUX
ADDR2MUX
ADDR1MUX
ADDR1MUX
+
[10:0]
SEXT
[8:0]
SEXT
[5:0]
SEXT
[4:0]
SEXT
FINITE STATE MACHINE
SR2MUX
N
Z
P
LD.CC
LOGIC
R
ALU
LD.IR
IR
gateALU
GateMDR
LD.MDR
MDR
MEMORY
MAR
LD.MAR
MEM.EN, R.W
INPUT
OUTPUT

LDR

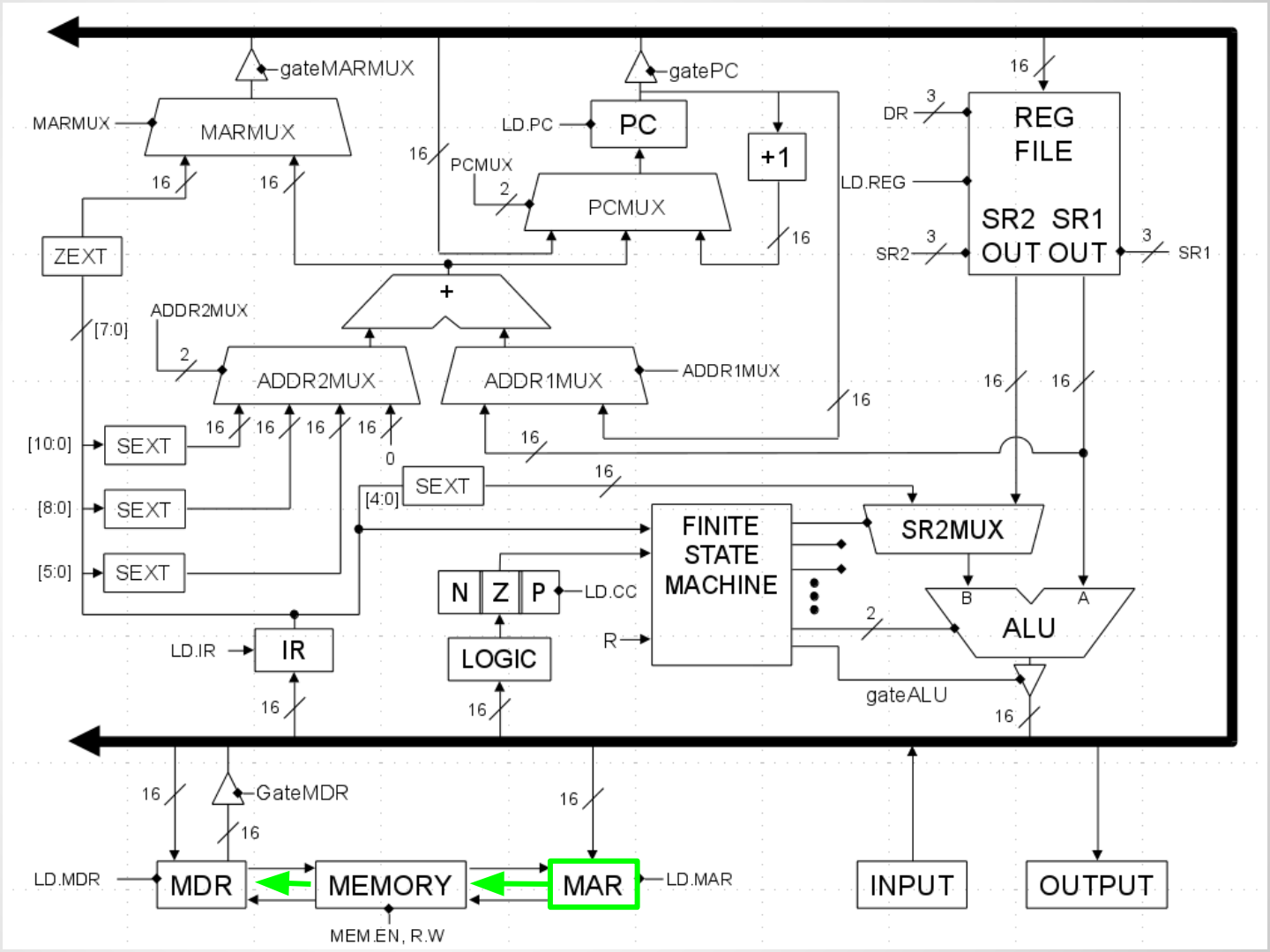
CC1:
ADDR1MUX = BaseR
ADDR2MUX =
SEXT[5:0]
MARMUX = ADDER
gateMARMUX
LD.MAR



LDR

CC1:
ADDR1MUX = BaseR
ADDR2MUX =
SEXT[5:0]
MARMUX = ADDER
gateMARMUX
LD.MAR

CC2:
MEM.EN
LD.MDR

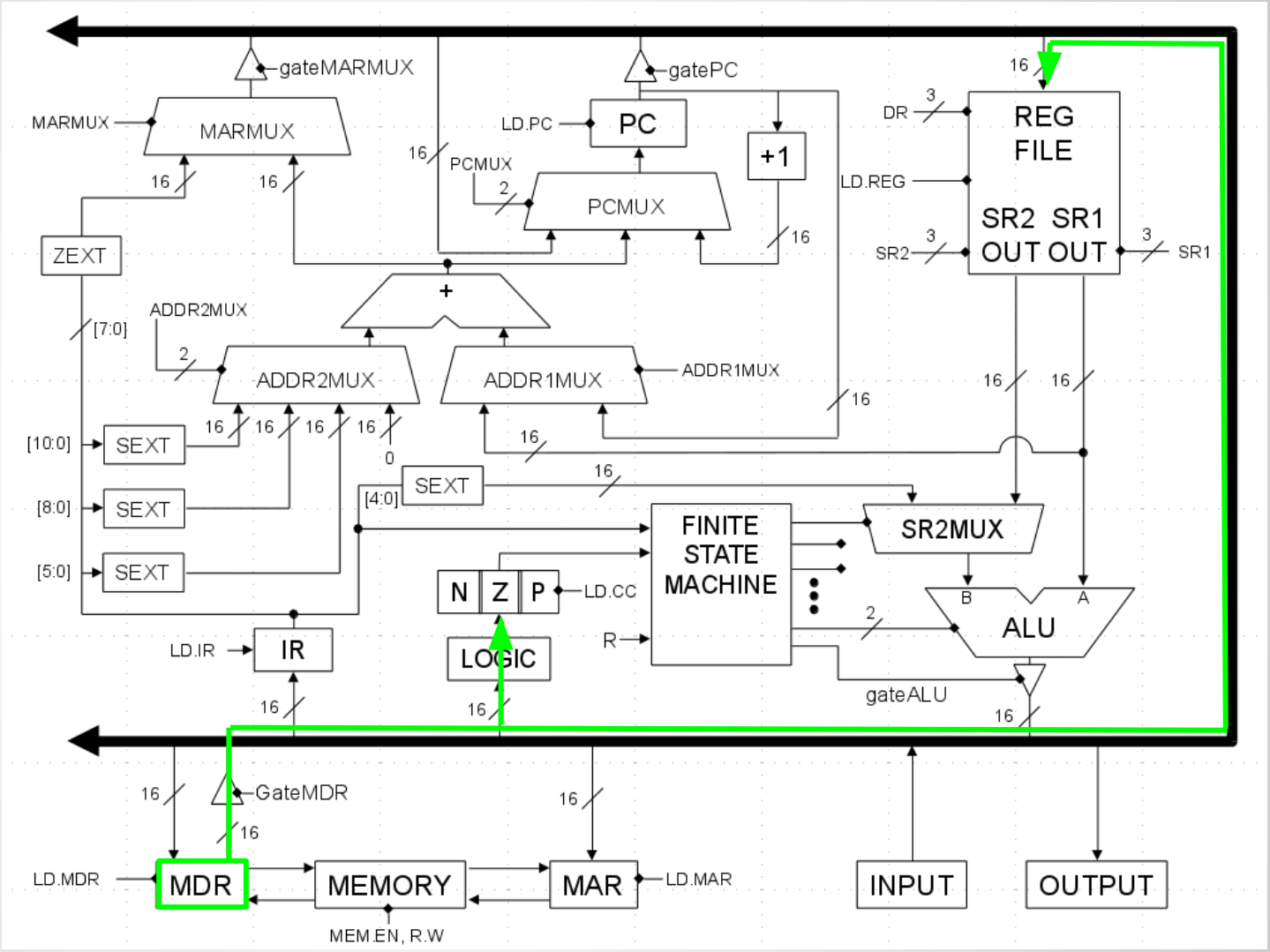


LDR

CC1:
ADDR1MUX = BaseR
ADDR2MUX =
SEXT[5:0]
MARMUX = ADDER
gateMARMUX
LD.MAR

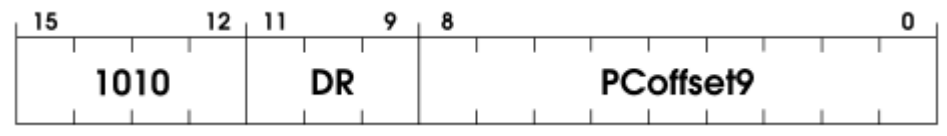
CC2:
MEM.EN
LD.MDR

CC3:
gateMDR
DR = IR[11:9]
LD.REG
LD.CC



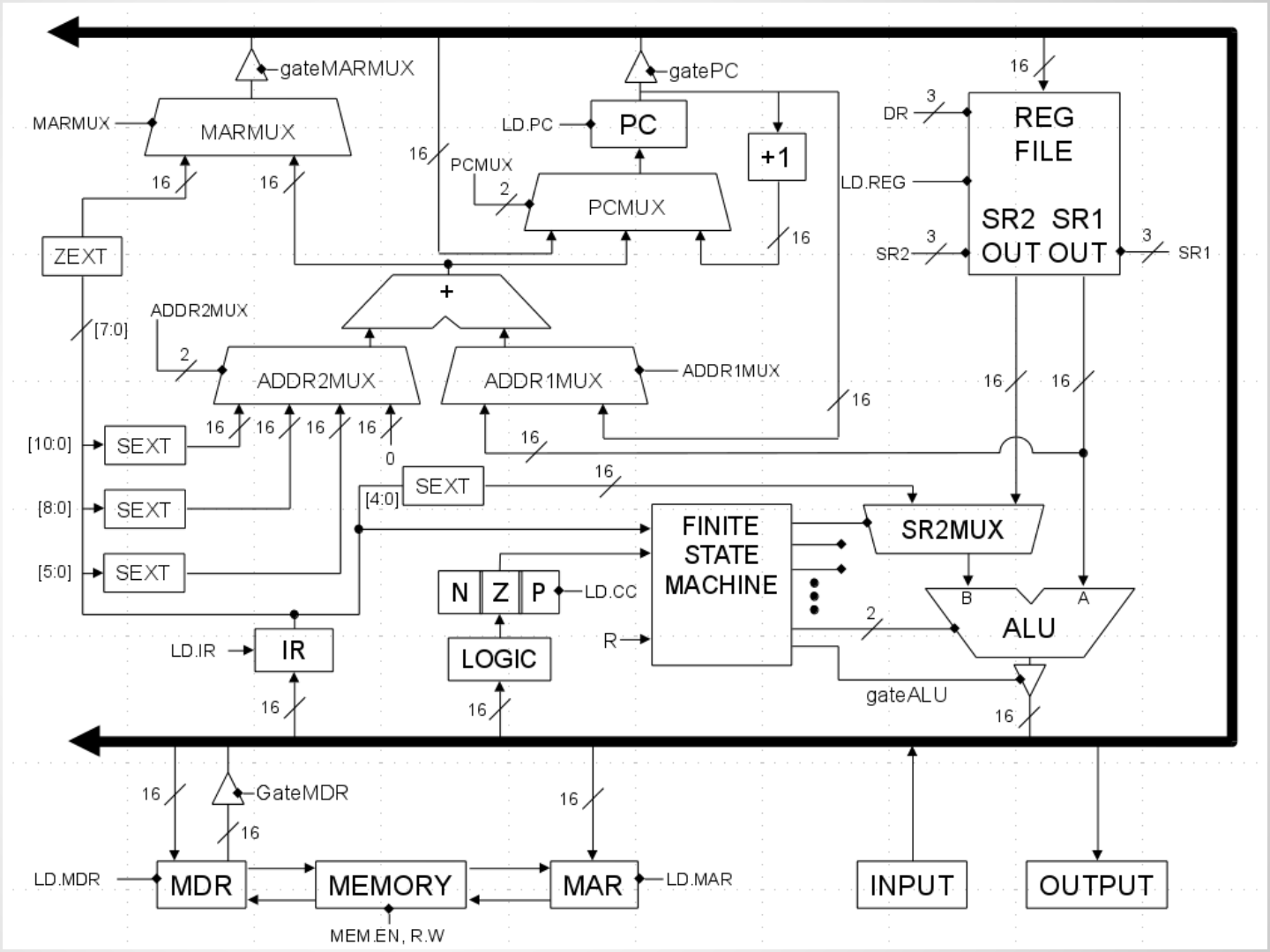
LDI

The LDI instruction takes a value from memory nearby the instruction itself, gets a pointer from that memory location, then loads the contents of the pointed location into a register.



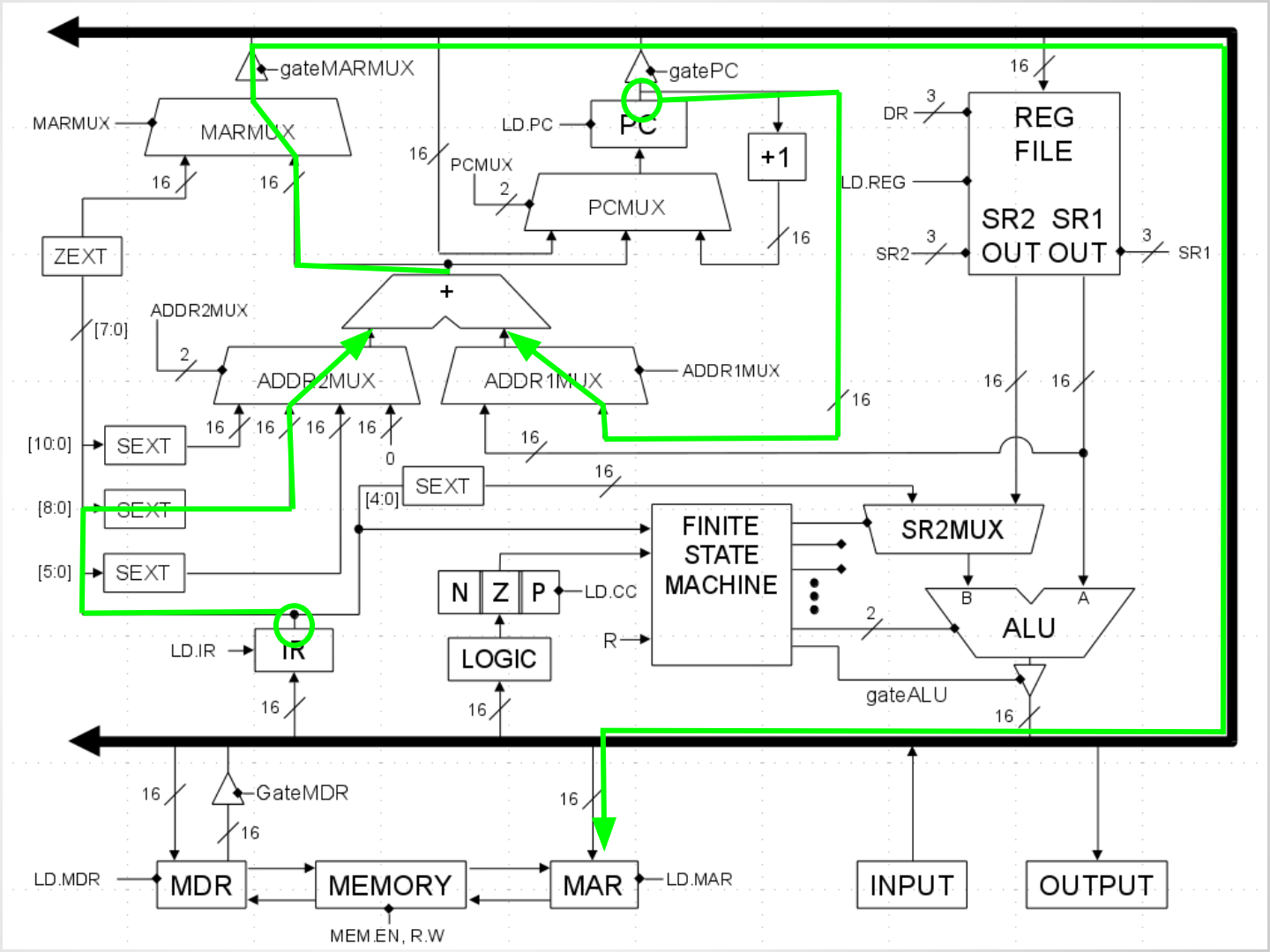
DR ← mem[mem[PC + PCoffset9]]

LDI


gateMARMUX
MARMUX
MARMUX
gatePC
LD.PC
PC
+1
PCMUX
PCMUX
2
16
16
16
16
16
DR
3
REG
FILE
LD.REG
SR2 SR1
OUT OUT
SR2
3
3
SR1
ZEXT
[7:0]
+
ADDR2MUX
2
ADDR2MUX
ADDR1MUX
ADDR1MUX
16
16
16
16
16
16
16
16
16
16
[10:0]
SEXT
[8:0]
SEXT
[5:0]
SEXT
0
SEXT
[4:0]
16
FINITE
STATE
MACHINE
SR2MUX
N
Z
P
LD.CC
LOGIC
R
B
A
2
ALU
LD.IR
IR
gateALU
16
16
16
16
GateMDR
16
16
16
LD.MDR
MDR
MEMORY
MAR
LD.MAR
INPUT
OUTPUT
MEM.EN, R.W

LDI

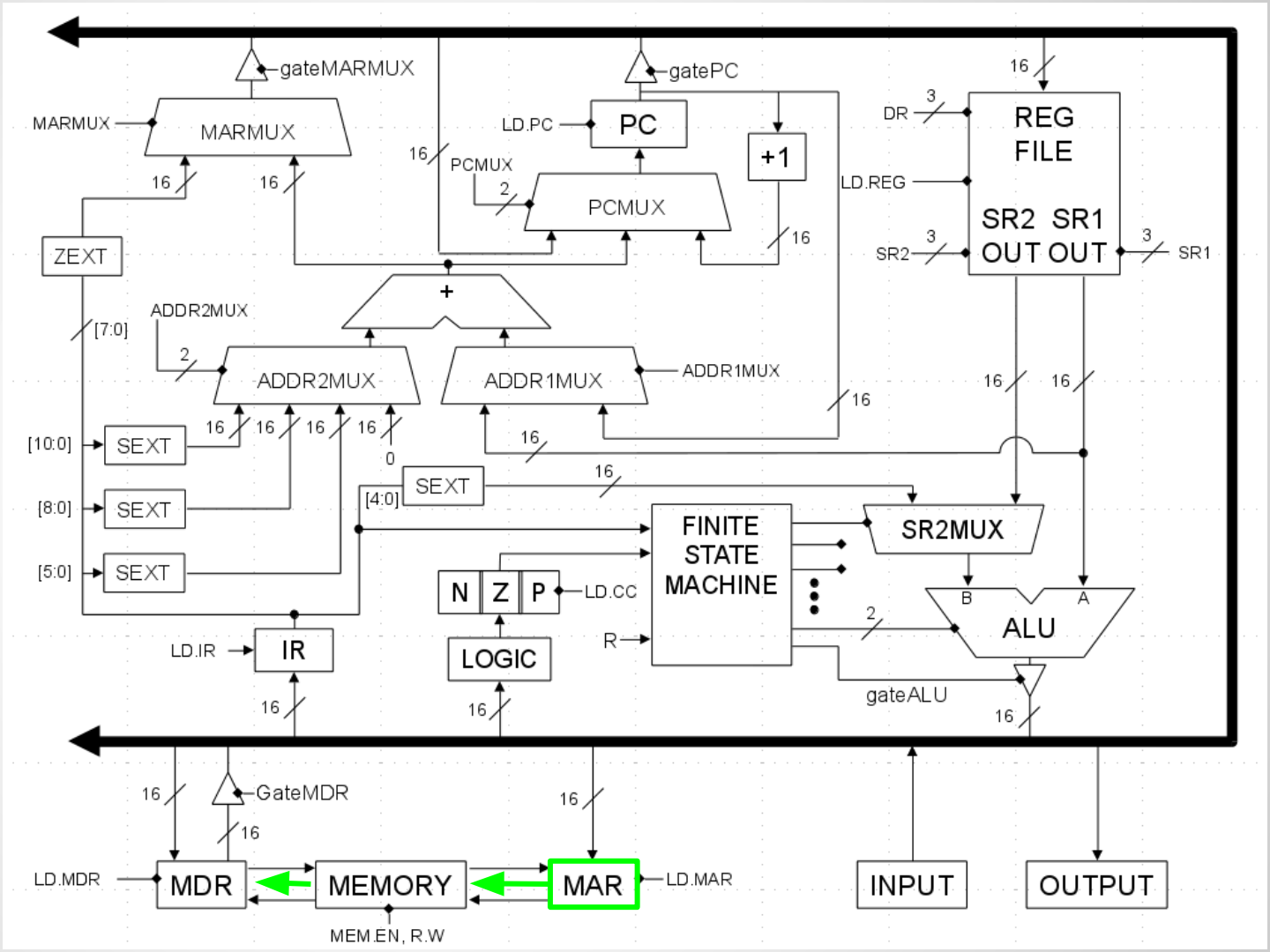
CC1:
ADDR1MUX = PC
ADDR2MUX =
SEXT[8:0]
MARMUX = ADDER
gateMARMUX
LD.MAR



