

# CS2200

## Systems and Networks

### Spring 2024

# Lecture 6: Control path

Alexandros (Alex) Daglis  
School of Computer Science  
Georgia Institute of Technology  
[adaglis@gatech.edu](mailto:adaglis@gatech.edu)

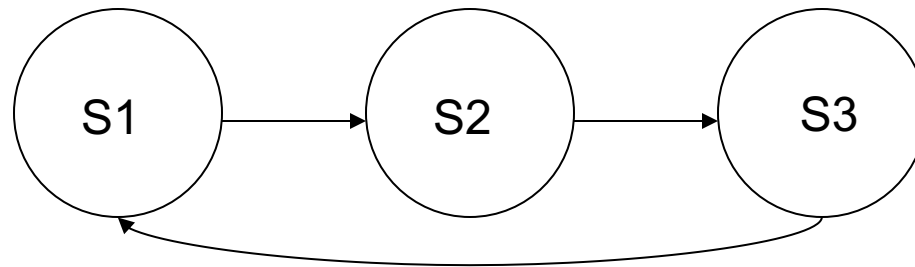
# Topics

---

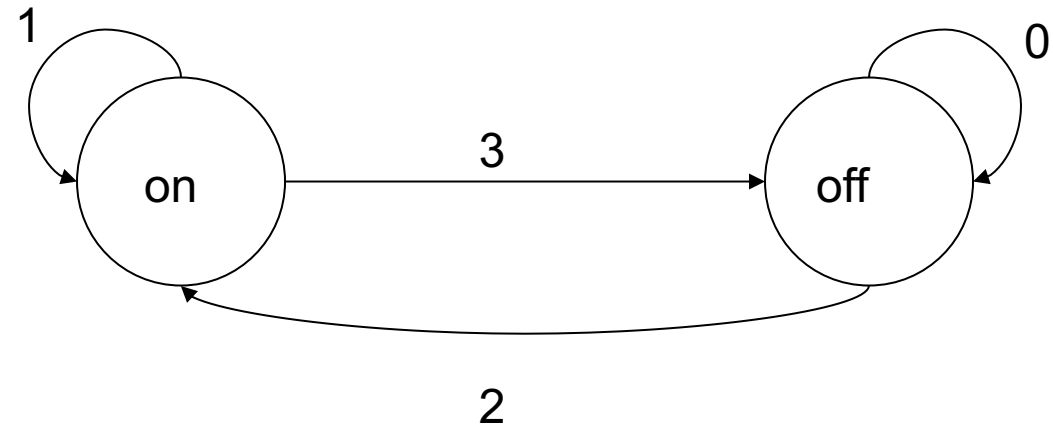
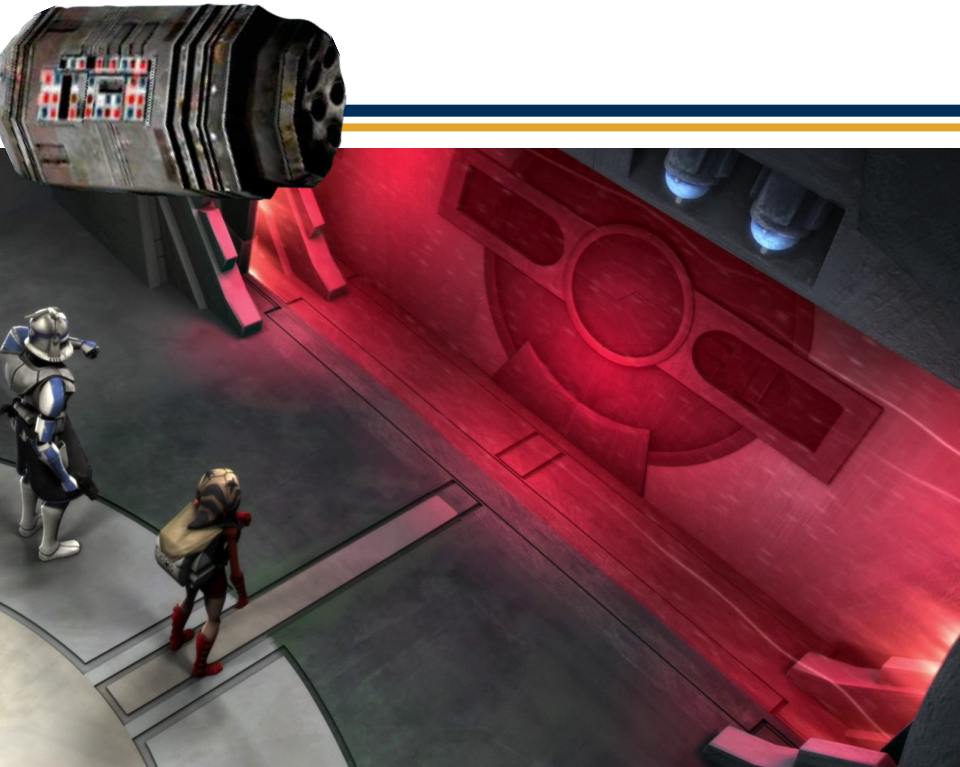
- ~~Logic design review~~
- ~~Data paths~~
- Finite State Machines

# Simple FSM Example

---

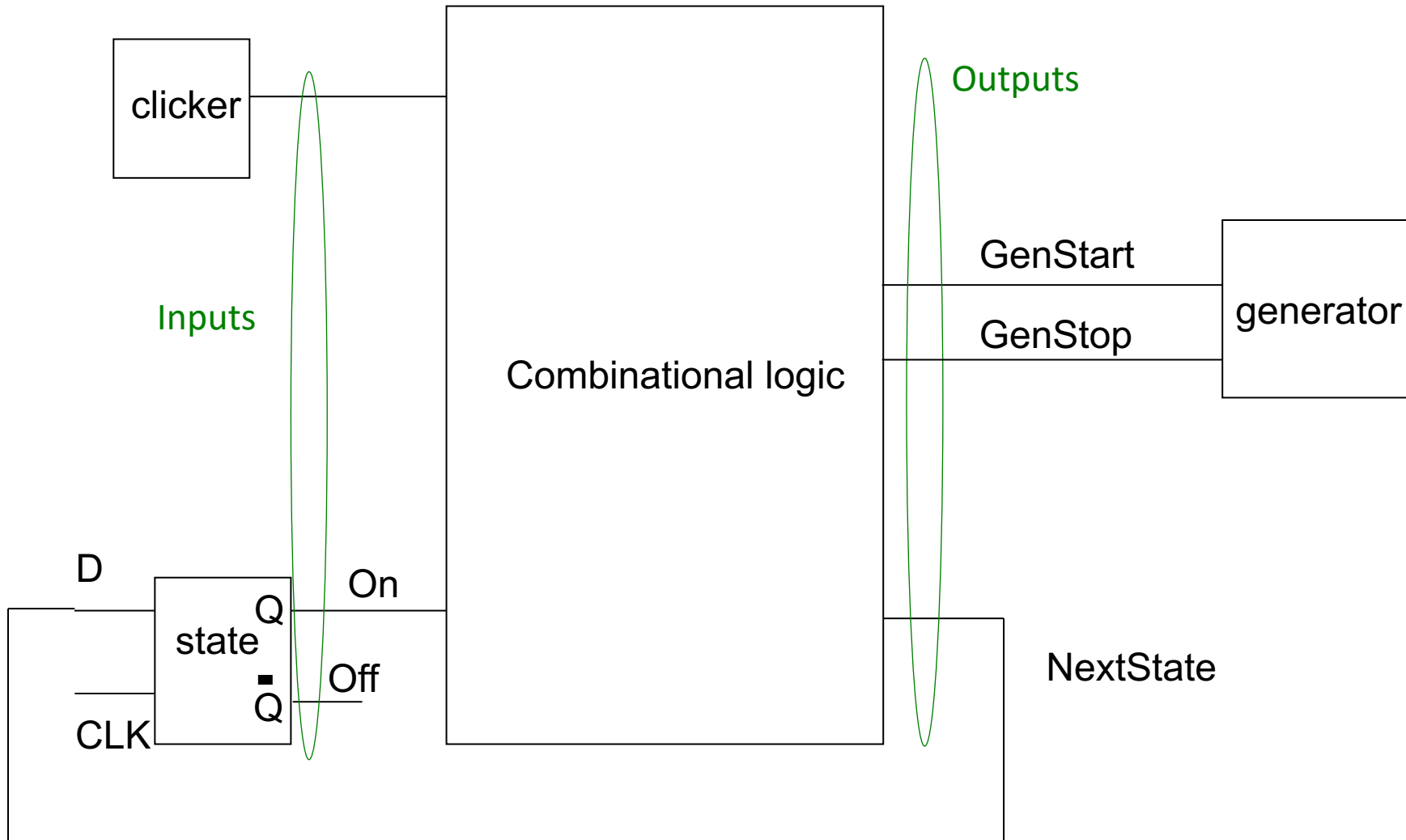


# Ray Shield



Transition No	Current State	Clicker	GenStart	GenStop	Next State
0	0	0	0	0	0
1	1	0	0	0	1
2	0	1	1	0	1
3	1	1	0	1	0

# Ray Shield Controller



# Combinational Logic

- $\text{GenStart} = \text{CurrentState}' \& \text{Clicker}$
- $\text{GenStop} = \text{CurrentState} \& \text{Clicker}$
- $\text{NextState} = (\text{CurrentState} \& \text{Clicker}') \mid (\text{CurrentState}' \& \text{Clicker})$

Transition No	Current State	Clicker	GenStart	GenStop	Next State
0	0	0	0	0	0
1	1	0	0	0	1
2	0	1	1	0	1
3	1	1	0	1	0

# Can We Replace Combinational Logic with a ROM?

- Since we can describe combinational logic as boolean expressions, what if we could replace a **combinational circuit** with a **hardware truth table**?
- Think of a properly programmed ROM as a literal truth table describing an FSM
  - The **address** represents the input bits (including current state)
  - The **contents** of the ROM produce the output bits (including the next state)
- Have you seen this somewhere before?