LDI

CC1:
ADDR1MUX = PC
ADDR2MUX =
SEXT[8:0]
MARMUX = ADDER
gateMARMUX
LD.MAR

CC2:
MEM.EN
LD.MDR

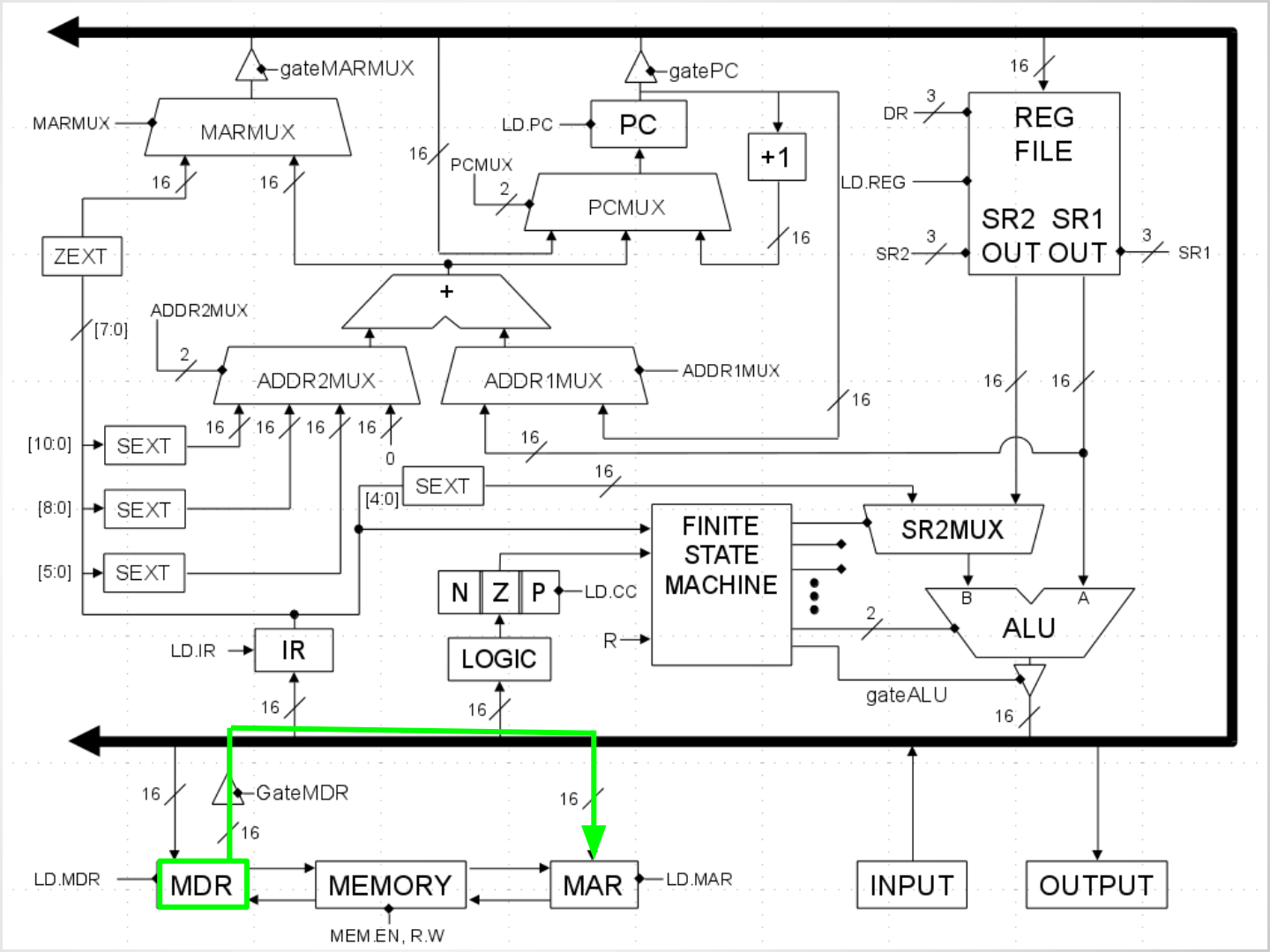


LDI

CC1:
ADDR1MUX = PC
ADDR2MUX =
SEXT[8:0]
MARMUX = ADDER
gateMARMUX
LD.MAR

CC2:
MEM.EN
LD.MDR

CC3:
gateMDR
LD.MAR



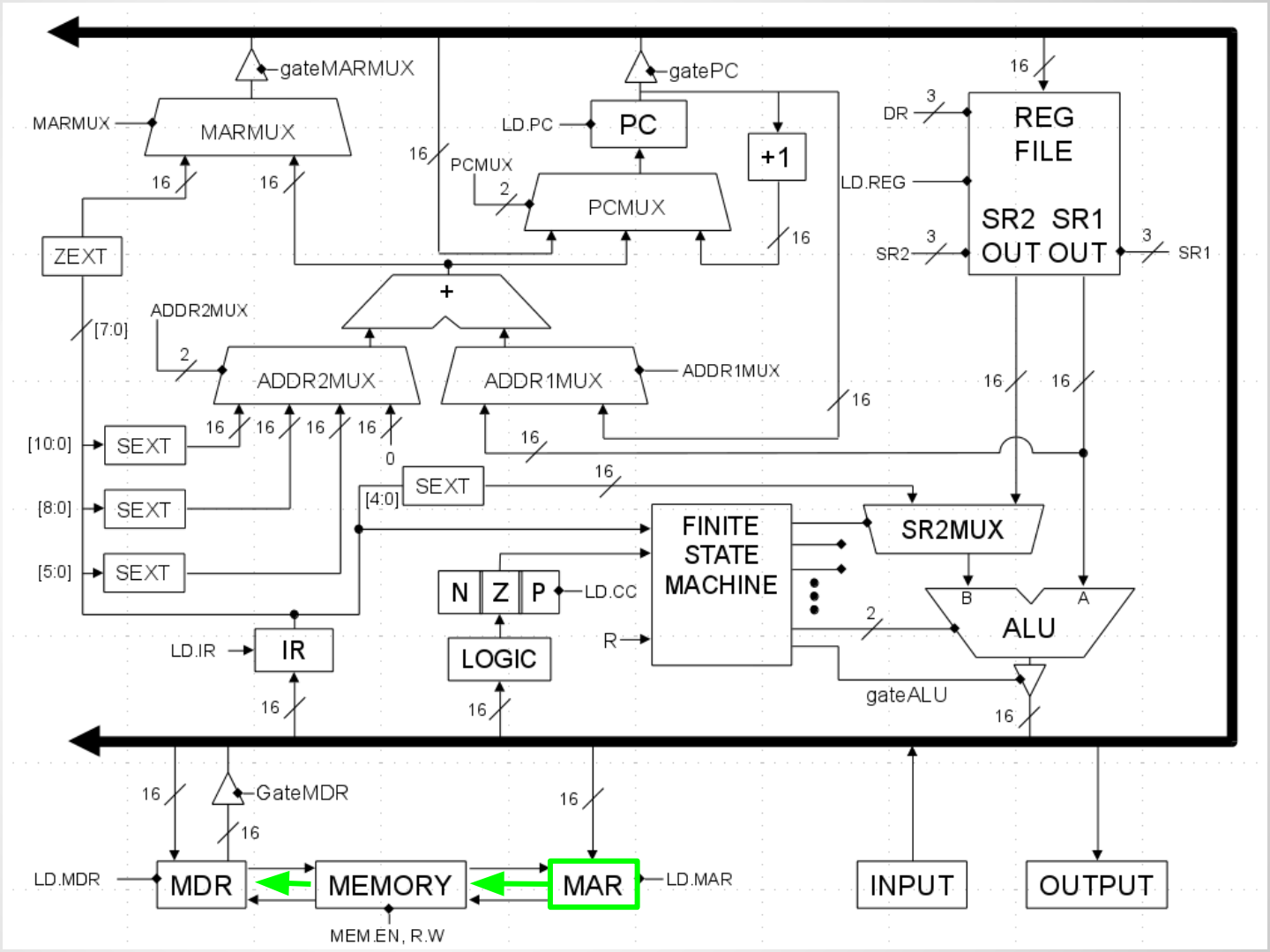
LDI

CC1:
ADDR1MUX = PC
ADDR2MUX =
SEXT[8:0]
MARMUX = ADDER
gateMARMUX
LD.MAR

CC2:
MEM.EN
LD.MDR

CC3:
gateMDR
LD.MAR

CC4:
MEM.EN
LD.MDR



LDI

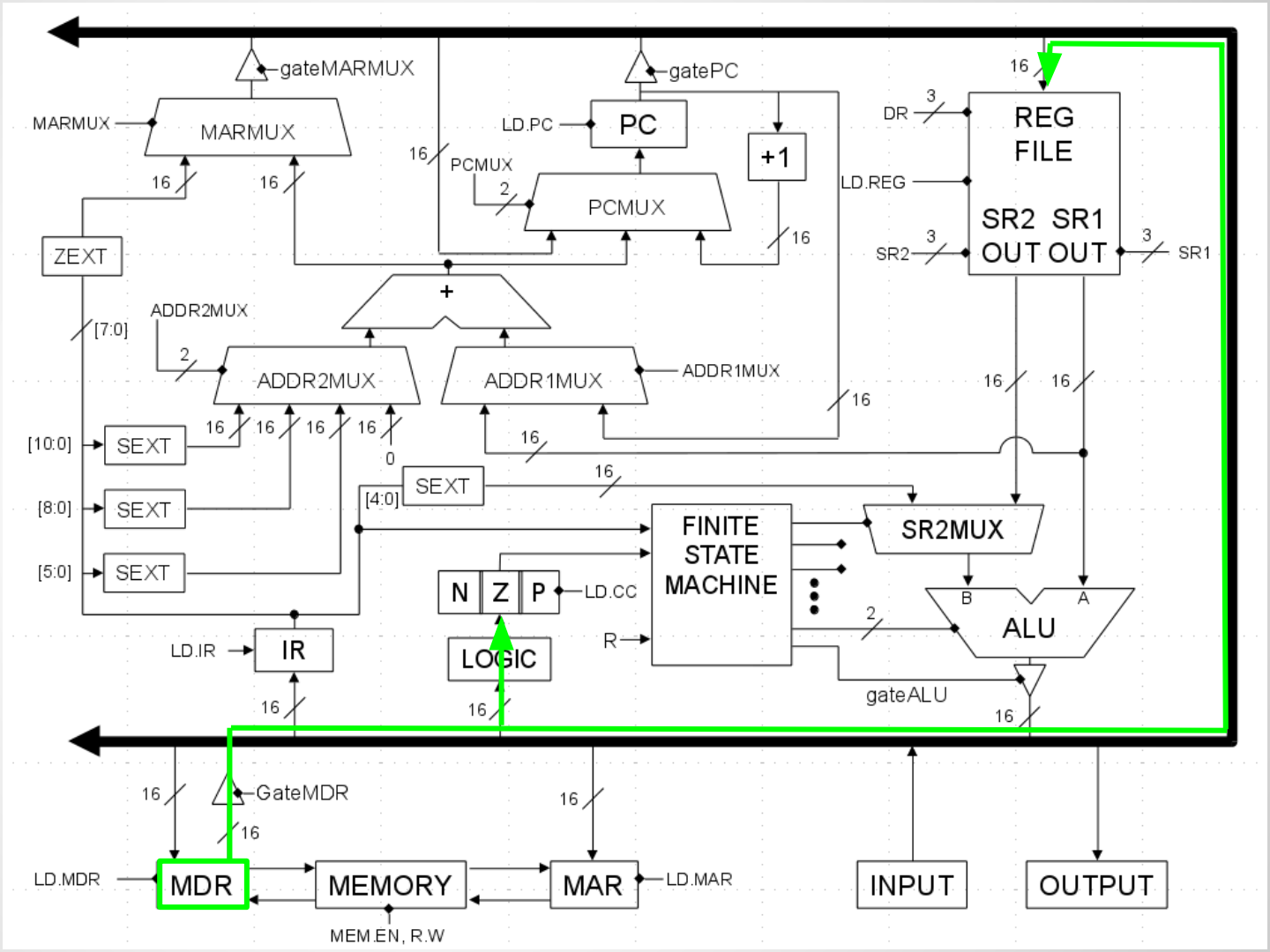
CC1:
[...]

CC2:
MEM.EN
LD.MDR

CC3:
gateMDR
LD.MAR

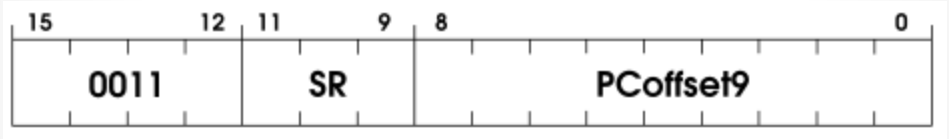
CC4:
MEM.EN
LD.MDR

CC5:
gateMDR
DR = IR[11:9]
LD.REG
LD.CC



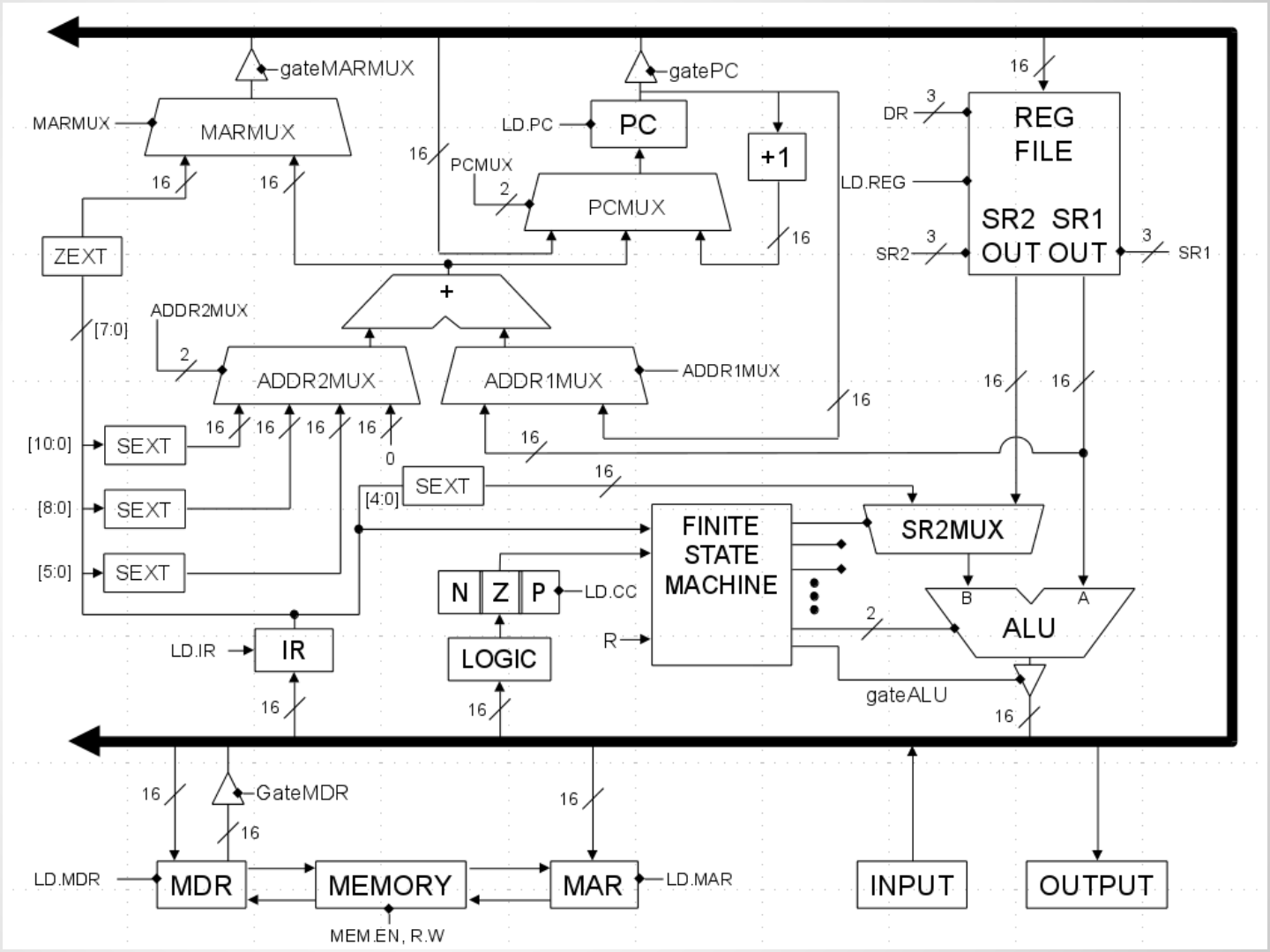
ST

The ST instruction stores a value from a register into memory, at a location near the instruction itself.



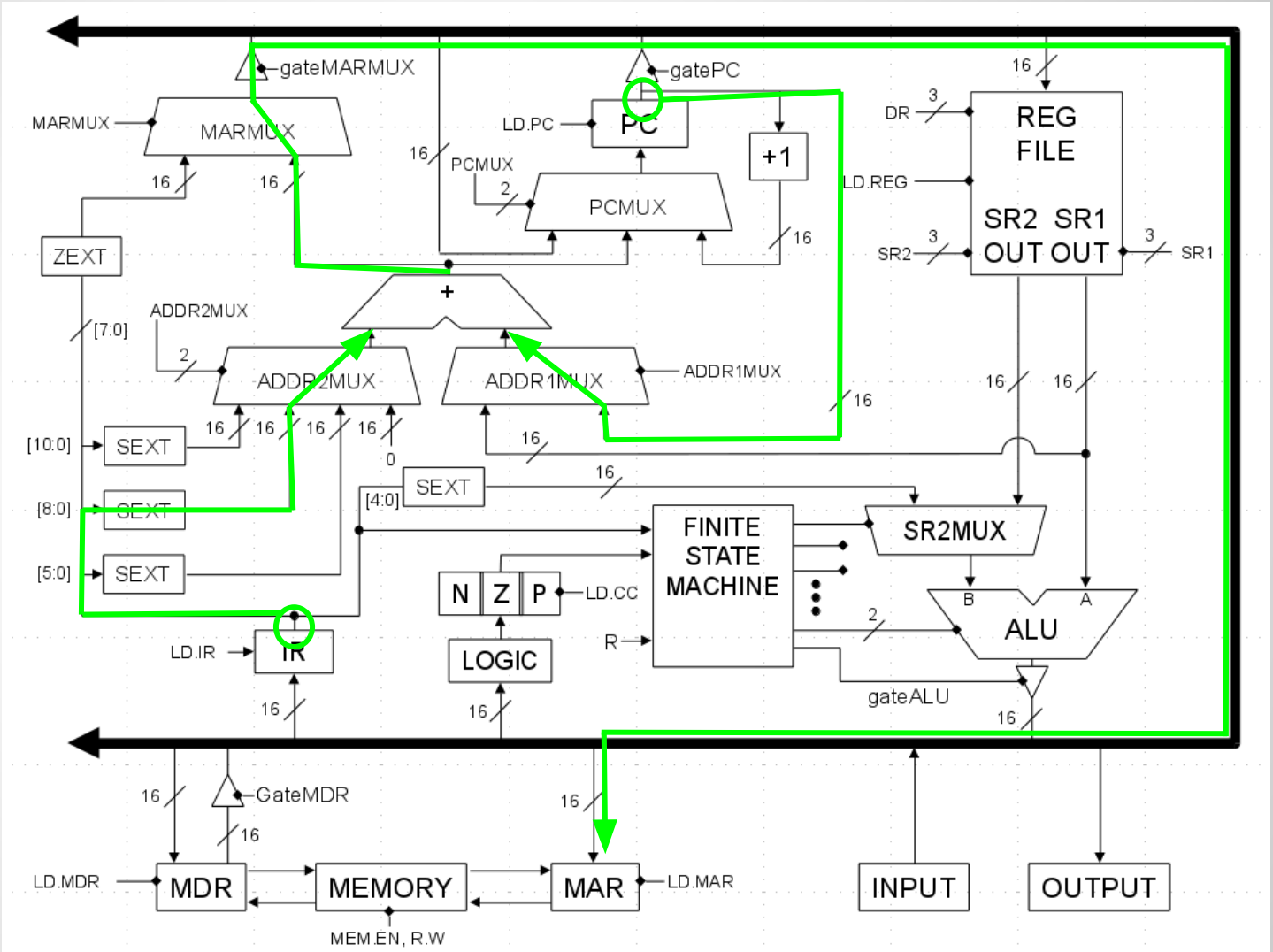
mem[PC + PCoffset9] ← SR

ST


gateMARMUX
MARMUX
MARMUX
gatePC
LD.PC
PC
+1
PCMUX
PCMUX
2
16
16
16
16
16
DR
3
REG
FILE
LD.REG
SR2 SR1
OUT OUT
SR2
3
3
SR1
ZEXT
[7:0]
+
ADDR2MUX
2
ADDR2MUX
ADDR1MUX
ADDR1MUX
16
16
16
16
16
16
16
16
[10:0]
SEXT
[8:0]
SEXT
[5:0]
SEXT
0
[4:0]
SEXT
16
FINITE
STATE
MACHINE
SR2MUX
N
Z
P
LD.CC
B
A
2
ALU
LD.IR
IR
LOGIC
R
gateALU
16
16
16
16
GateMDR
16
16
16
LD.MDR
MDR
MEMORY
MAR
LD.MAR
INPUT
OUTPUT
MEM.EN, R.W