x	1	2	3	4	5	6	7	8	9	10
1	1	2	3	4	5	6	7	8	9	10
2	2	4	6	8	10	12	14	16	18	20
3	3	6	9	12	15	18	21	24	27	30
4	4	8	12	16	20	24	28	32	36	40
5	5	10	15	20	25	30	35	40	45	50
6	6	12	18	24	30	36	42	48	54	60
7	7	14	21	28	35	42	49	56	63	70
8	8	16	24	32	40	48	56	64	72	80
9	9	18	27	36	45	54	63	72	81	90
10	10	20	30	40	50	60	70	80	90	100

# From FSM to ROM

Transition No	Current State	Clicker	GenStart	GenStop	Next State
0	0	0	0	0	0
1	1	0	0	0	1
2	0	1	1	0	1
3	1	1	0	1	0

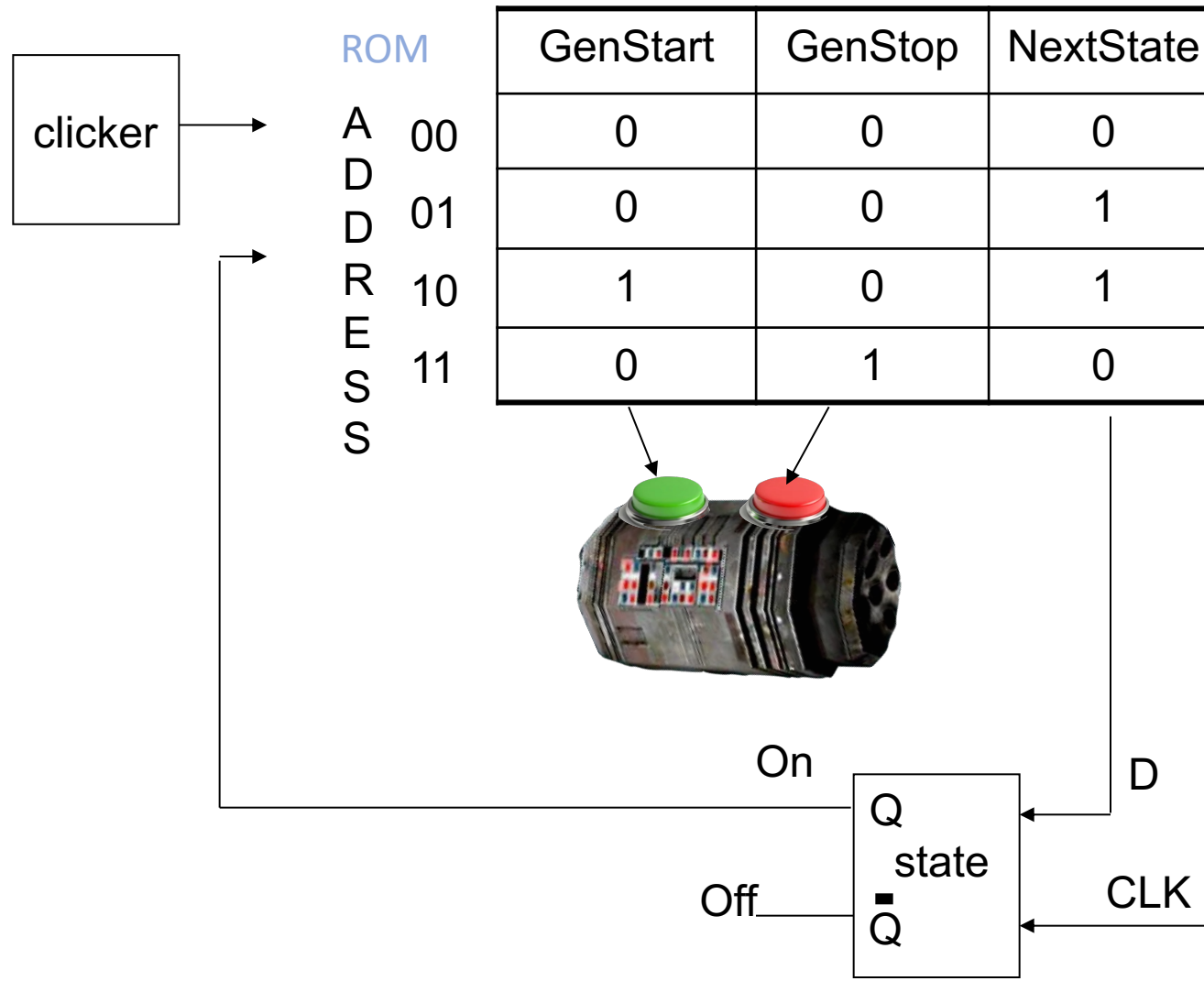
ROM: 3-bit word x 4 words  
→ 3 data bits, 2 address bits