ST

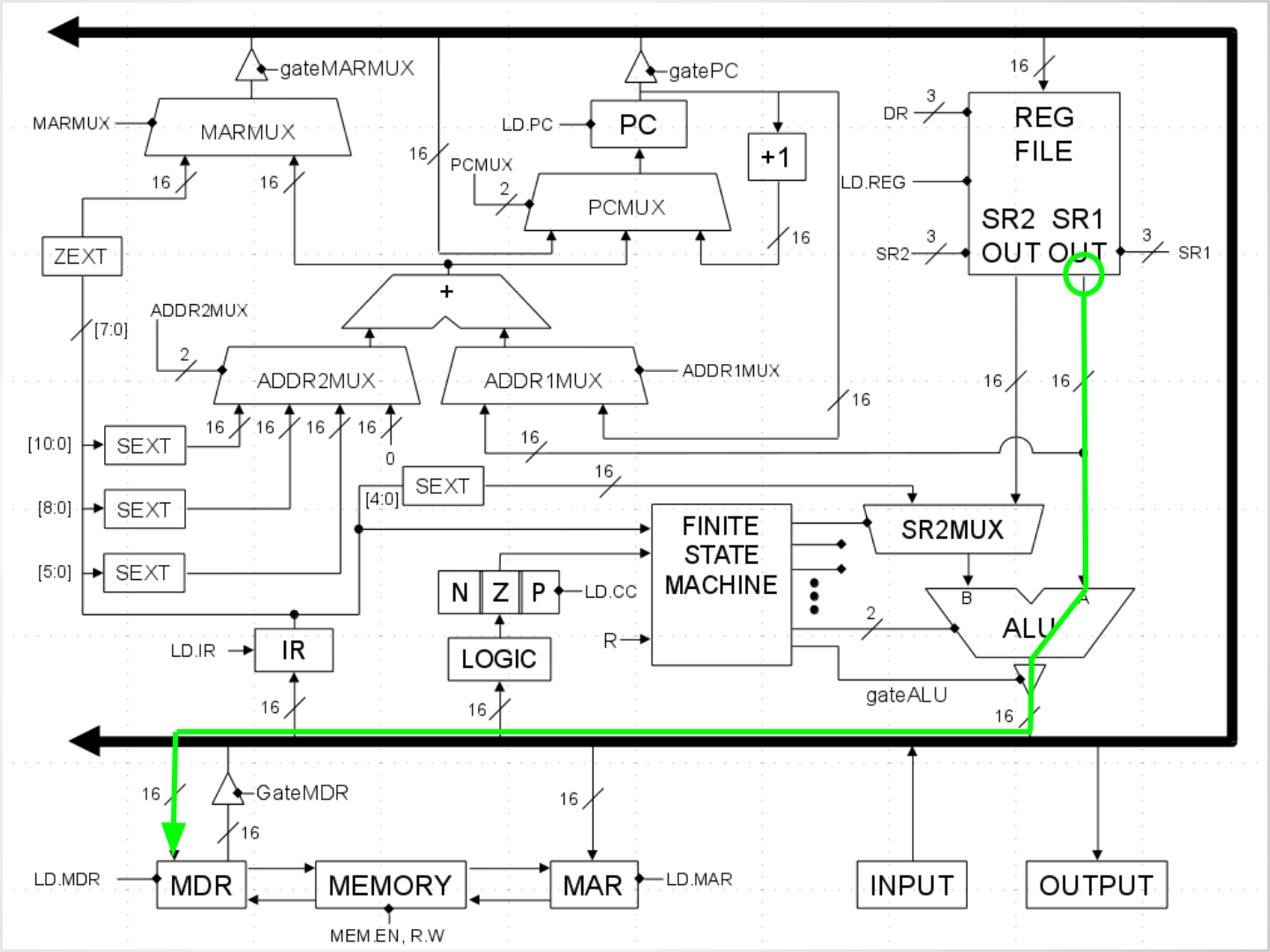
CC1:
ADDR1MUX = PC
ADDR2MUX =
SEXT[8:0]
MARMUX = ADDER
gateMARMUX
LD.MAR



ST

CC1:
ADDR1MUX = PC
ADDR2MUX =
SEXT[8:0]
MARMUX = ADDER
gateMARMUX
LD.MAR

CC2:
SR1 = IR[11:9]
ALUK = PASSA
gateALU
LD.MDR

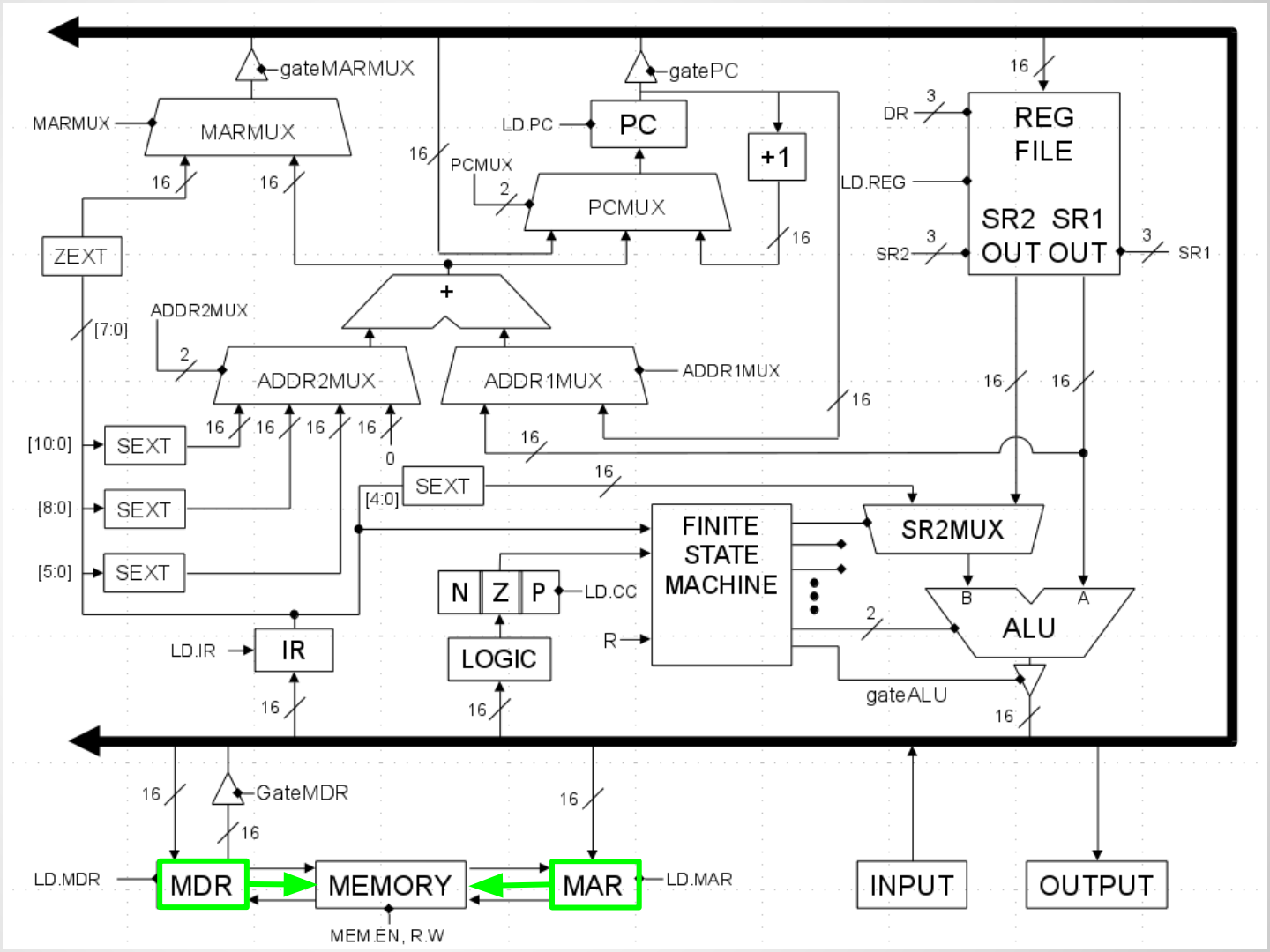


ST

CC1:
ADDR1MUX = PC
ADDR2MUX =
SEXT[8:0]
MARMUX = ADDER
gateMARMUX
LD.MAR

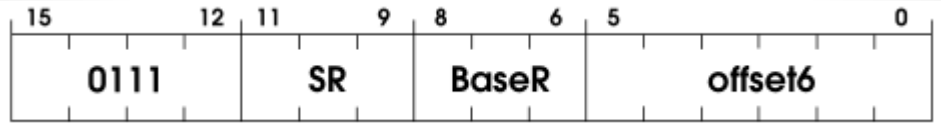
CC2:
SR1 = IR[11:9]
ALUK = PASSA
gateALU
LD.MDR

CC3:
MEM.EN
R.W



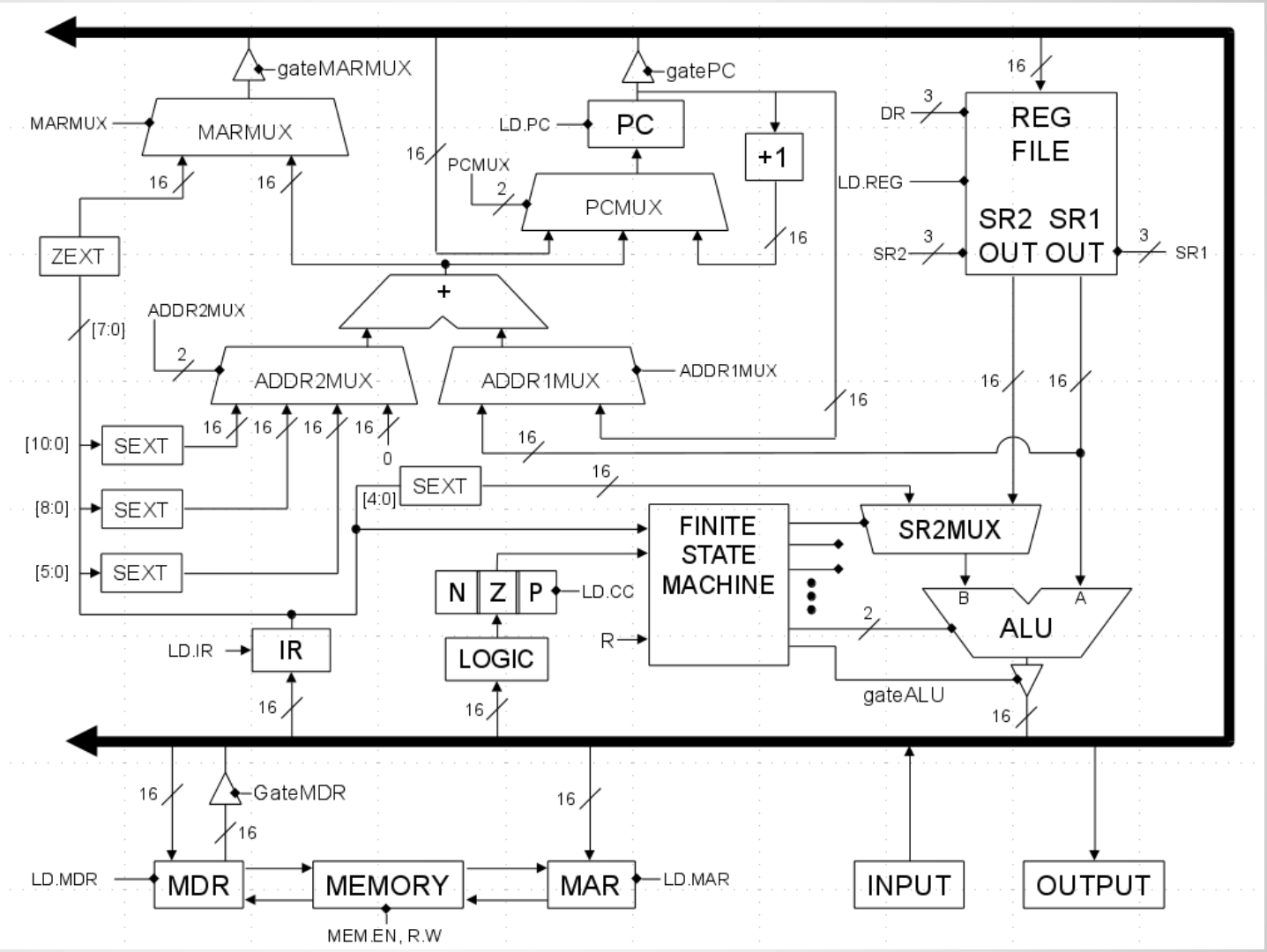
STR

The STR instruction takes a location pointed to by a base register and offset from that location, and stores the value from a source register at that location.



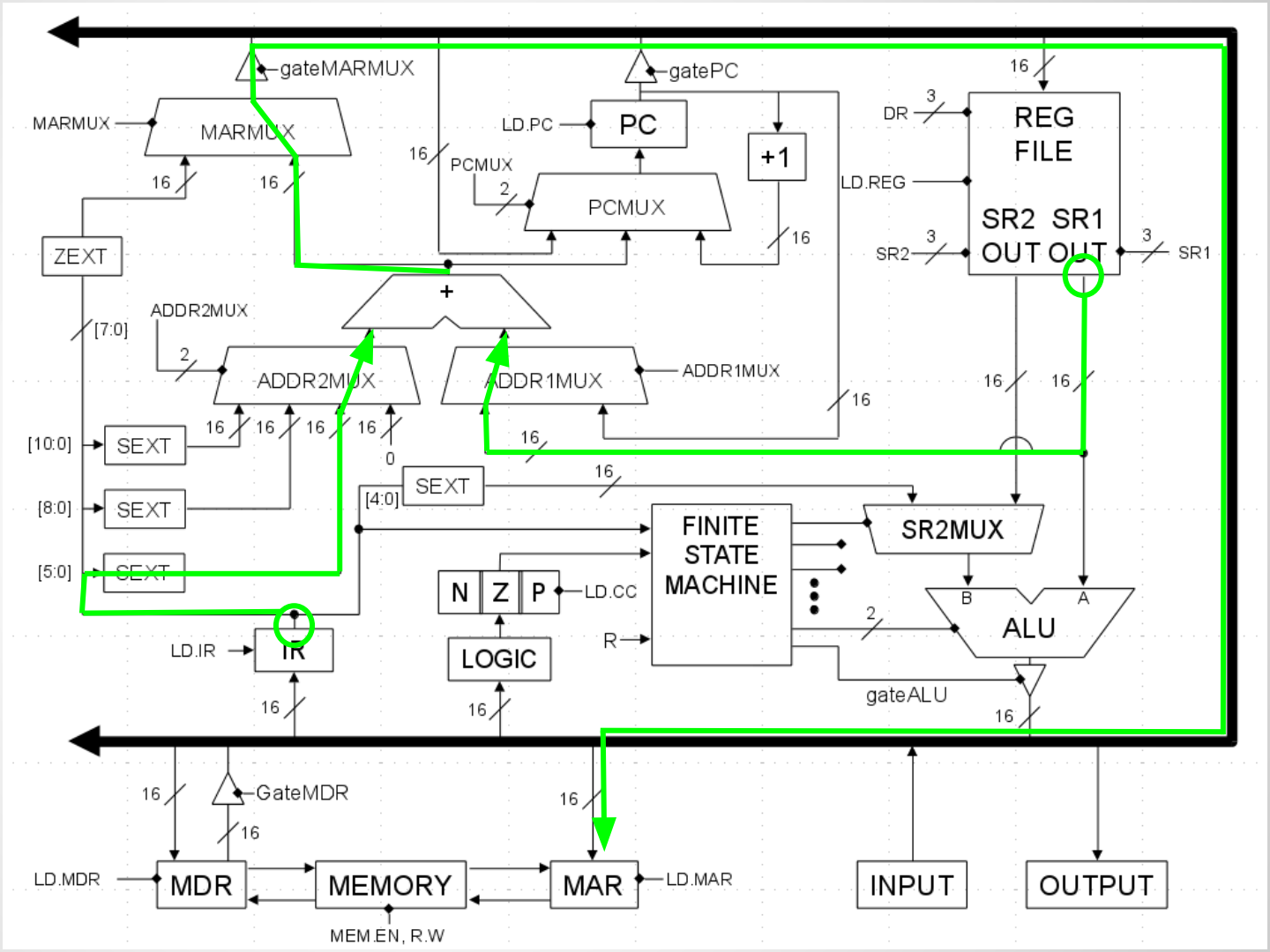
mem[BaseR + offset6] ← SR

STR


gateMARMUX
MARMUX
MARMUX
gatePC
LD.PC
PC
+1
PCMUX
PCMUX
2
16
16
16
16
16
DR
3
REG
FILE
LD.REG
SR2 SR1
OUT OUT
SR2
3
3
SR1
ZEXT
[7:0]
+
ADDR2MUX
2
ADDR2MUX
ADDR1MUX
ADDR1MUX
16
16
16
16
16
16
16
16
[10:0]
SEXT
[8:0]
SEXT
[5:0]
SEXT
0
SEXT
[4:0]
16
FINITE
STATE
MACHINE
SR2MUX
N
Z
P
LD.CC
LOGIC
R
2
B
A
ALU
LD.IR
IR
gateALU
16
16
16
16
GateMDR
16
16
16
LD.MDR
MDR
MEMORY
MAR
LD.MAR
INPUT
OUTPUT
MEM.EN, R.W

STR

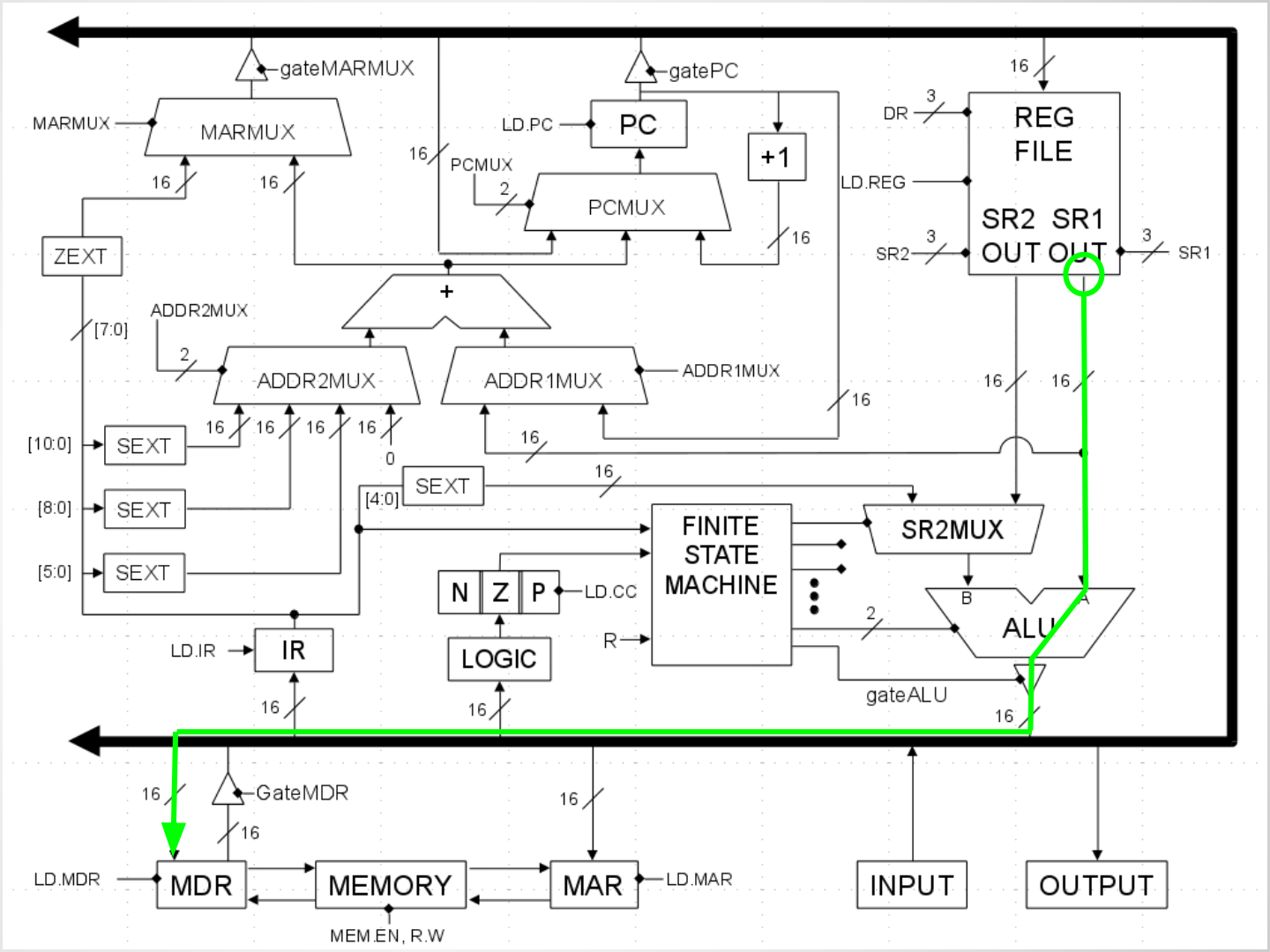
CC1:
ADDR1MUX = BaseR
ADDR2MUX =
SEXT[5:0]
MARMUX = ADDER
gateMARMUX
LD.MAR



STR

CC1:
ADDR1MUX = BaseR
ADDR2MUX =
SEXT[5:0]
MARMUX = ADDER
gateMARMUX
LD.MAR

CC2:
SR1 = IR[11:9]
ALUK = PASSA
gateALU
LD.MDR

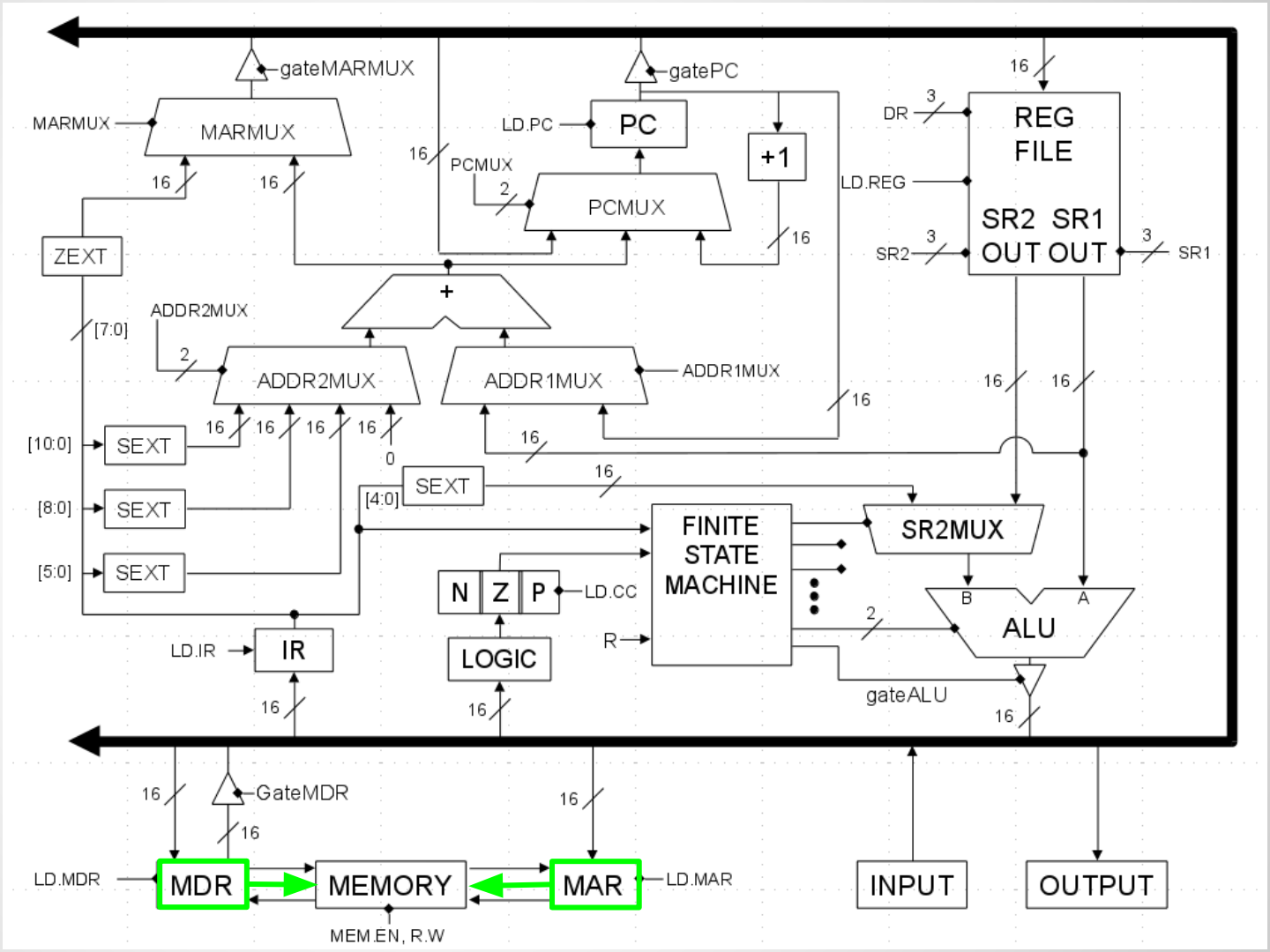


STR

CC1:
ADDR1MUX = BaseR
ADDR2MUX =
SEXT[5:0]
MARMUX = ADDER
gateMARMUX
LD.MAR

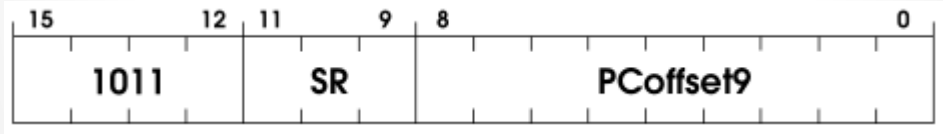
CC2:
SR1 = IR[11:9]
ALUK = PASSA
gateALU
LD.MDR

CC3:
MEM.EN
R.W



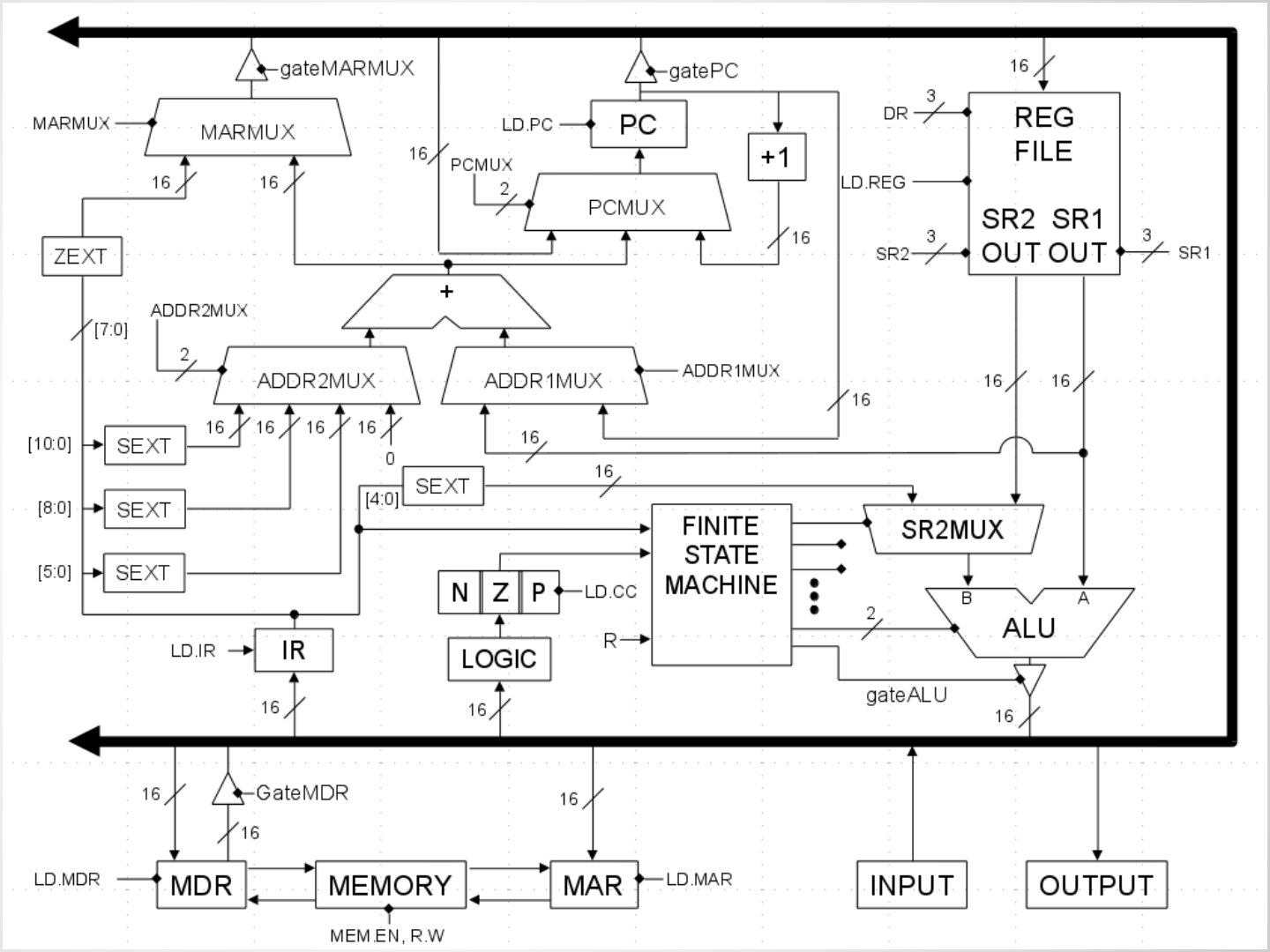
STI

The STI instruction takes a value from memory nearby the instruction itself, gets a pointer to that memory location, then stores the contents of a source register at that location.



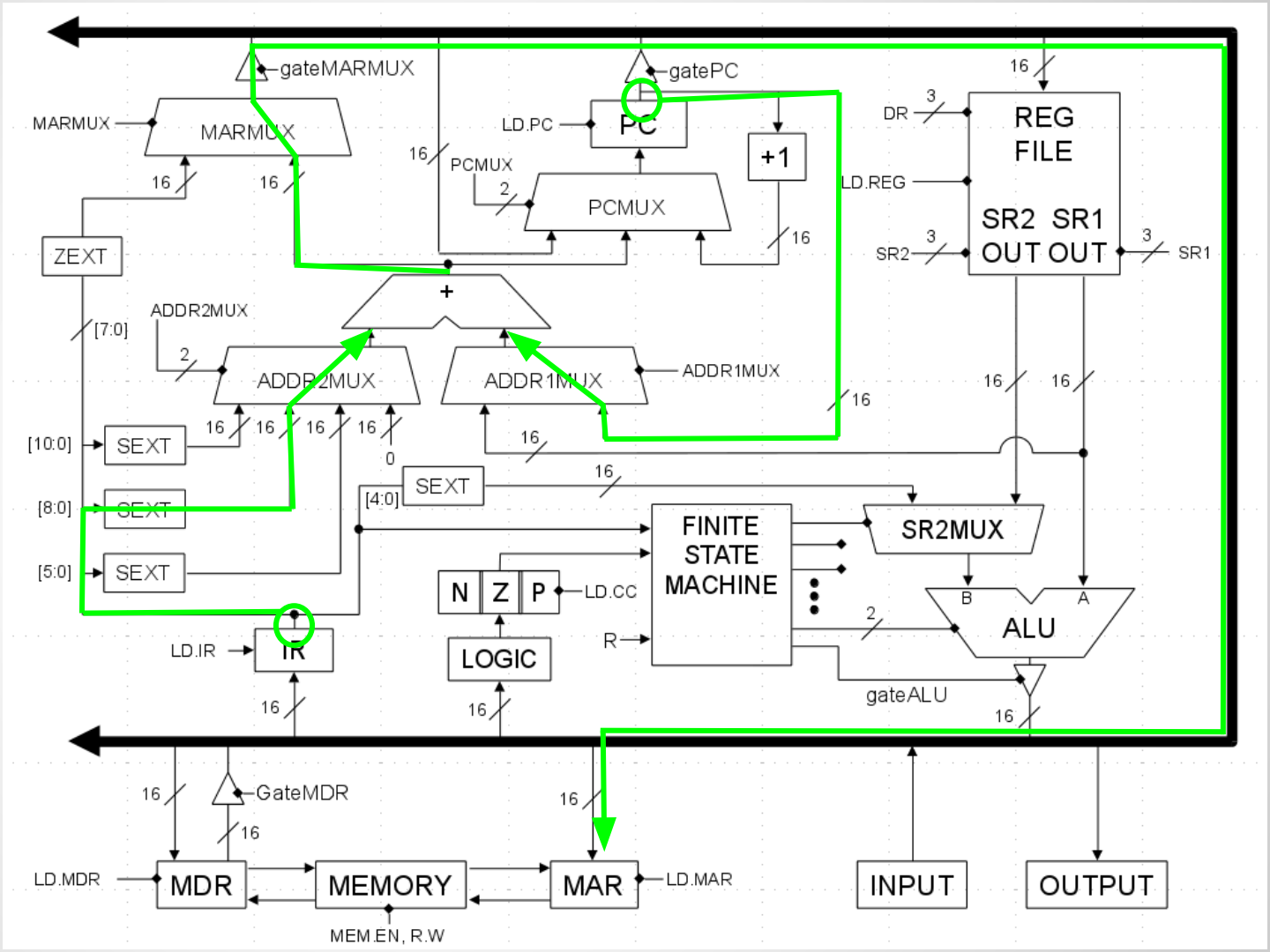
mem[mem[PC + PCoffset9]] ← SR

STI


gateMARMUX
MARMUX
MARMUX
gatePC
LD.PC
PC
+1
PCMUX
PCMUX
2
16
16
16
16
16
DR
3
REG
FILE
LD.REG
SR2 SR1
OUT OUT
SR2
3
3
SR1
ZEXT
[7:0]
+
ADDR2MUX
2
ADDR2MUX
ADDR1MUX
ADDR1MUX
16
16
16
[10:0]
SEXT
16
16
16
16
0
[8:0]
SEXT
[5:0]
SEXT
[4:0]
SEXT
16
16
16
FINITE
STATE
MACHINE
SR2MUX
N
Z
P
LD.CC
R
2
B
A
ALU
LD.IR
IR
LOGIC
gateALU
16
16
16
16
GateMDR
16
16
16
LD.MDR
MDR
MEMORY
MAR
LD.MAR
INPUT
OUTPUT
MEM.EN, R.W

STI

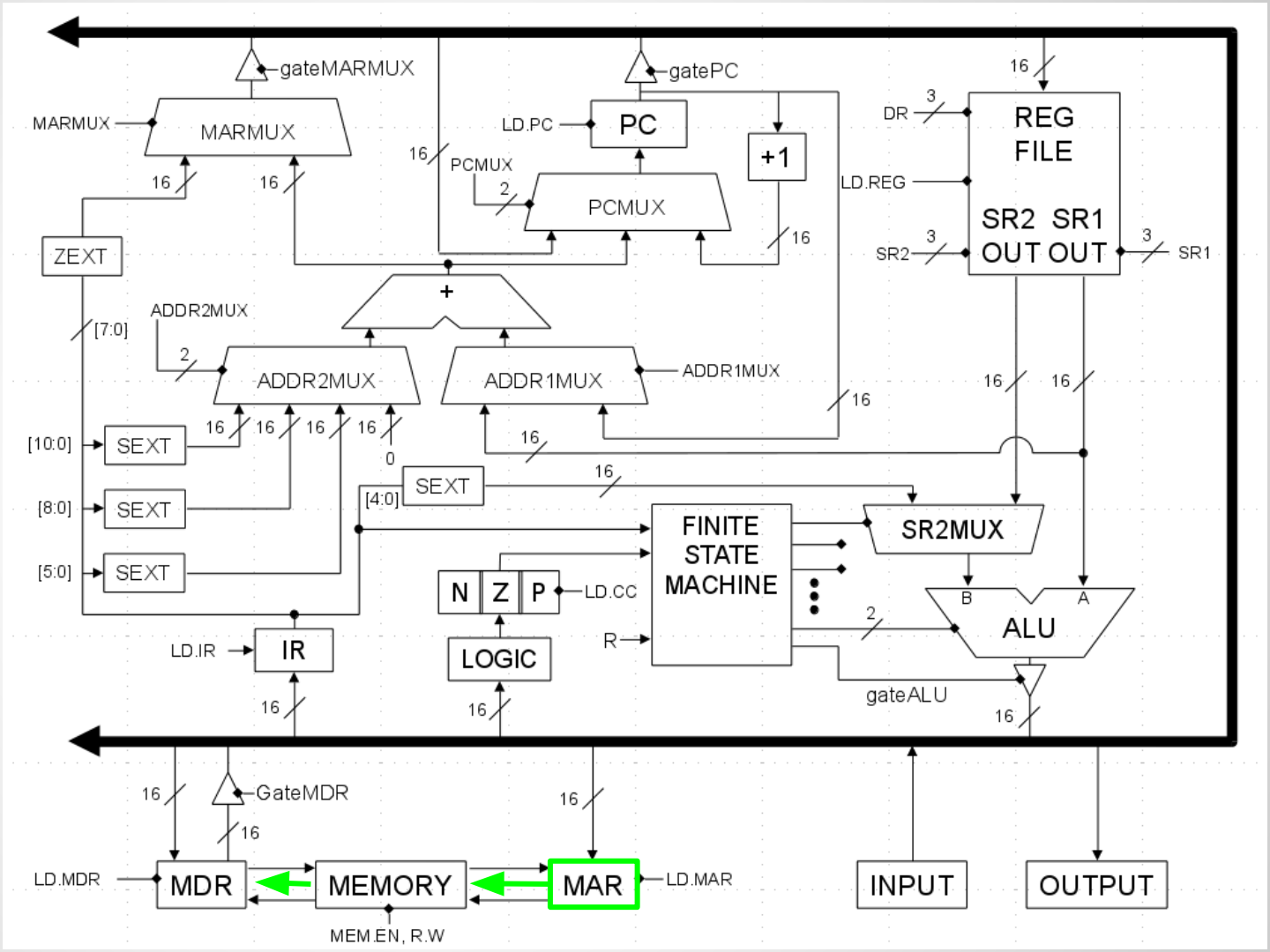
CC1:
ADDR1MUX = PC
ADDR2MUX =
    SEXT[8:0]
MARMUX = ADDER
gateMARMUX
LD.MAR



STI

CC1:
ADDR1MUX = PC
ADDR2MUX =
    SEXT[8:0]
MARMUX = ADDER
gateMARMUX
LD.MAR

CC2:
MEM.EN
LD.MDR

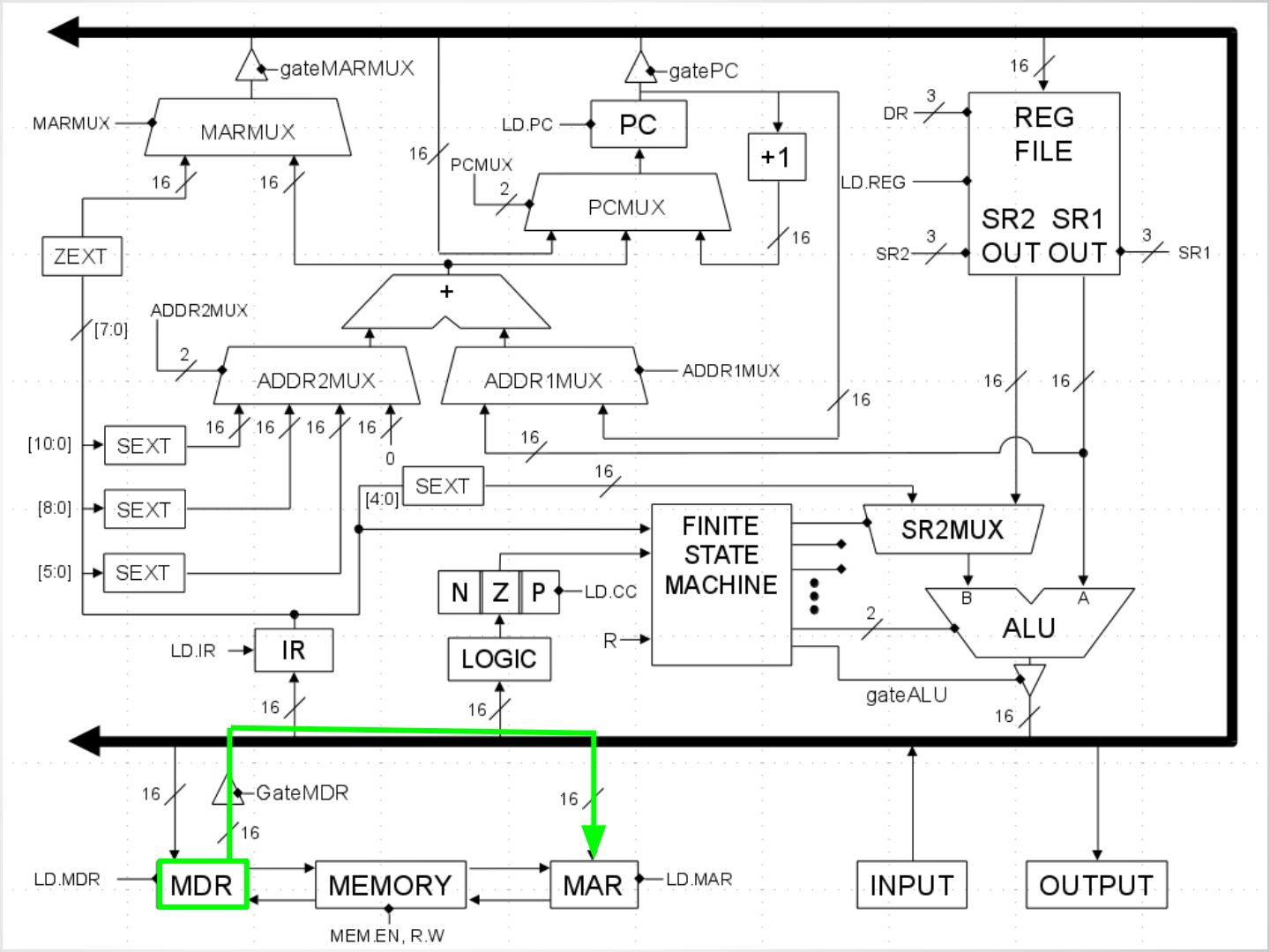


STI

CC1:
ADDR1MUX = PC
ADDR2MUX =
SEXT[8:0]
MARMUX = ADDER
gateMARMUX
LD.MAR

CC2:
MEM.EN
LD.MDR

CC3:
gateMDR
LD.MAR



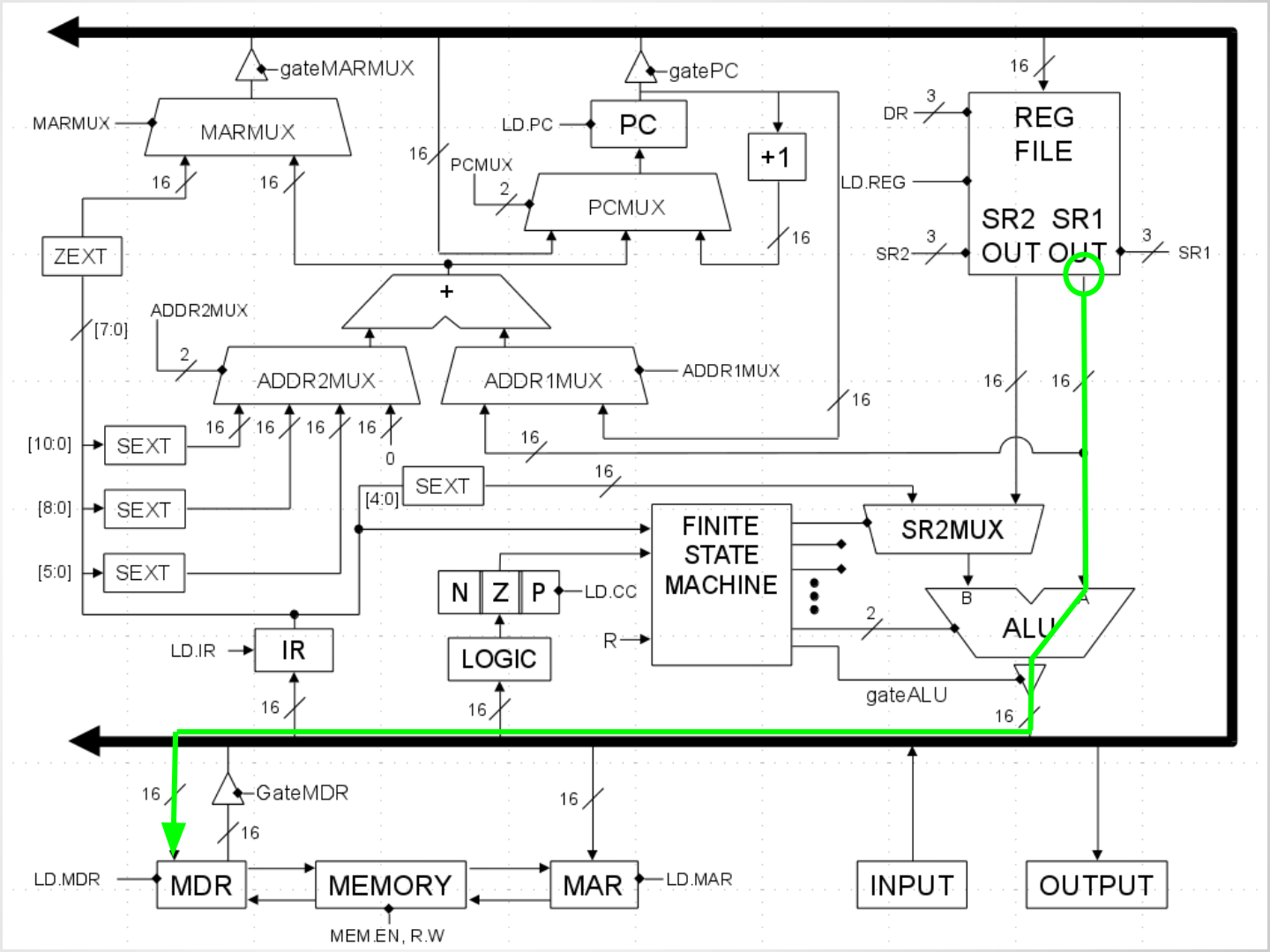
STI

CC1:
[...]