A  
D  
D  
R  
E  
S  
S

	GenStart	GenStop	NextState
00	0	0	0
01	0	0	1
10	1	0	1
11	0	1	0



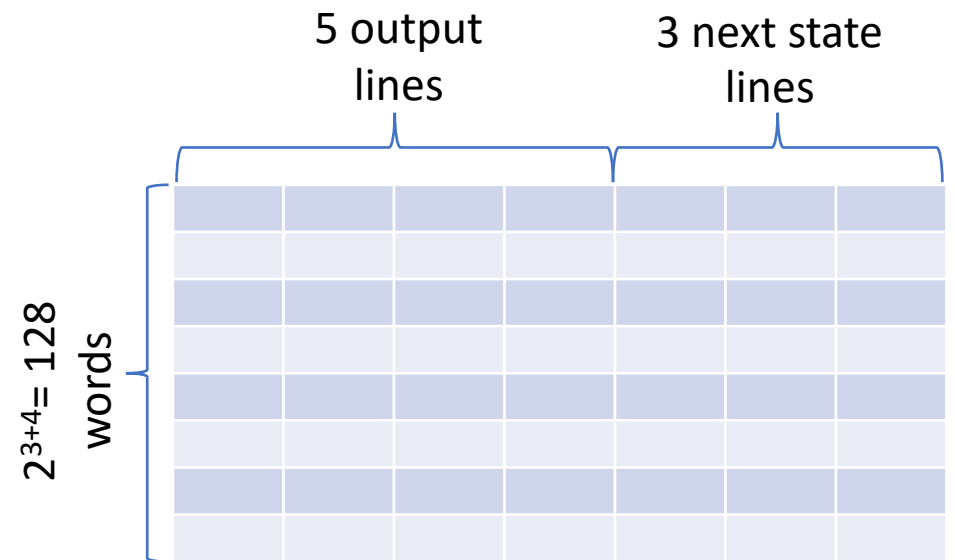
# Replacing Discrete Logic with a ROM



# How large a ROM?

If you have a truth table with a 4-bit input, 8 states (i.e., 3 state bits), and 5 outputs, what size ROM should you use to encode it?

- A.  $2^7$  words of 8 bits
- B.  $2^4$  words of 5 bits
- C.  $2^8$  words of 7 bits
- D.  $2^4$  words of 8 bits

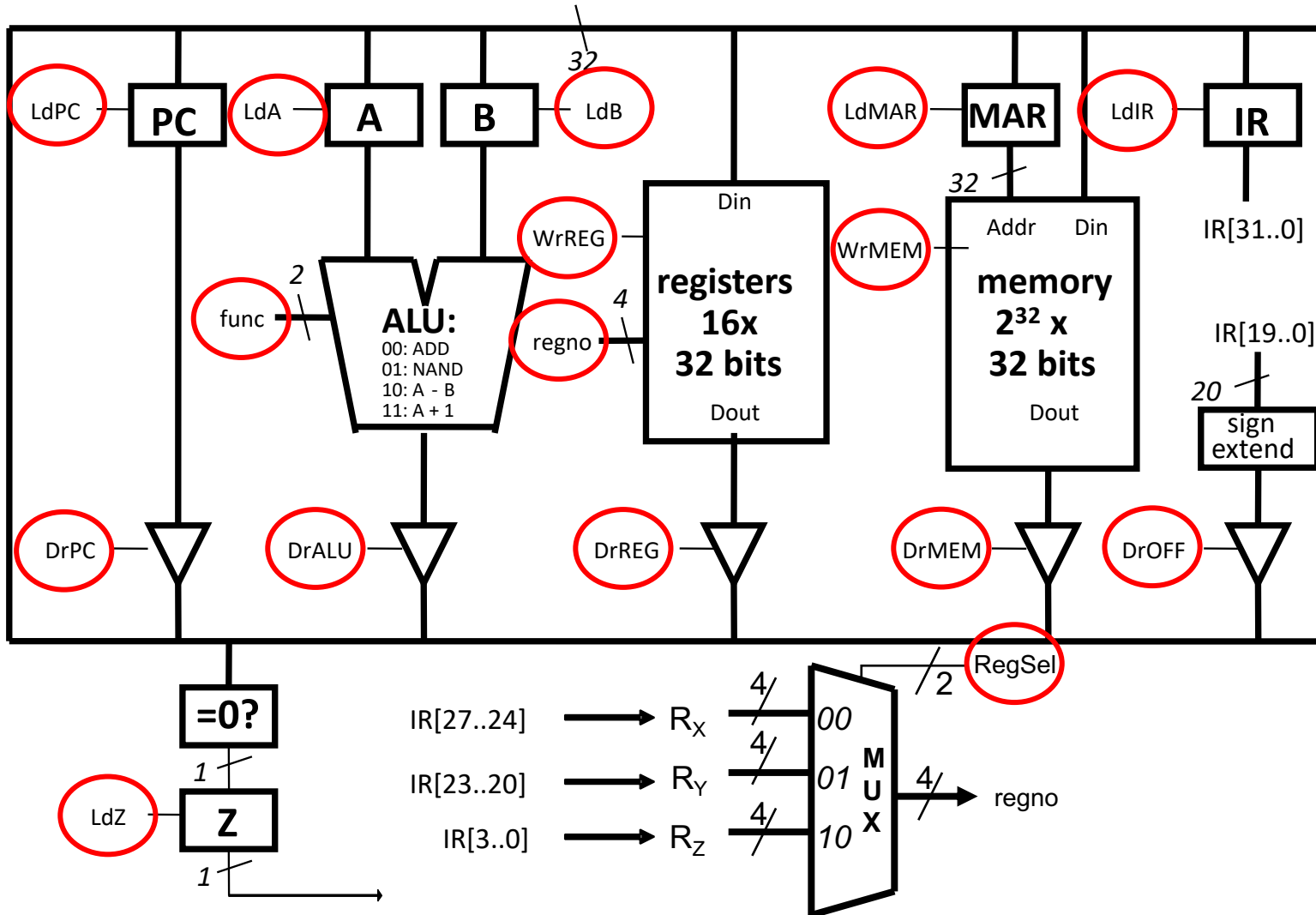


# Checkpoint

---

- Basics of logic design
  - Combinational
  - Sequential
- Elements of the datapath
  - Registers & register file
  - ALU
  - Mux
  - Decoders
  - Clock & clock width
  - Finite state machine (combinational and truth table)