CC2:
MEM.EN
LD.MDR

CC3:
gateMDR
LD.MAR

CC4:
SR1 = IR[11:9]
ALUK = PASSA
gateALU
LD.MDR



STI

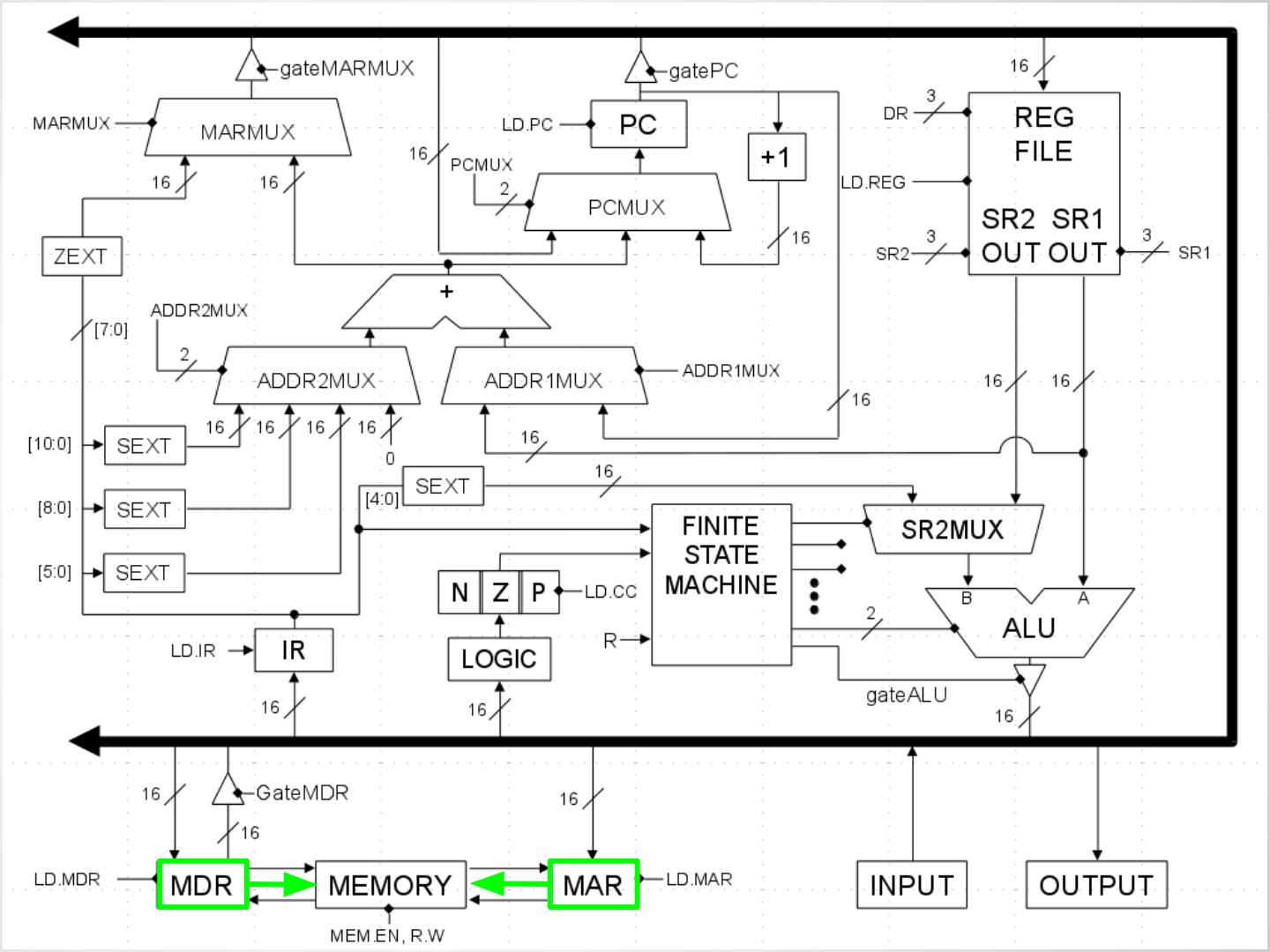
CC1:
[...]

CC2:
[...]

CC3:
gateMDR
LD.MAR

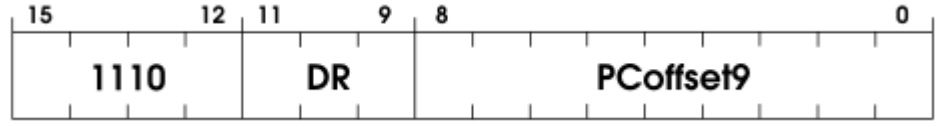
CC4:
SR1 = IR[11:9]
ALUK = PASSA
gateALU
LD.MDR

CC5:
MEM.EN
R.W



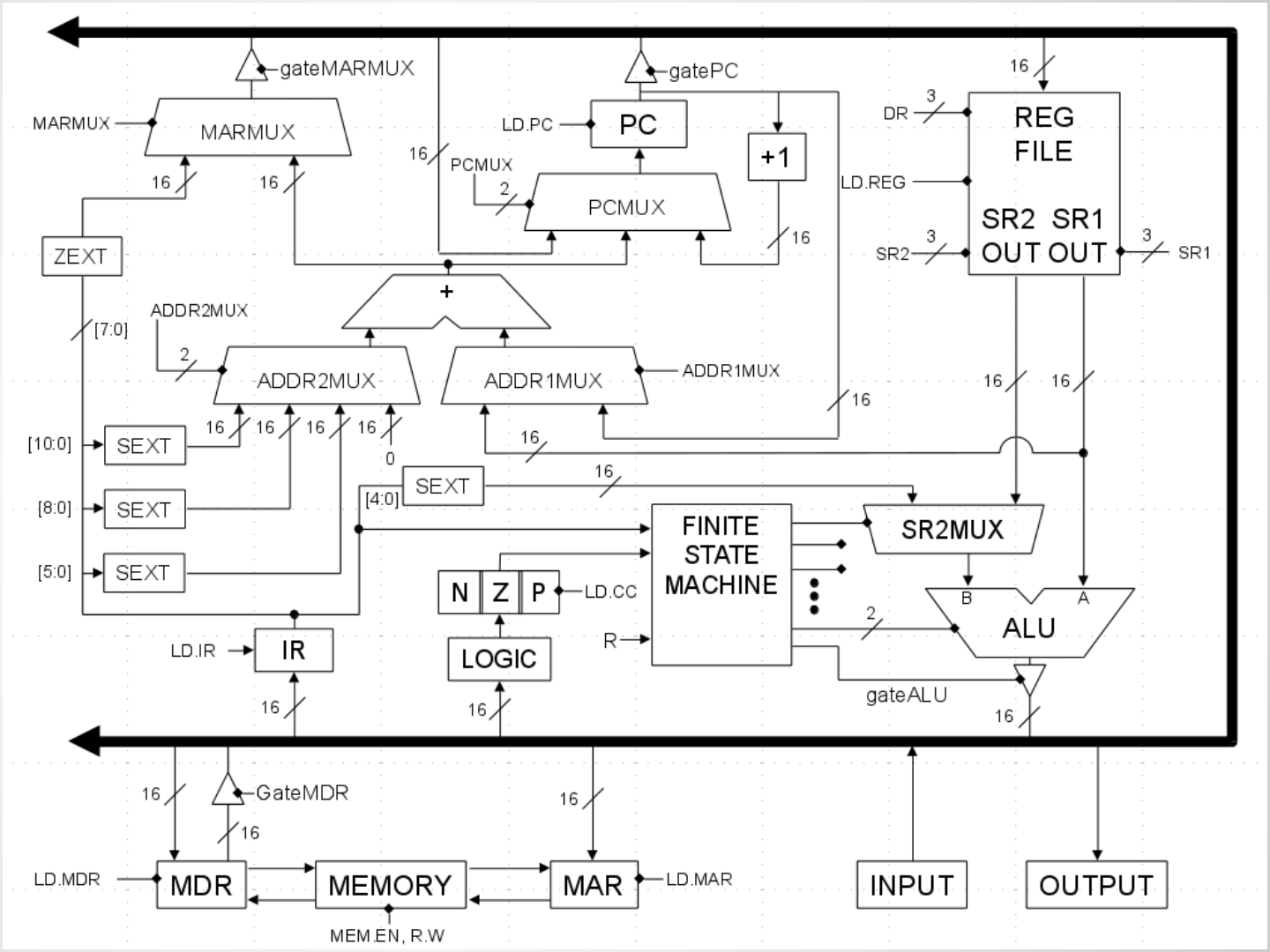
LEA

The LEA instruction adds the PC and an offset, and stores the result in a register. This is useful in determining the address of a label at run-time.



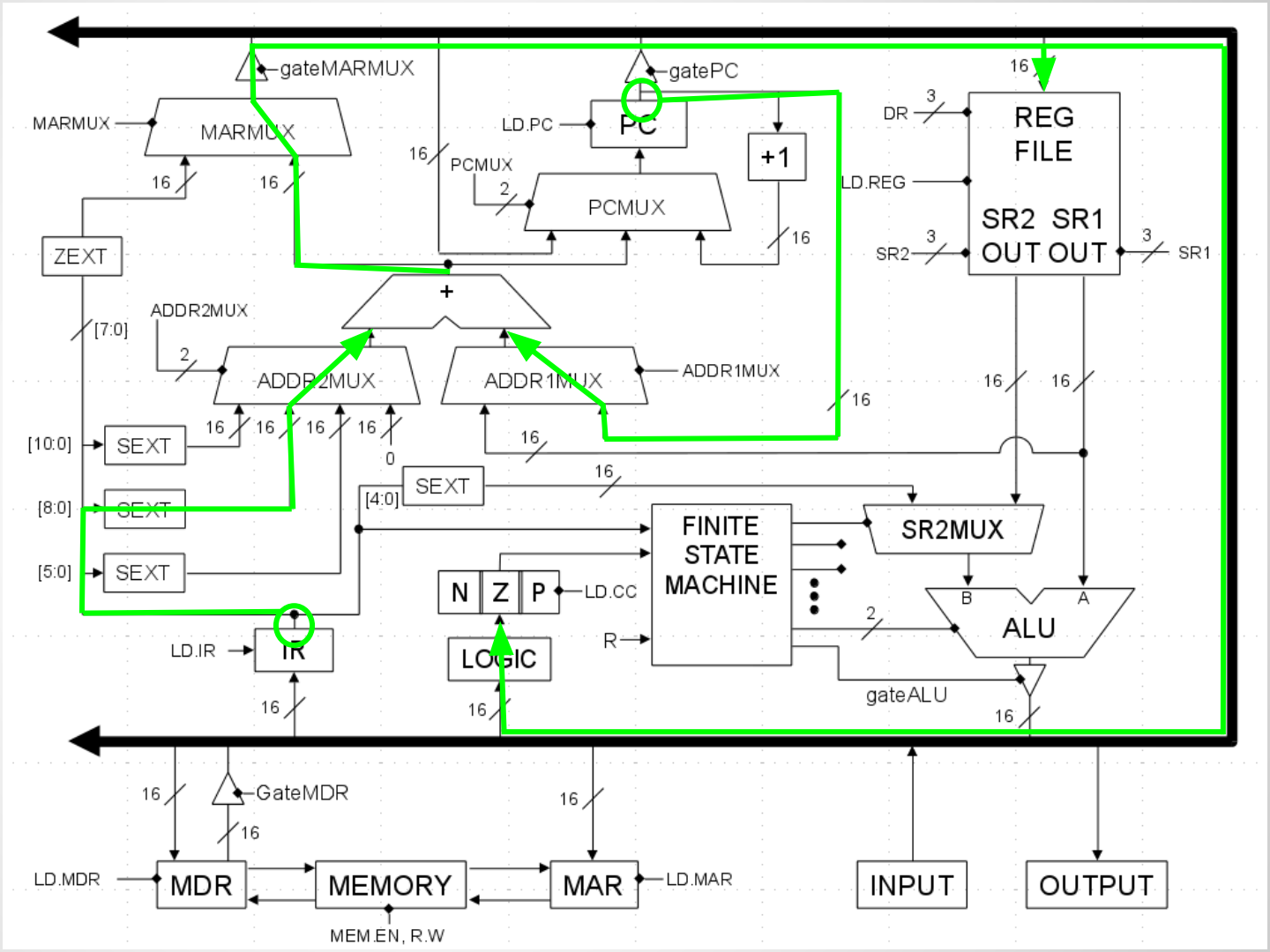
DR ← PC + PCoffset9

LEA


gateMARMUX
gatePC
MARMUX
MARMUX
LD.PC
PC
+1
PCMUX
PCMUX
2
16
16
16
16
16
DR
3
REG
FILE
LD.REG
SR2 SR1
OUT OUT
SR2
3
3
SR1
ZEXT
+
ADDR2MUX
[7:0]
2
ADDR2MUX
ADDR1MUX
ADDR1MUX
16
16
16
[10:0]
SEXT
16
16
16
16
0
[8:0]
SEXT
[4:0]
SEXT
16
16
[5:0]
SEXT
FINITE
STATE
MACHINE
SR2MUX
N
Z
P
LD.CC
B
A
2
ALU
LD.IR
IR
LOGIC
R
gateALU
16
16
16
16
GateMDR
16
16
16
LD.MDR
MDR
MEMORY
MAR
LD.MAR
INPUT
OUTPUT
MEM.EN, R.W

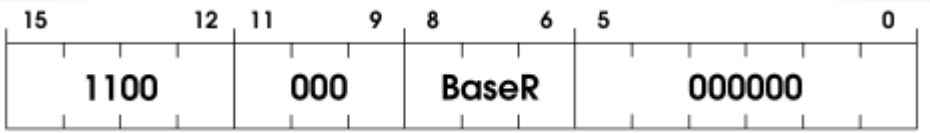
LEA

CC1:
ADDR1MUX = PC
ADDR2MUX =
SEXT[8:0]
MARMUX =
ADDER
gateMARMUX
DR = IR[11:9]
LD.REG
LD.CC



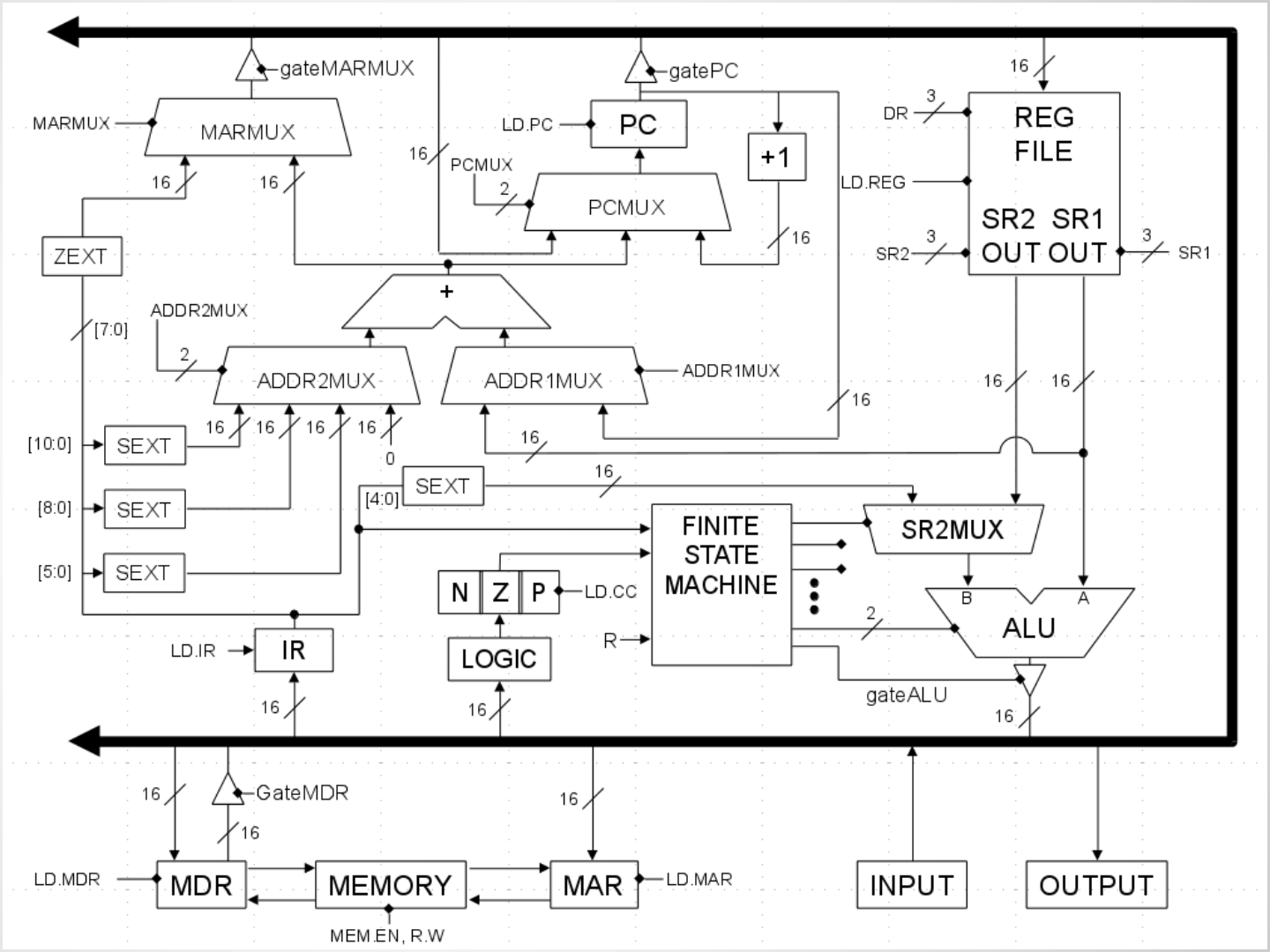
JMP/RET

The JMP instruction sets the PC to the value of a register, effectively “jumping” to the location it points to. RET is a pseudo-instruction that simply performs JMP R7.



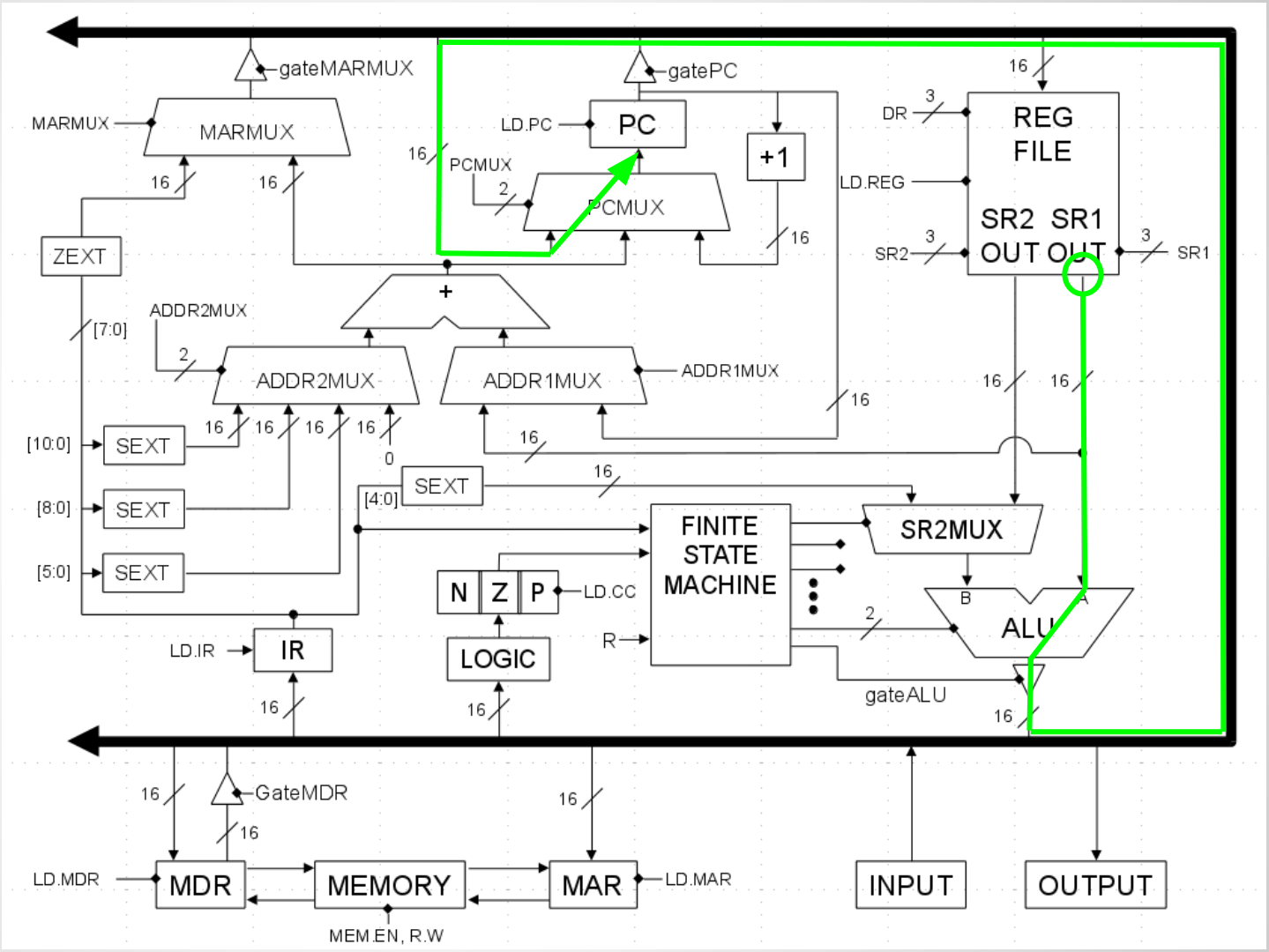
PC ← BaseR

JMP


gateMARMUX
MARMUX
MARMUX
16
16
ZEXT
[7:0]
gatePC
LD.PC
PC
+1
16
PCMUX
2
PCMUX
16
DR
3
REG
FILE
LD.REG
SR2 SR1
OUT OUT
SR2
3
3
SR1
16
+
ADDR2MUX
2
ADDR2MUX
ADDR1MUX
ADDR1MUX
16
16
16
16
16
16
16
16
0
[10:0]
SEXT
[8:0]
SEXT
[5:0]
SEXT
[4:0]
SEXT
16
FINITE
STATE
MACHINE
SR2MUX
N
Z
P
LD.CC
LOGIC
R
B
A
2
ALU
LD.IR
IR
16
16
gateALU
16
16
GateMDR
16
16
16
LD.MDR
MDR
MEMORY
MAR
LD.MAR
MEM.EN, R.W
INPUT
OUTPUT

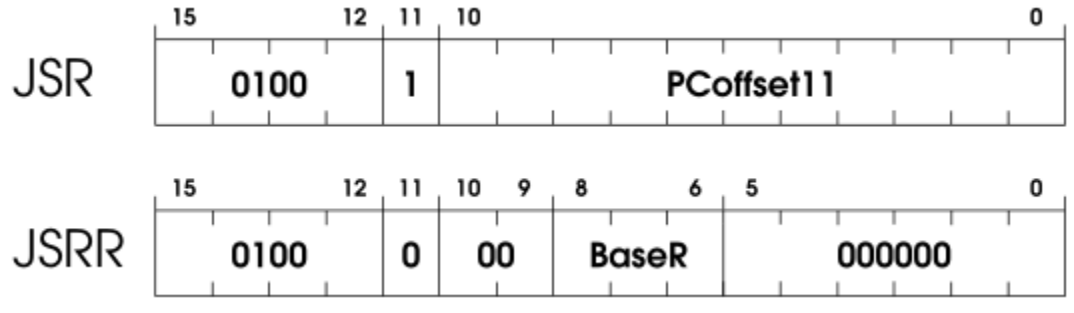
JMP

CC1:
SR1 = IR[8:6]
ALUK = PASSA
gateALU
PCMUX = BUS
LD.PC



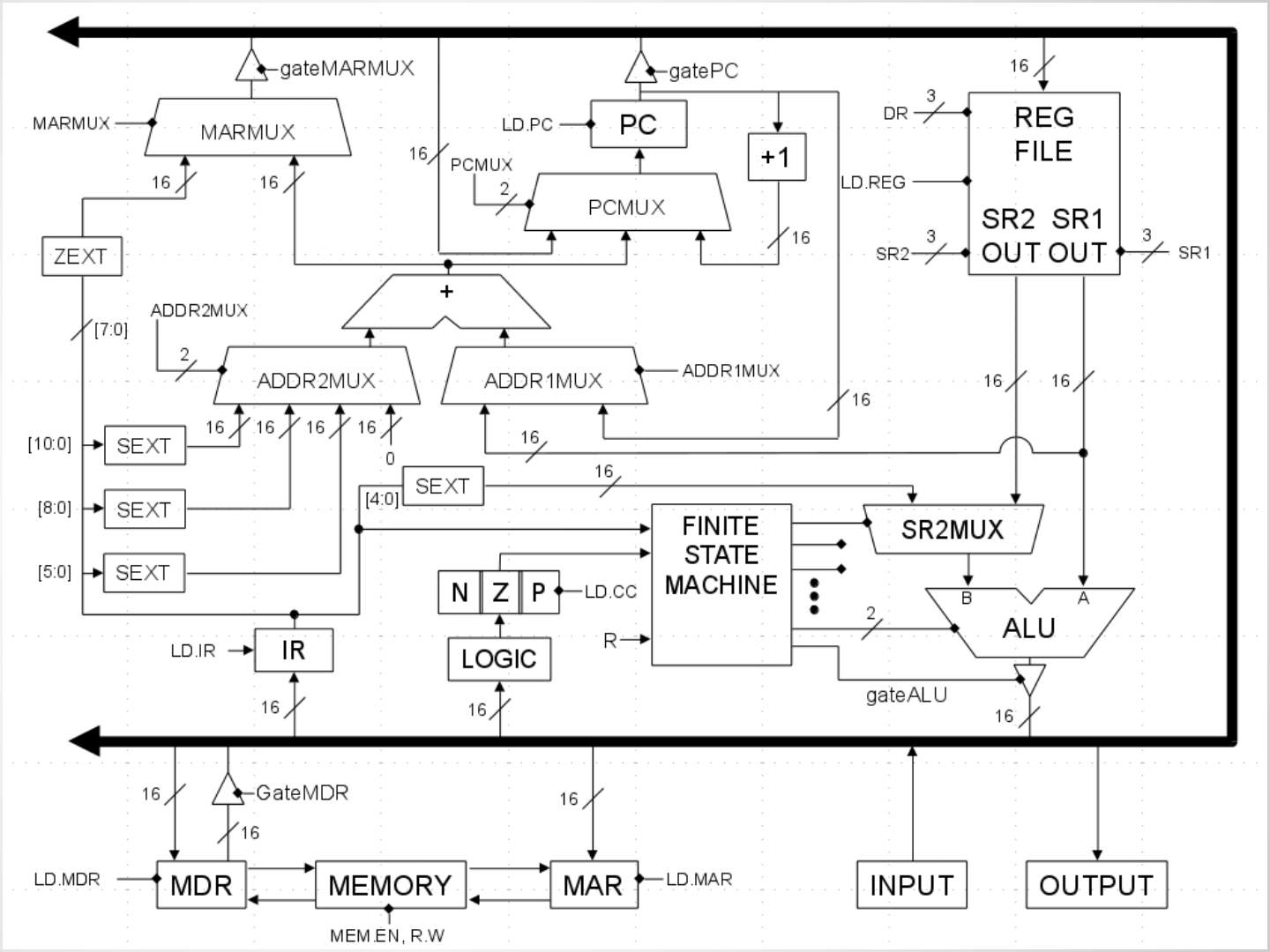
JSR/JSRR

The JSR instruction stores the current PC in R7, and sets the PC to itself + an offset; it is used to jump to a subroutine. JSRR does the same thing, but instead uses a base register instead of an offset.



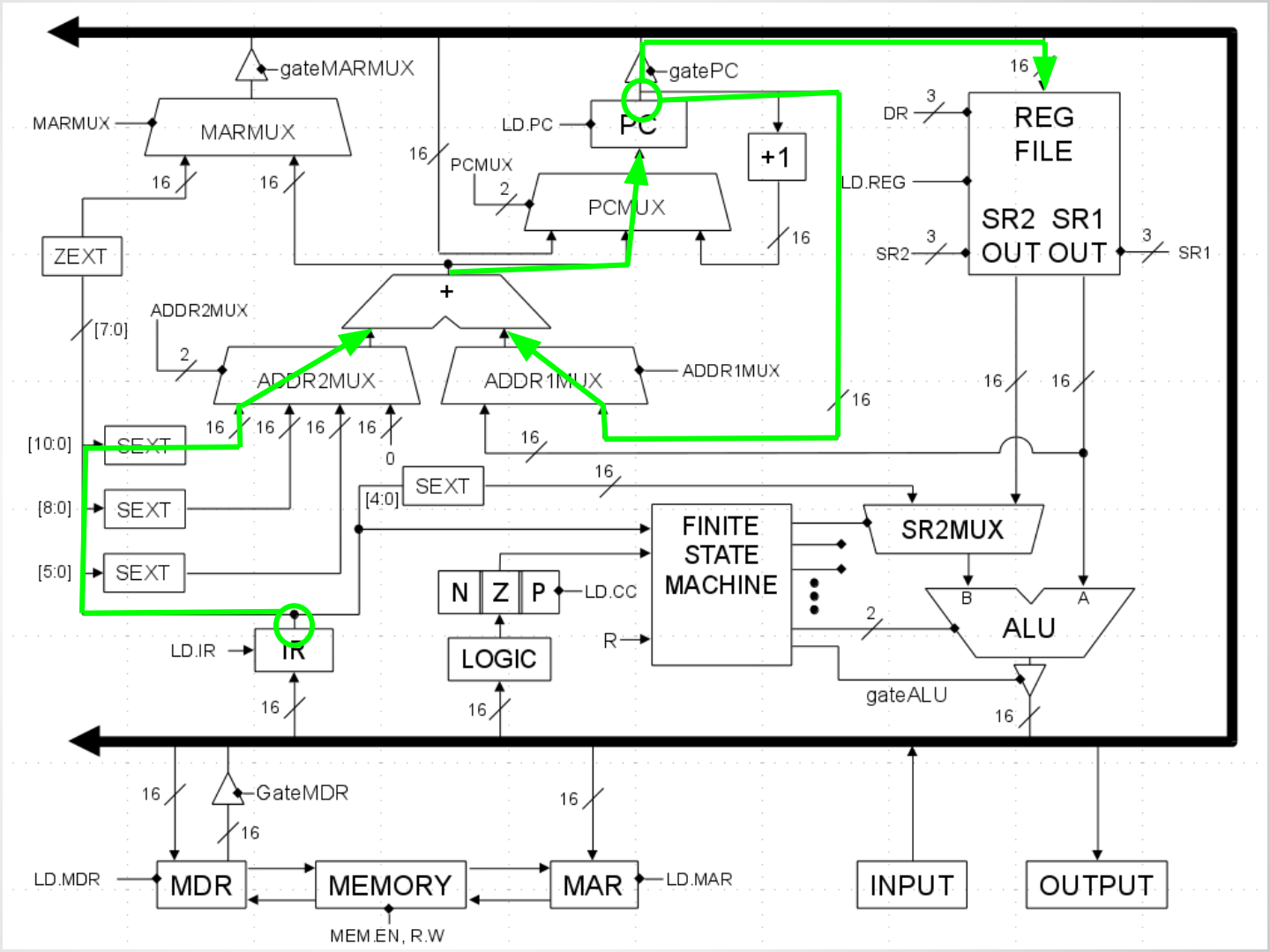
R7 = PC; PC ← (IR[11] == 1 ? PCoffset11 : BaseR)

JSR

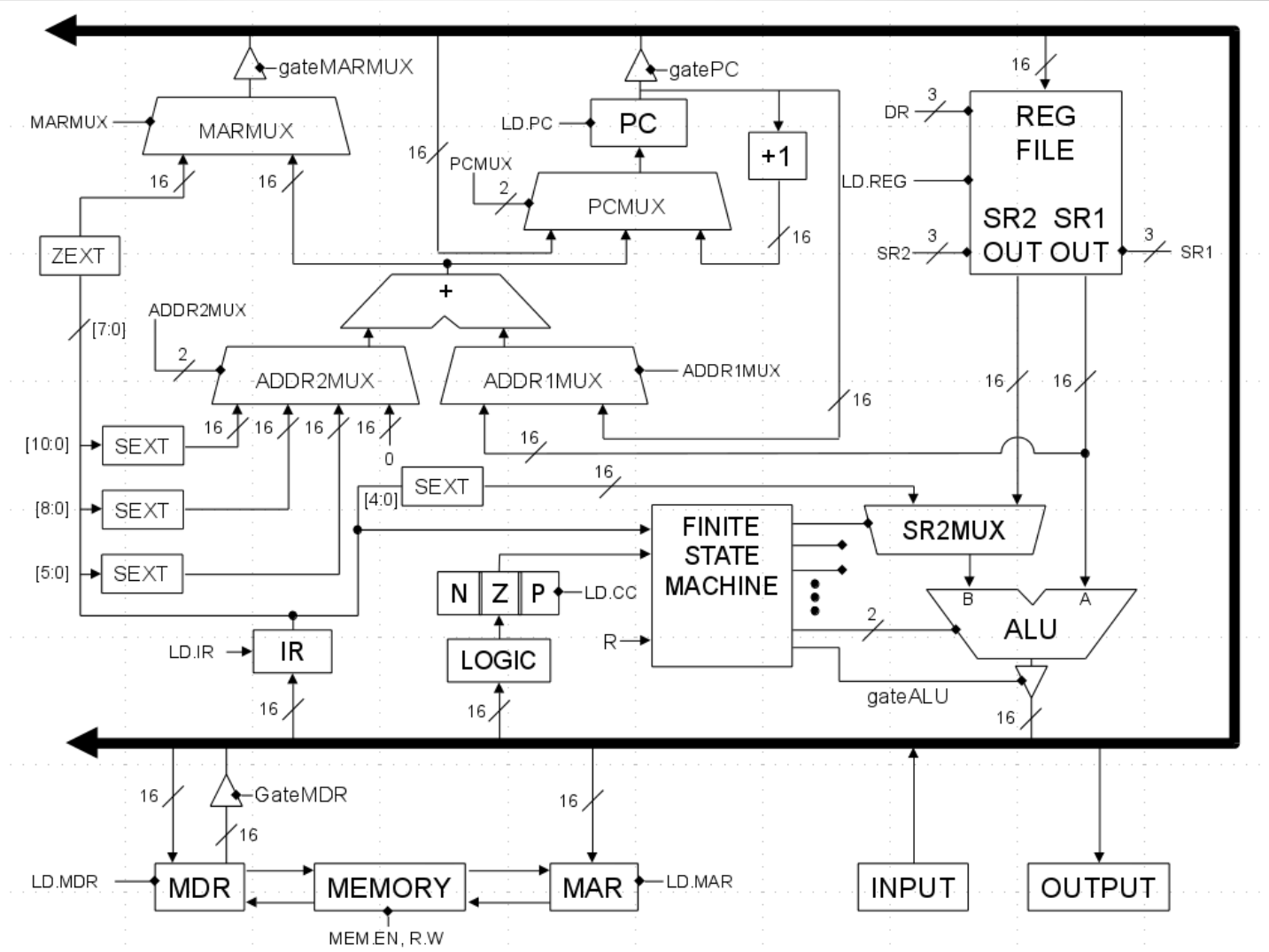

gateMARMUX
gatePC
MARMUX
MARMUX
LD.PC
PC
+1
PCMUX
PCMUX
2
16
16
16
16
16
DR
3
REG
FILE
LD.REG
SR2 SR1
OUT OUT
SR2
3
3
SR1
ZEXT
+
ADDR2MUX
[7:0]
2
ADDR2MUX
ADDR1MUX
ADDR1MUX
16
16
16
16
16
16
16
16
16
16
[10:0]
SEXT
[8:0]
SEXT
[5:0]
SEXT
0
SEXT
[4:0]
16
FINITE
STATE
MACHINE
SR2MUX
N
Z
P
LD.CC
B
A
ALU
2
LD.IR
IR
LOGIC
R
gateALU
16
16
16
16
GateMDR
16
16
16
LD.MDR
MDR
MEMORY
MAR
LD.MAR
INPUT
OUTPUT
MEM.EN, R.W

JSR

CC1:
gatePC
DR = 7
LD.REG
ADDR1MUX = PC
ADDR2MUX =
SEXT[10:0]
PCMUX = ADDER
LD.PC

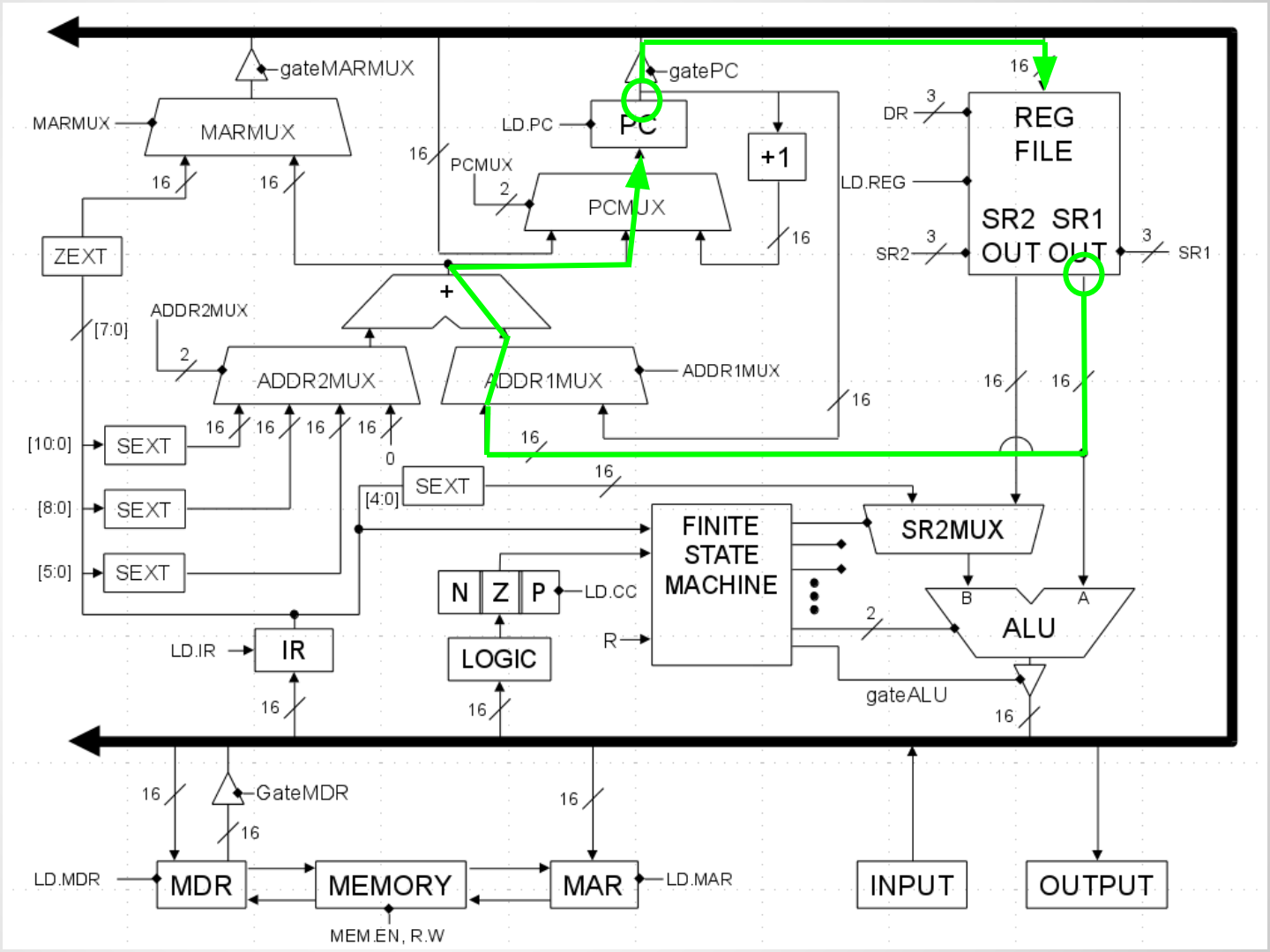


JSRR


gateMARMUX
gatePC
MARMUX
MARMUX
LD.PC
PC
+1
PCMUX
PCMUX
REG FILE
DR
LD.REG
SR2 SR1
OUT OUT
SR2
SR1
ZEXT
+
ADDR2MUX
ADDR2MUX
ADDR1MUX
ADDR1MUX
[7:0]
[10:0]
[8:0]
[5:0]
[4:0]
SEXT
SEXT
SEXT
SEXT
FINITE STATE MACHINE
SR2MUX
N
Z
P
LD.CC
LOGIC
R
ALU
B
A
LD.IR
IR
gateALU
GateMDR
LD.MDR
MDR
MEMORY
MAR
LD.MAR
MEM.EN, R.W
INPUT
OUTPUT

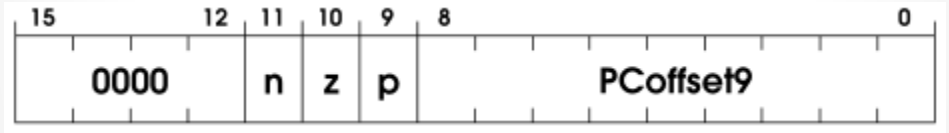
JSRR

CC1:
gatePC
DR = 7
LD.REG
SR1 = IR[8:6]
ADDR1MUX = BaseR
ADDR2MUX = 0
PCMUX = ADDER
LD.PC



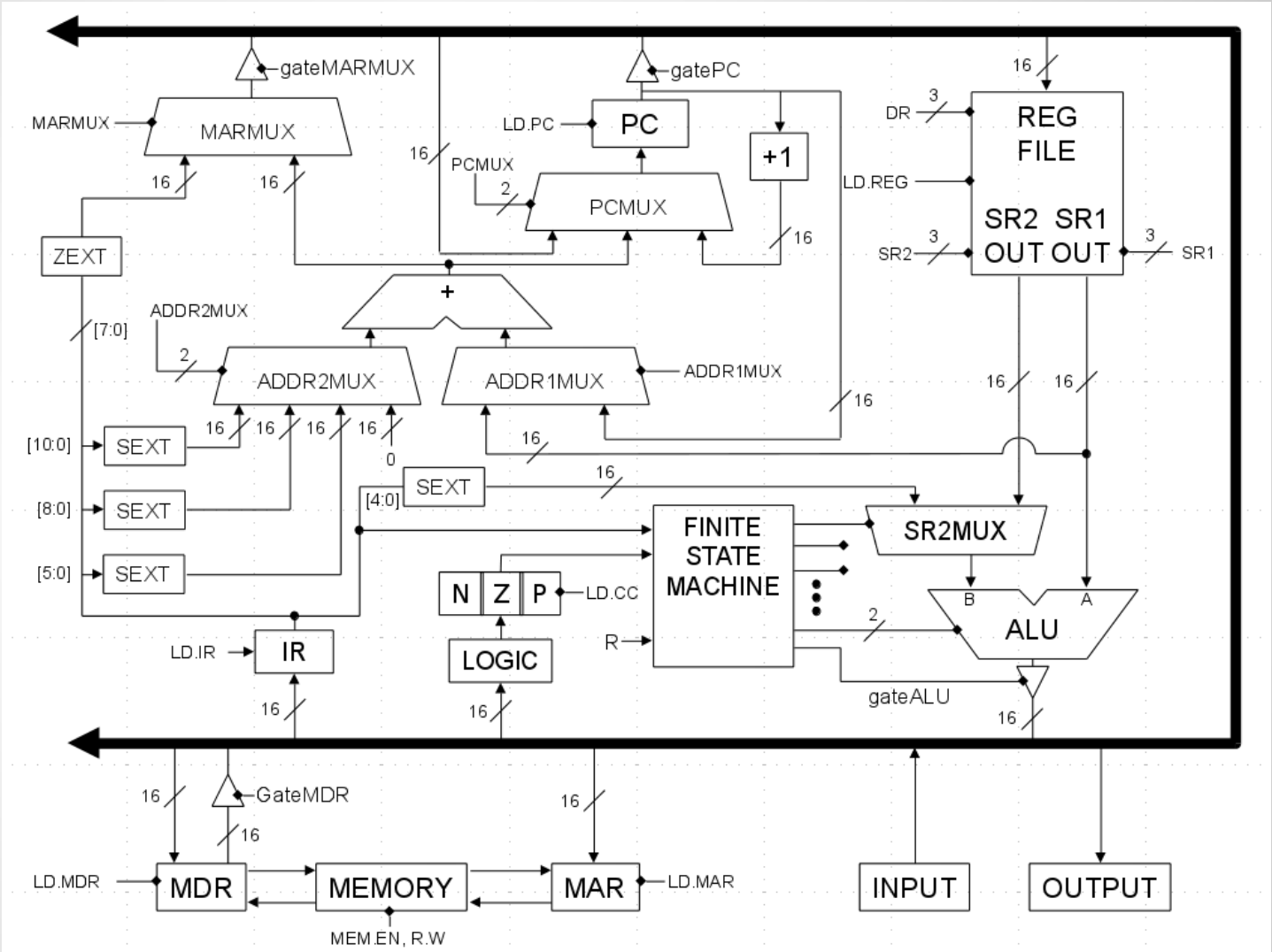
BR

The BR instruction contains 3 condition code bits; if any of the corresponding bits are set in the NZP register, add the offset to the current PC. Otherwise, do nothing.