# We've Got a Datapath for LC-2200!

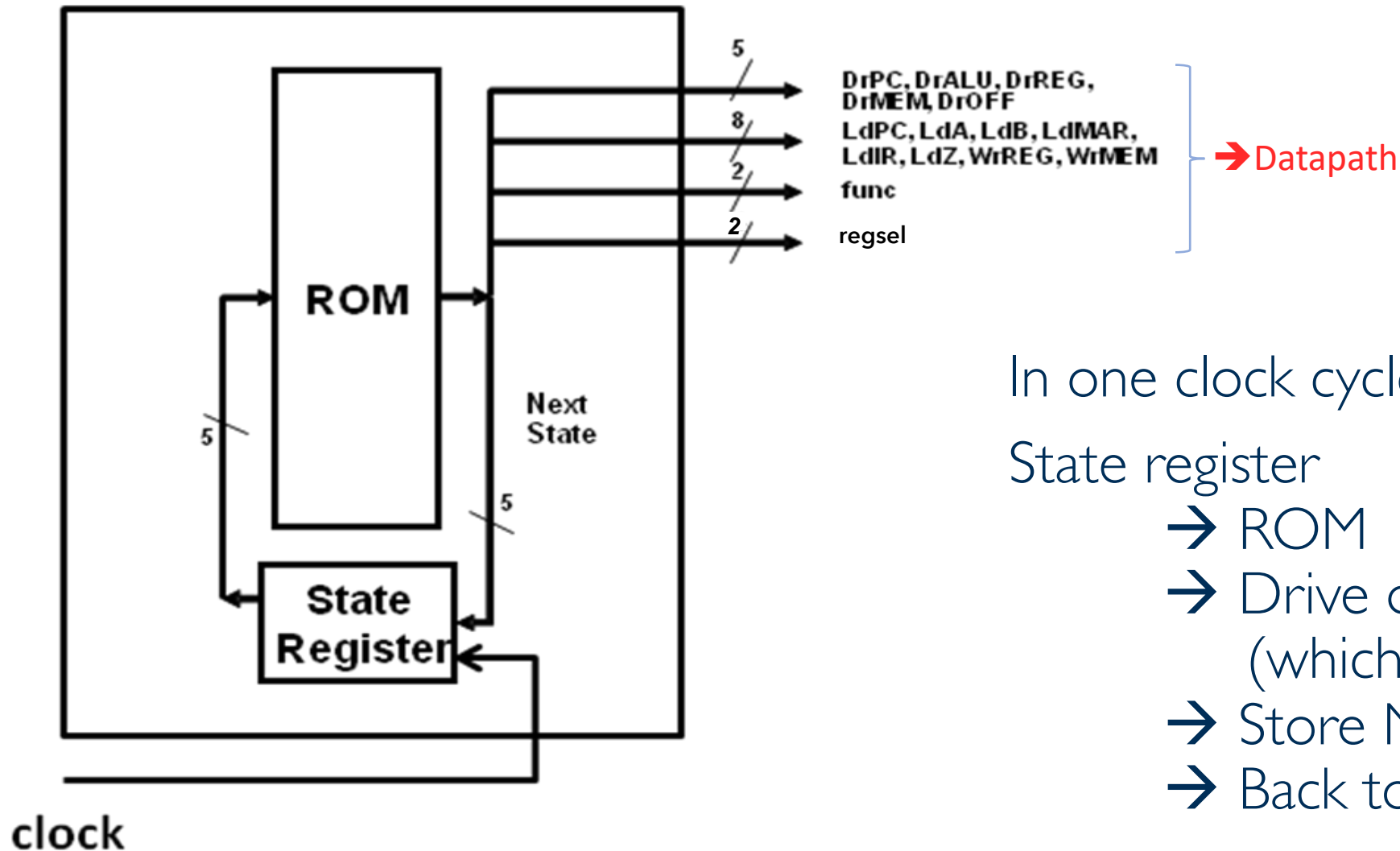


What else do we need??

Who sets all these signals?