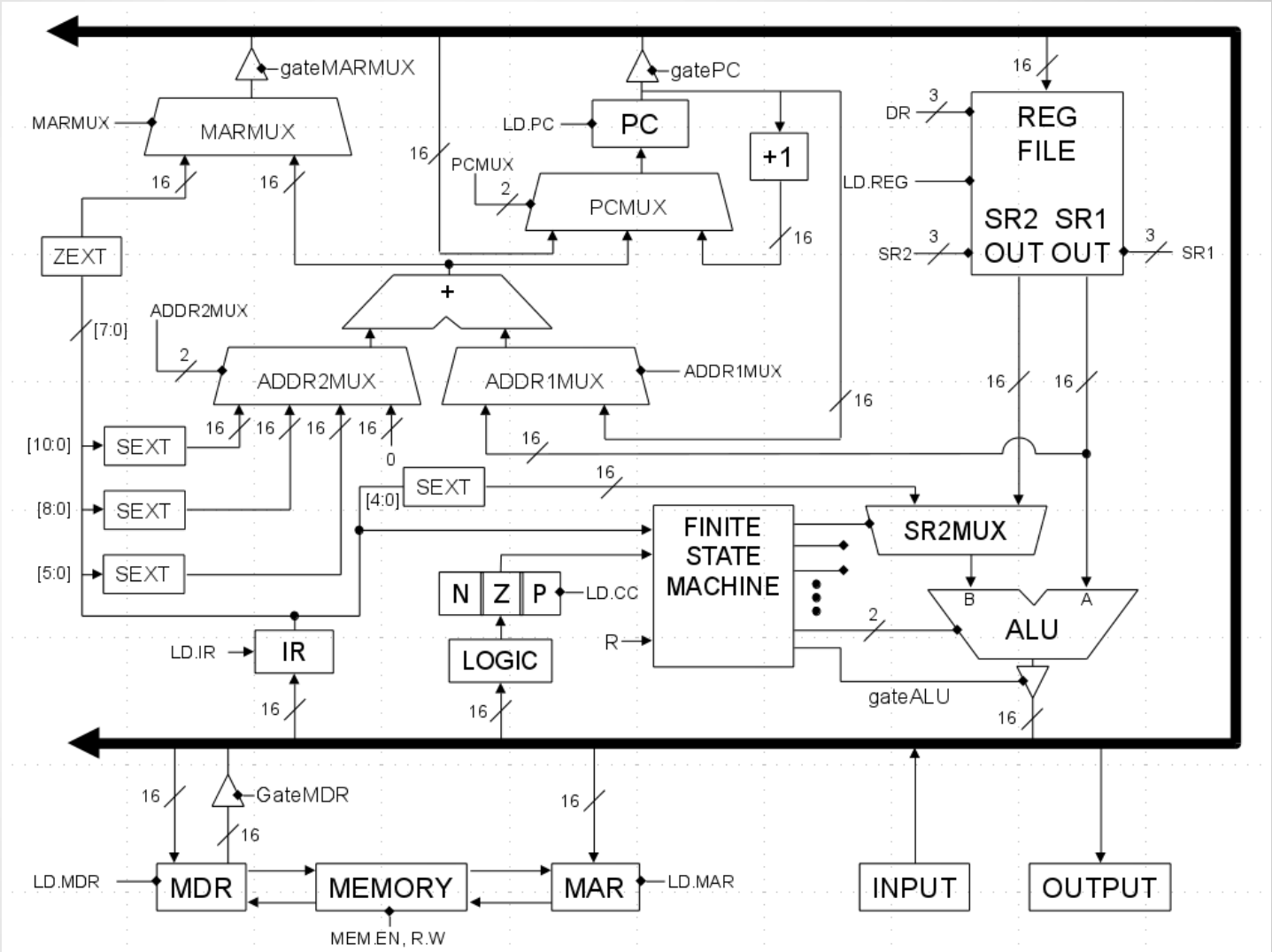

```
if(NZP & IR[11:9] != 0) {
    PC = PC + PCoffset9
}
```

BR (not taken)


gateMARMUX
MARMUX
MARMUX
gatePC
LD.PC
PC
+1
PCMUX
PCMUX
2
16
DR
3
REG
FILE
LD.REG
SR2 SR1
OUT OUT
SR2
3
3
SR1
ZEXT
[7:0]
+
ADDR2MUX
2
ADDR2MUX
ADDR1MUX
ADDR1MUX
[10:0]
SEXT
[8:0]
SEXT
[5:0]
SEXT
0
[4:0]
SEXT
FINITE
STATE
MACHINE
SR2MUX
N
Z
P
LD.CC
LOGIC
R
B
A
ALU
2
gateALU
LD.IR
IR
GateMDR
LD.MDR
MDR
MEMORY
MAR
LD.MAR
MEM.EN, R.W
INPUT
OUTPUT

BR (not taken)

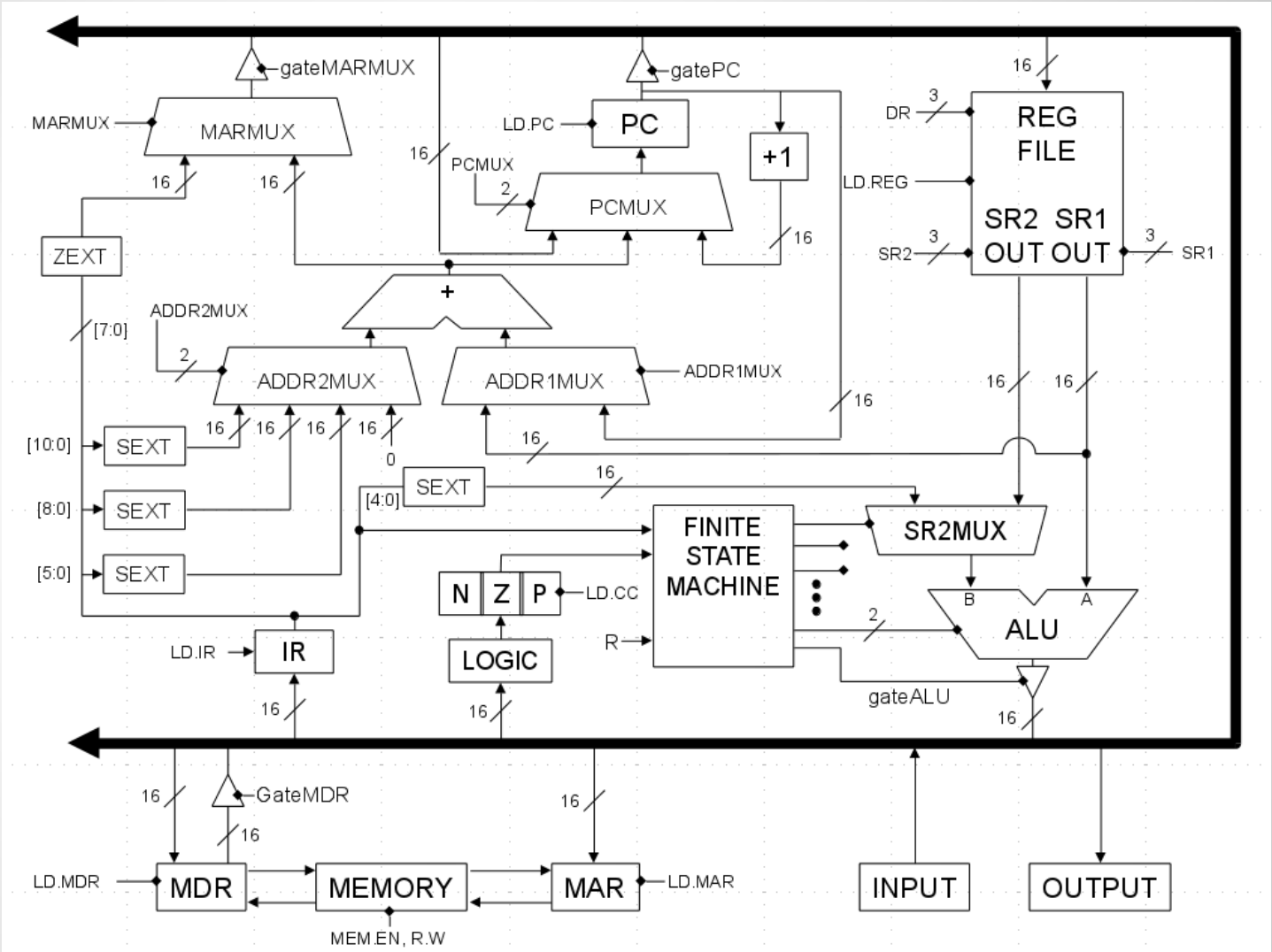
CC1:
Check NZP
(internally in FSM)



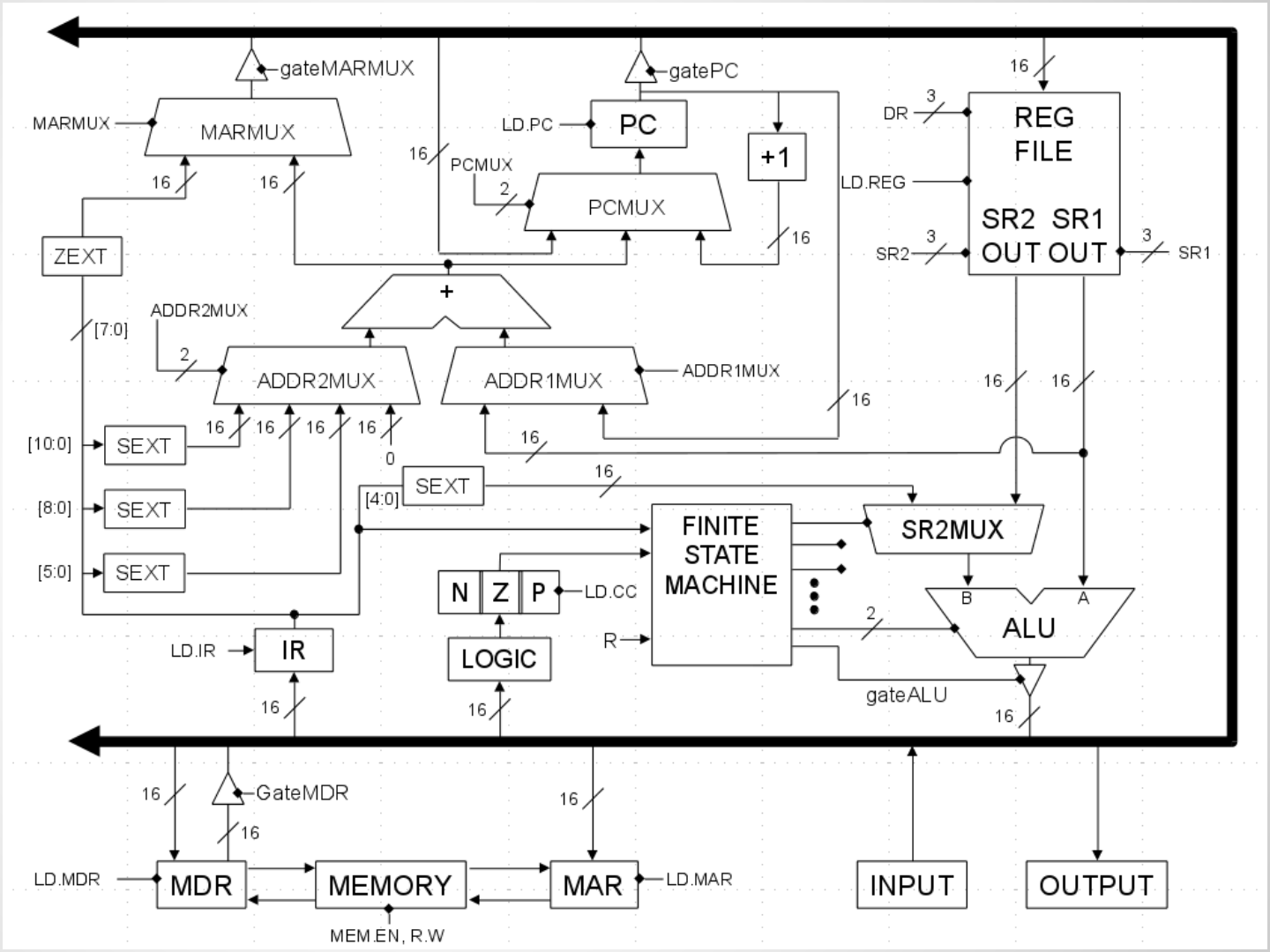
BR (not taken)

CC1:
Check NZP
(internally in FSM)

No branch; return to fetch.

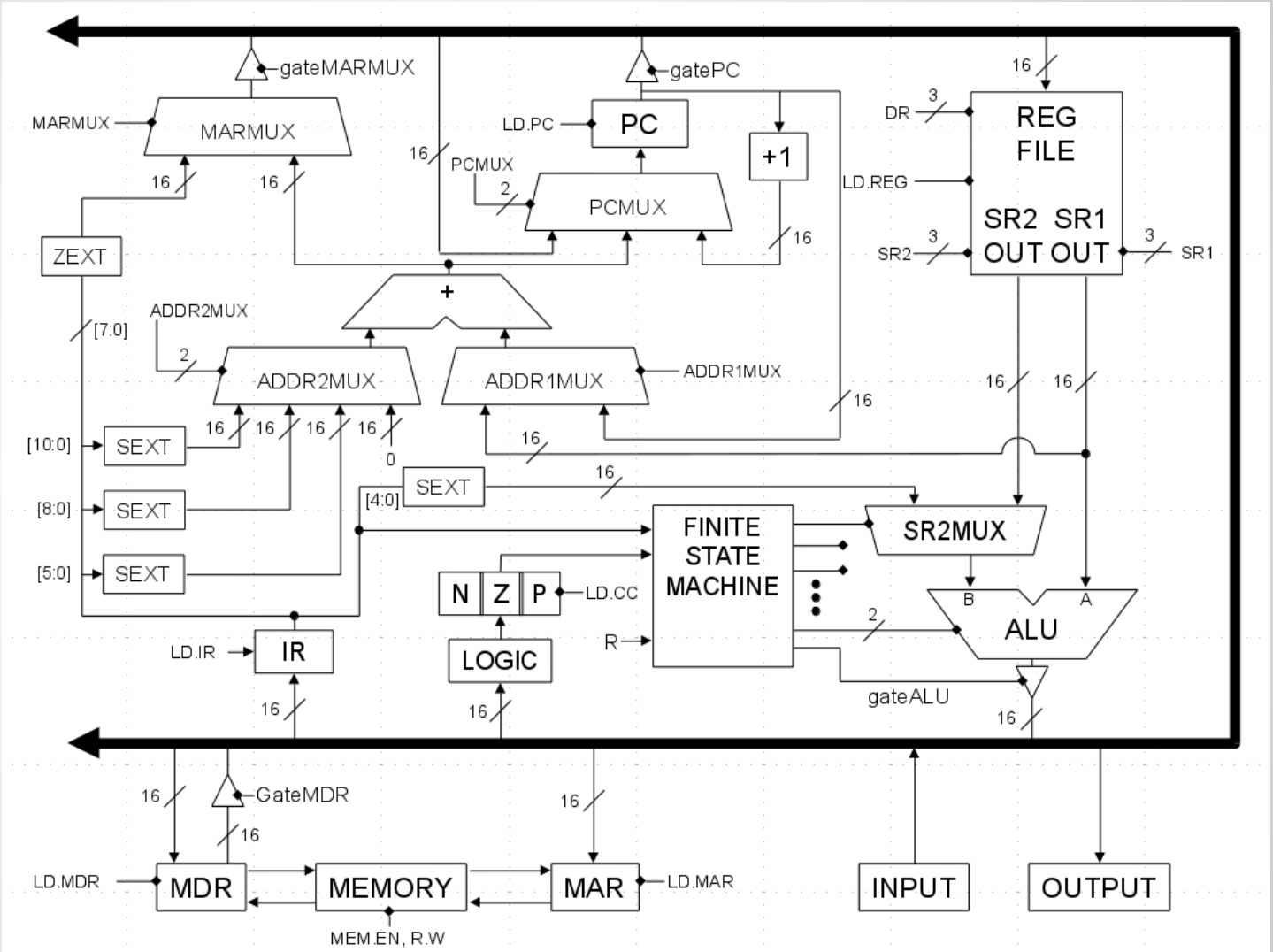


BR (taken)


gateMARMUX
gatePC
MARMUX
MARMUX
LD.PC
PC
+1
PCMUX
PCMUX
REG FILE
DR
LD.REG
SR2 SR1 OUT OUT
SR2
SR1
ZEXT
+
ADDR2MUX
ADDR2MUX
ADDR1MUX
ADDR1MUX
[7:0]
[10:0]
[8:0]
[5:0]
[4:0]
SEXT
SEXT
SEXT
SEXT
FINITE STATE MACHINE
SR2MUX
N
Z
P
LD.CC
LOGIC
R
ALU
B
A
LD.IR
IR
gateALU
GateMDR
LD.MDR
MDR
MEMORY
MAR
LD.MAR
MEM.EN, R.W
INPUT
OUTPUT

BR (taken)

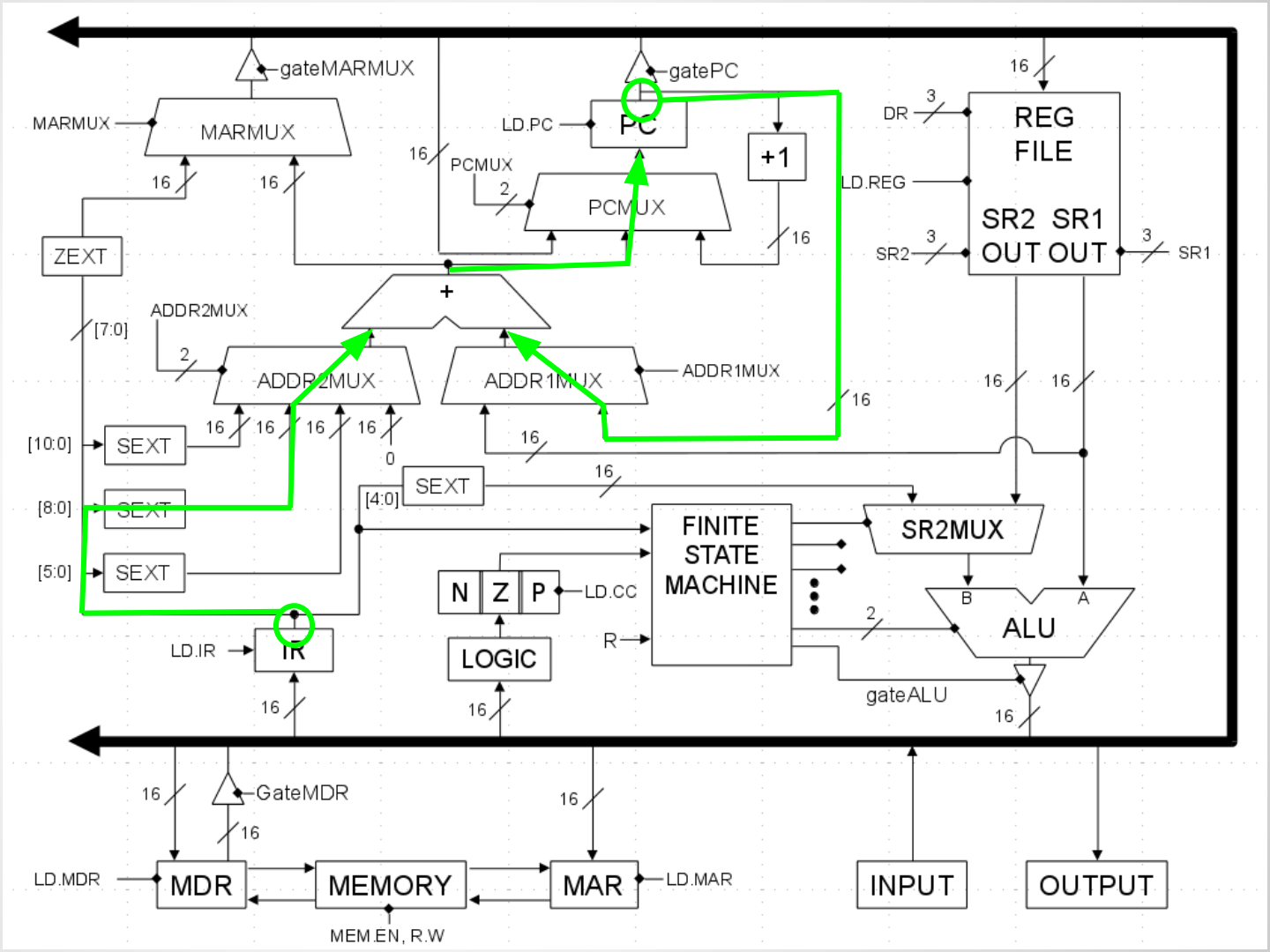
CC1:
Check NZP
(internally in FSM)



BR (taken)

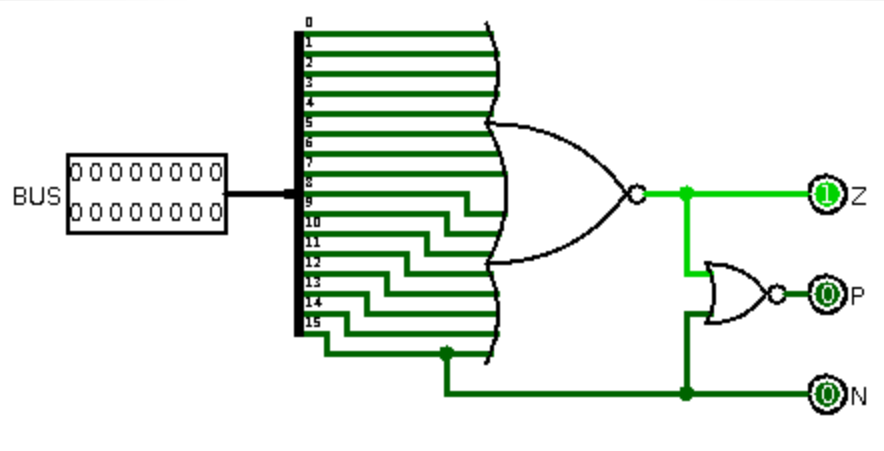
CC1:
Check NZP
(internally in FSM)