# A Control Unit!



In one clock cycle:

State register

→ ROM

→ Drive datapath  
(which does the work)

→ Store Next State

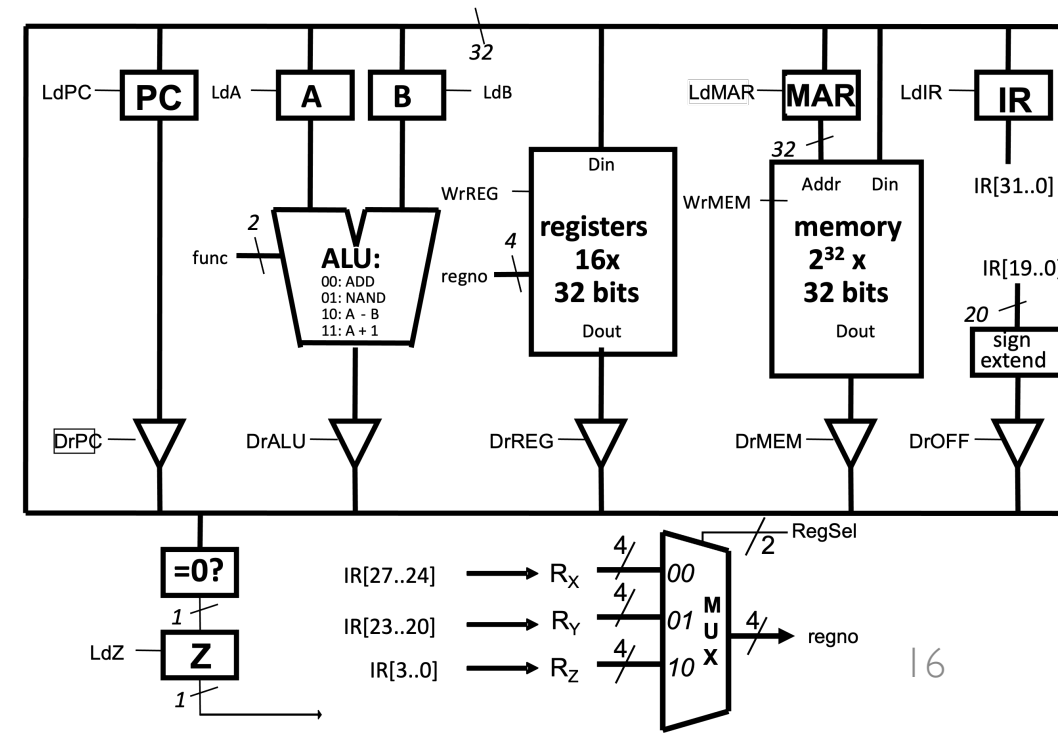
→ Back to the start

# What's in the ROM?

	Drive Signals					Load Signals						Write Signals			
Current State	PC	ALU	R e g	MEM	O F F	P C	A	B	M A R	I R	Z	M E M	REG	func	regSel

Recognize all these as the control signals to drive the datapath

You will find each one on the datapath diagram!





# The Next State is Stored in the ROM, Too!

	Drive Signals					Load Signals						Write Signals				
Current State	P C	ALU	Reg	MEM	OFF	PC	A	B	MAR	IR	Z	MEM	REG	func	regSel	Next State
...																

In addition to being the Current State, this is also necessarily the address of the word in the ROM



# This Means the ROM Contents Are Our Microprogram!

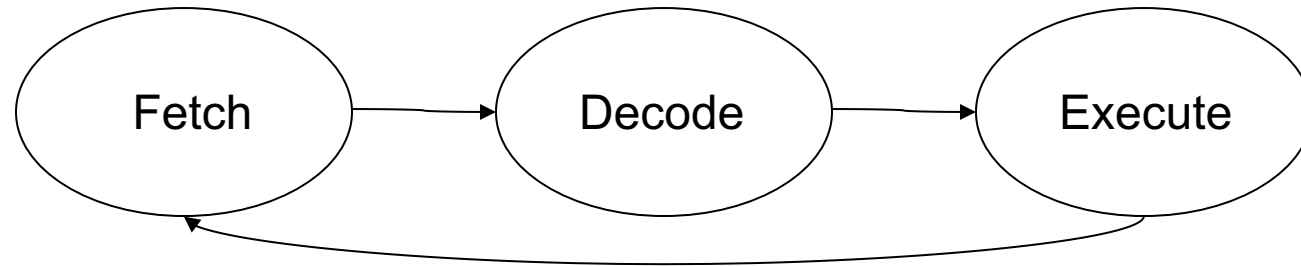
	Drive Signals					Load Signals						Write Signals				
Current State	P C	ALU	Reg	MEM	OFF	PC	A	B	MAR	IR	Z	MEM	REG	func	regSel	Next State
00000	1						1		1							00001
...																

For short, we might write this microinstruction as  
00000: DrPC LdA LdMAR next=00001

# A Familiar State Diagram?

---

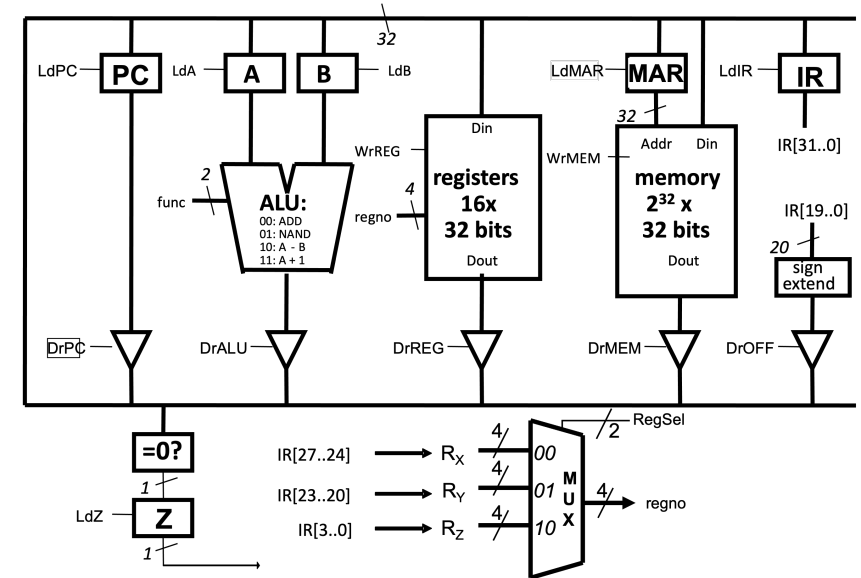
- What state diagram do you think a CPU implementer might be concerned with?