CC2:
ADDR1MUX = PC
ADDR2MUX =
SEXT[8:0]
PCMUX = ADDER
LD.PC



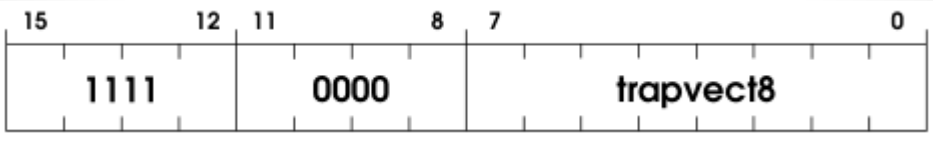
BR

“Logic”:



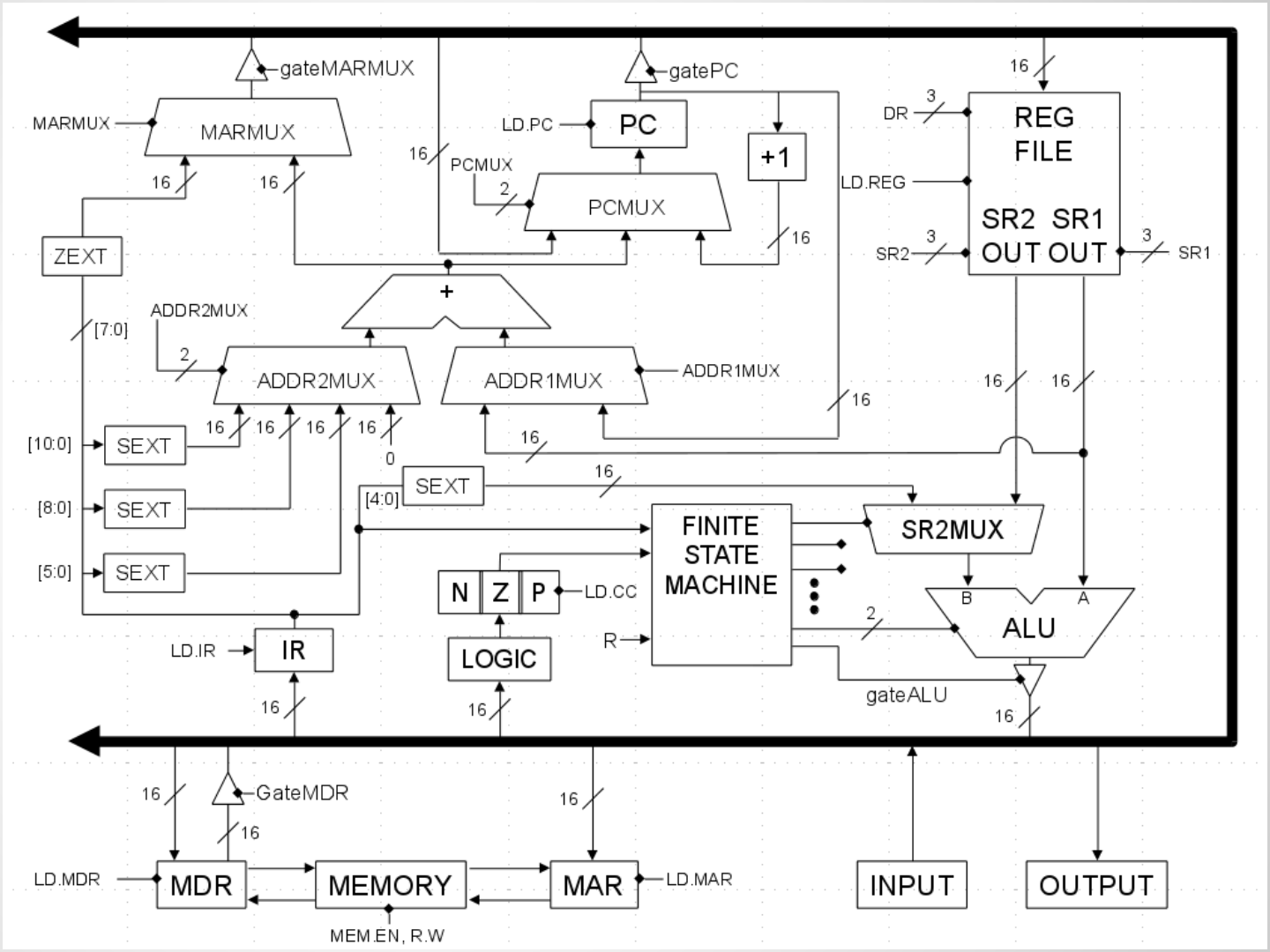
TRAP

The TRAP instruction stores the current PC in R7, locates an 8-bit TRAP vector at the beginning of memory, and sets the PC to that vector's value.



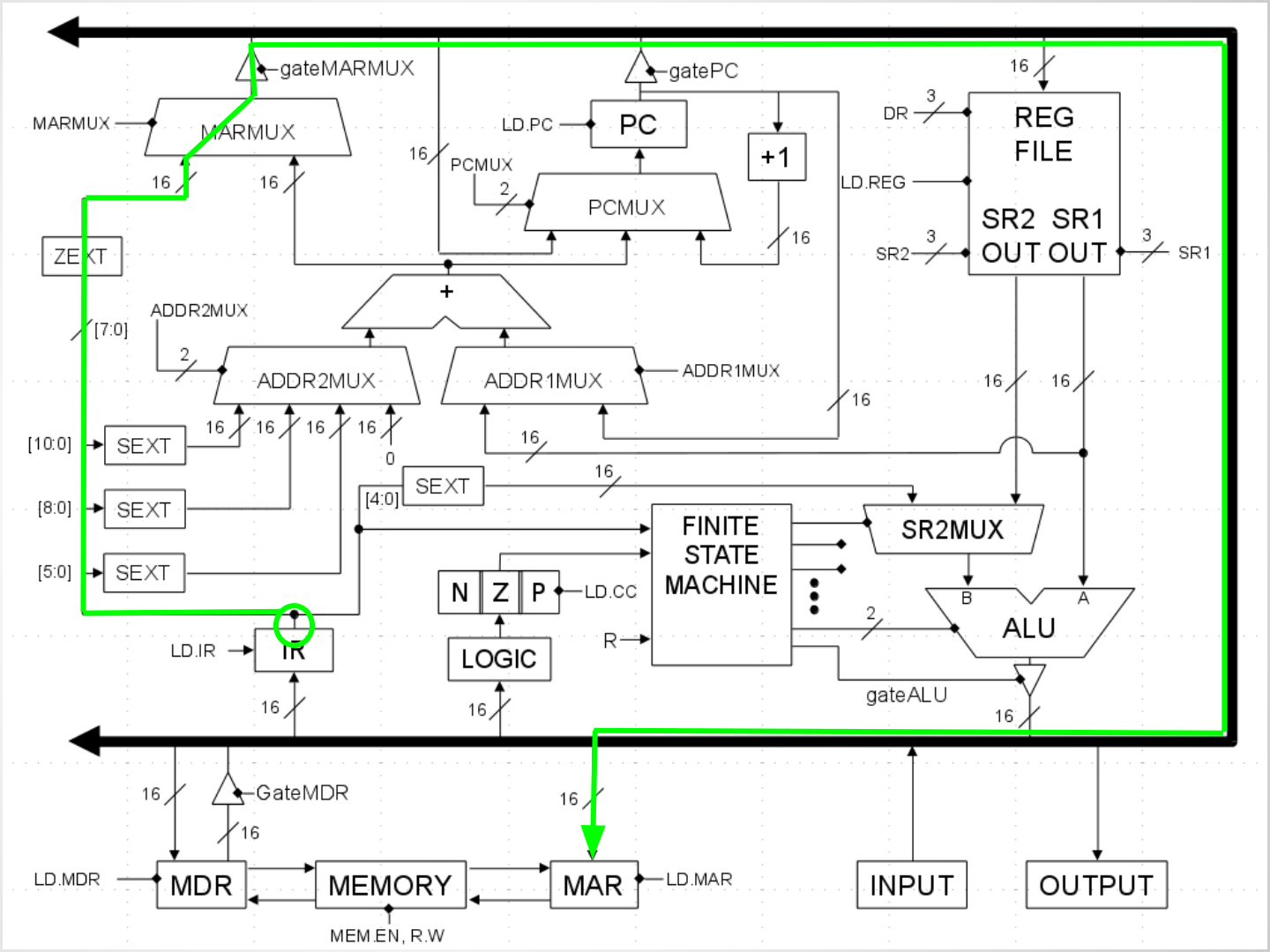
R7 = PC; PC ← mem[trapvect8]

TRAP


gateMARMUX
gatePC
MARMUX
MARMUX
LD.PC
PC
+1
PCMUX
PCMUX
REG FILE
SR2 SR1
OUT OUT
DR
LD.REG
SR2
SR1
ZEXT
[7:0]
ADDR2MUX
ADDR2MUX
ADDR1MUX
ADDR1MUX
+
[10:0]
SEXT
[8:0]
SEXT
[5:0]
SEXT
[4:0]
SEXT
FINITE STATE MACHINE
SR2MUX
N
Z
P
LD.CC
LOGIC
R
ALU
B
A
LD.IR
IR
gateALU
GateMDR
LD.MDR
MDR
MEMORY
MAR
LD.MAR
MEM.EN, R.W
INPUT
OUTPUT

TRAP

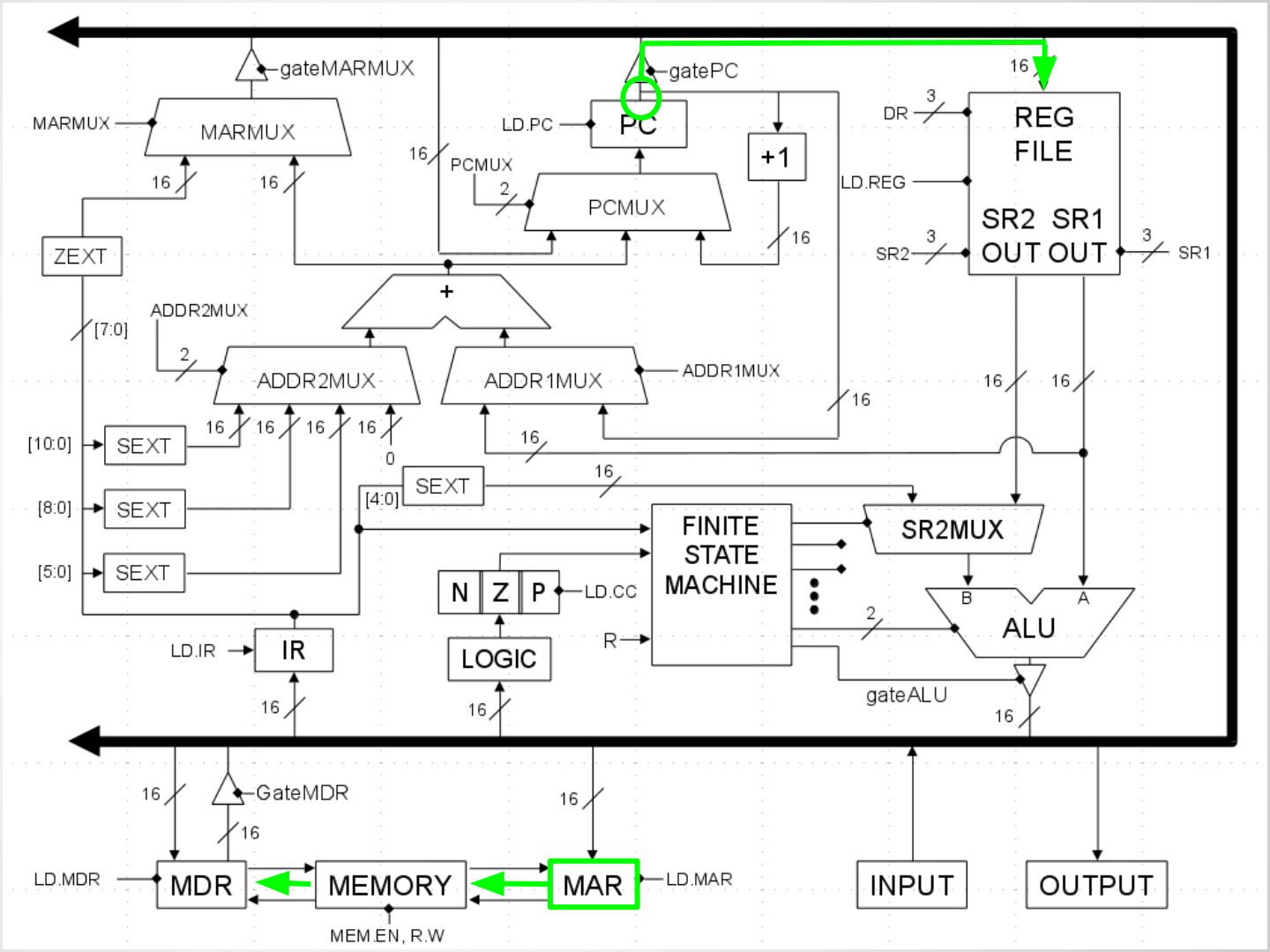
CC1:
MARMUX = ZEXT
gateMARMUX
LD.MAR



TRAP

CC1:
MARMUX = ZEXT
gateMARMUX
LD.MAR

CC2:
MEM.EN
LD.MDR
gatePC
DR = 7
LD.REG

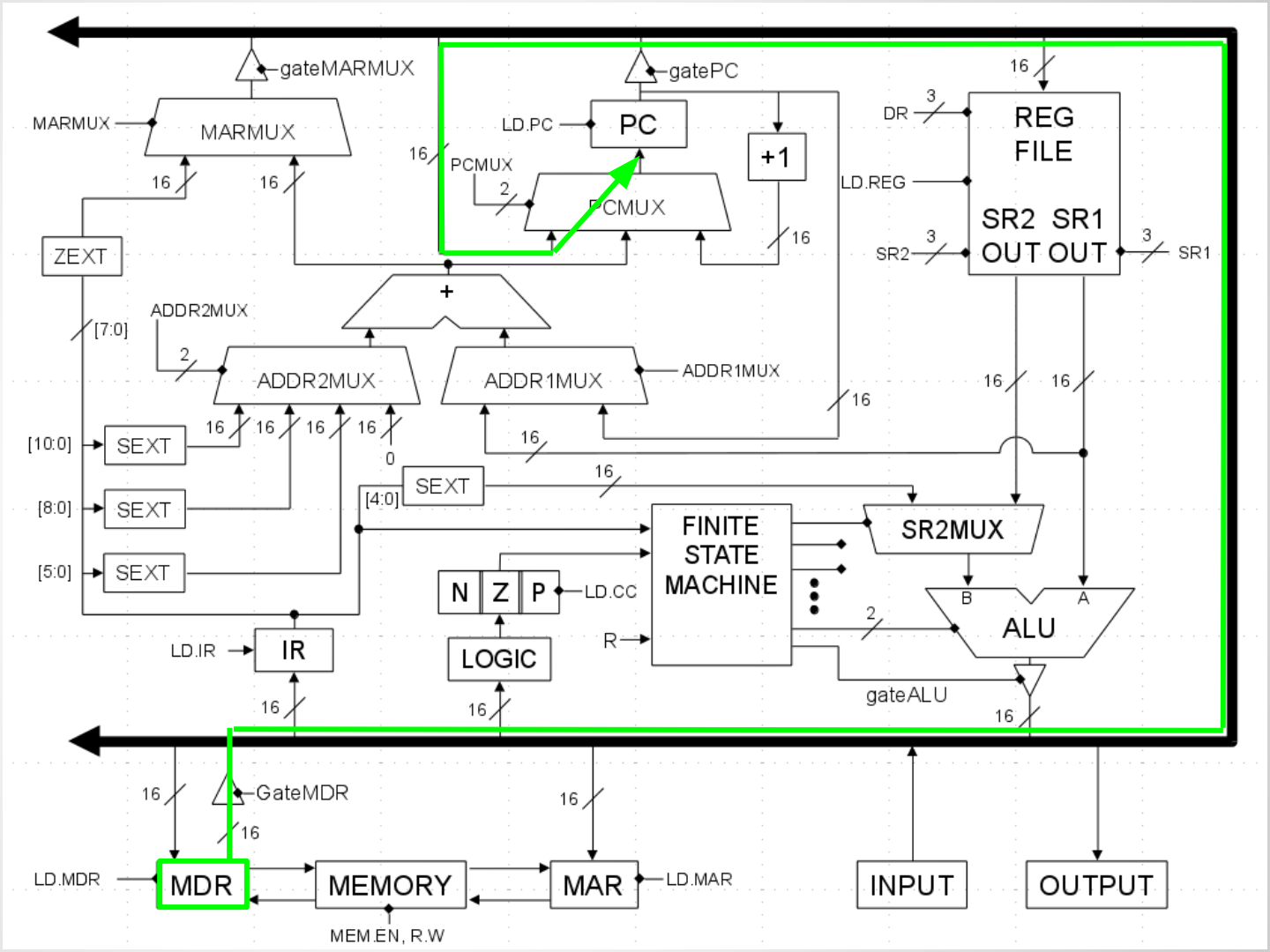


TRAP

CC1:
MARMUX = ZEXT
gateMARMUX
LD.MAR

CC2:
MEM.EN
LD.MDR
gatePC
DR = 7
LD.REG

CC3:
gateMDR
PCMUX = BUS
LD.PC



TRAP

The LC3 comes with several built-in TRAPs that perform practical functions:

- x20 (GETC): Read one ASCII character from the input buffer, and store it in R0
- x21 (OUT): Print a character from R0 to the console
- x22 (PUTS): Print a null-terminated string starting at the address pointed to by R0
- x25 (HALT): Halt the processor
- x23, x24 (IN, PUTSP) See the ISA

TRAP

You can write your own traps! Follow these simple steps:

- Write your trap code somewhere in memory
- Terminate the trap code with RET
- Write the address of the code to a free entry in the trap vector table
- Be careful not to clobber the contents of R7 during the trap code; using JSR/JSRR or TRAP within your trap code will clobber R7! You should save the contents of R7 somewhere and restore them before RET, if it must be modified in your code.
- Call your trap with TRAP x90, for example, if you wrote the address of the trap code to trap vector x90.

Pipelining

LC3 instructions with in the canonical datapath require from 5 to 9 clock cycles to complete. Could we do anything to speed up the datapath?

Pipelining

After looking at the LC3 instructions, you may start to notice a pattern in execution:

ADD:

- Get instruction
- Get operands
- Add them
- -
- Write to register

LDR:

- Get instruction
- Get SR1 and offset
- Add them
- Retrieve memory
- Write to register

ST:

- Get instruction
- Get SR and offset
- ADD PC and offset
- Write memory
- -

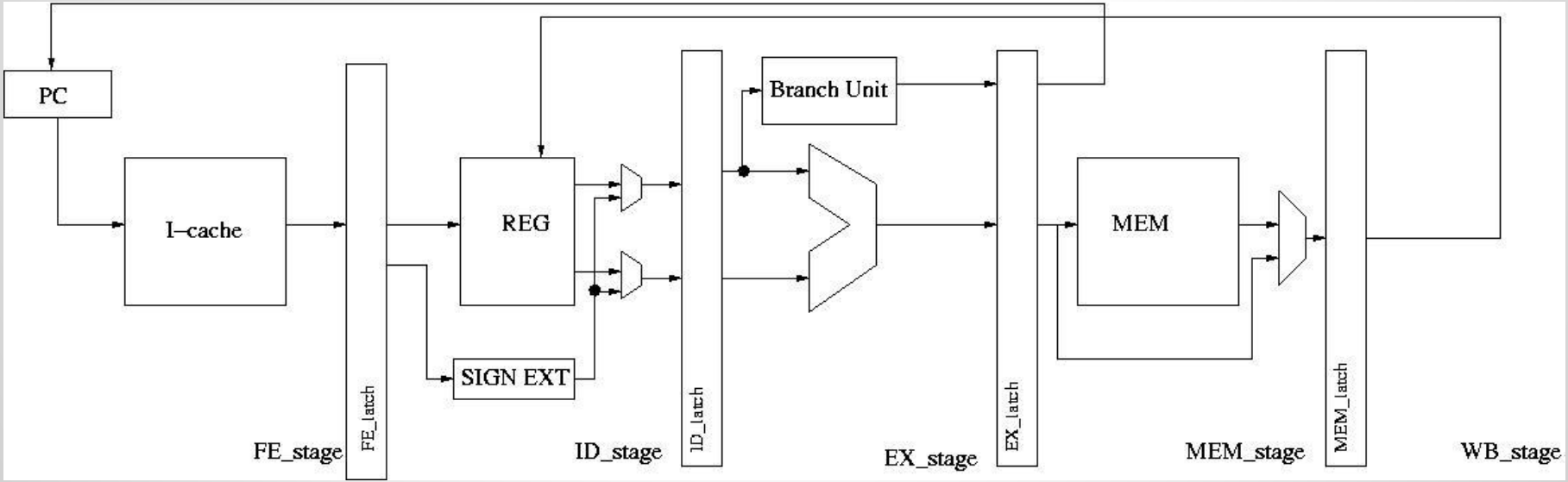
Pipelining

All the instructions follow a general pattern of fetch, decode, execute, memory access, register writeback; why not execute each in its own clock cycle, and start the next instruction while the current is executing?

	Fetch	Decode	Execute	Memory	Writeback
CC1:	inst 1				
CC2:	inst 2	inst 1			
CC3:	inst 3	inst 2	inst 1		
CC4:	inst 4	inst 3	inst 2	inst 1	
CC5:	inst 5	inst 4	inst 3	inst 2	inst 1

Pipelining

Place a register buffer between each stage; the instructions will ripple across the datapath, overall ~1 cycle per instr.



Pipelining

Problems to deal with in pipelining:

- Structural hazards
  - Component needed by multiple pipeline stages
- Data hazards
  - Data may be read before it’s written back
- Control hazards
  - Residual data in pipeline stages after branches

Take CS 2200!