- Is a processor implementation a Finite State Machine?
- What happens in each state?
- What resources are needed to execute each instruction?

# Implementing the LC-2200 ISA

- R-type instructions
- Sequence of machine states are similar
- Only the ALU op changes



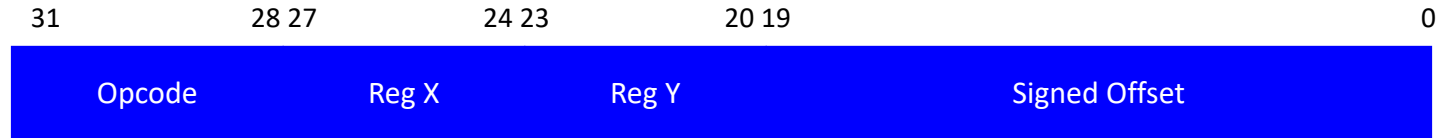
	Drive Signals					Load Signals						Write Signals				
Current State	P C	ALU	Reg	MEM	OF F	PC	A	B	MAR	I R	Z	MEM	REG	func	Reg Sel	Next State
add1			1				1								01	add2
add2			1					1							10	add3
add3		1											1	00	00	ifetch1

# Implementing the LC-2200 ISA

---

- R-type instructions
  - Sequence of machine states are similar
  - Only the ALU op changes
- J-type instructions
  - Straightforward
- I-type instructions (LW, SW, ADDI)
  - Straightforward
- I-type instructions (BEQ)
  - May take some thought...
  - Let's do that first

# So How Do We Handle BEQ?



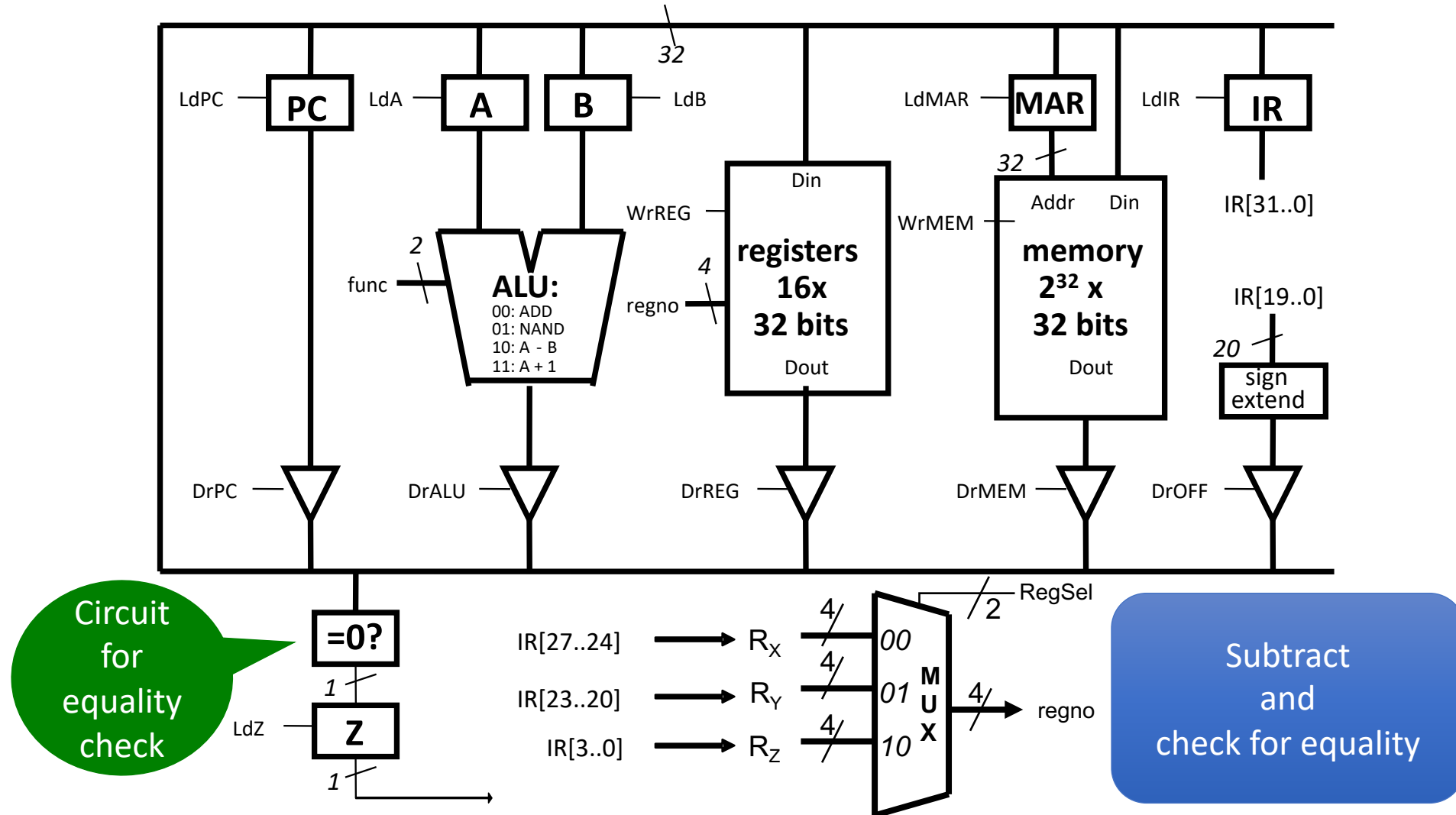
BEQ has the following semantics:

if  $\text{RegX} == \text{RegY}$  then  $\text{PC} \leftarrow \text{PC} + 1 + \text{signed-offset}$   
else nothing

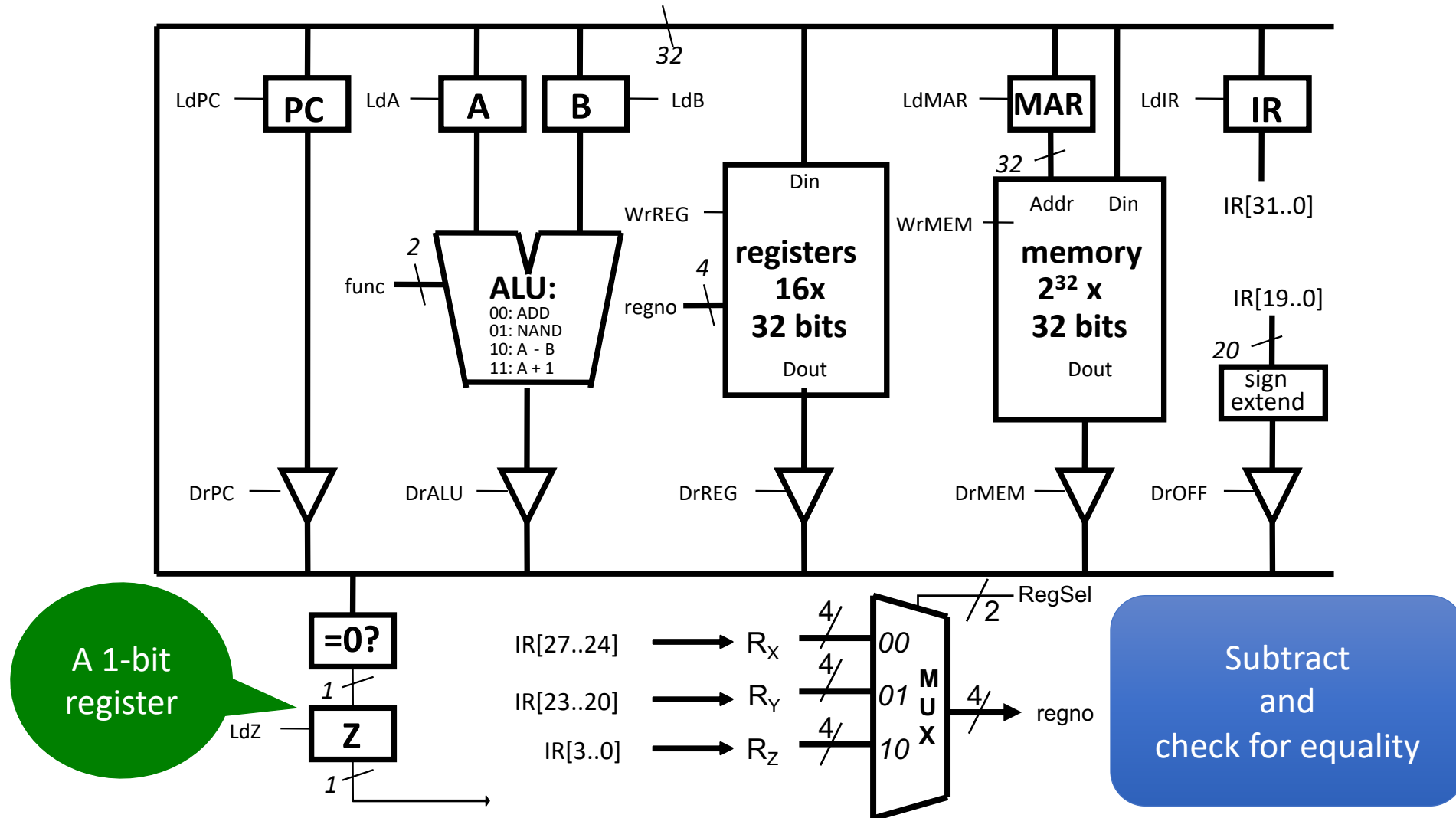
Read values of RegX and RegY & perform comparison

How do we do that?

# Implementing BEQ



# Implementing BEQ

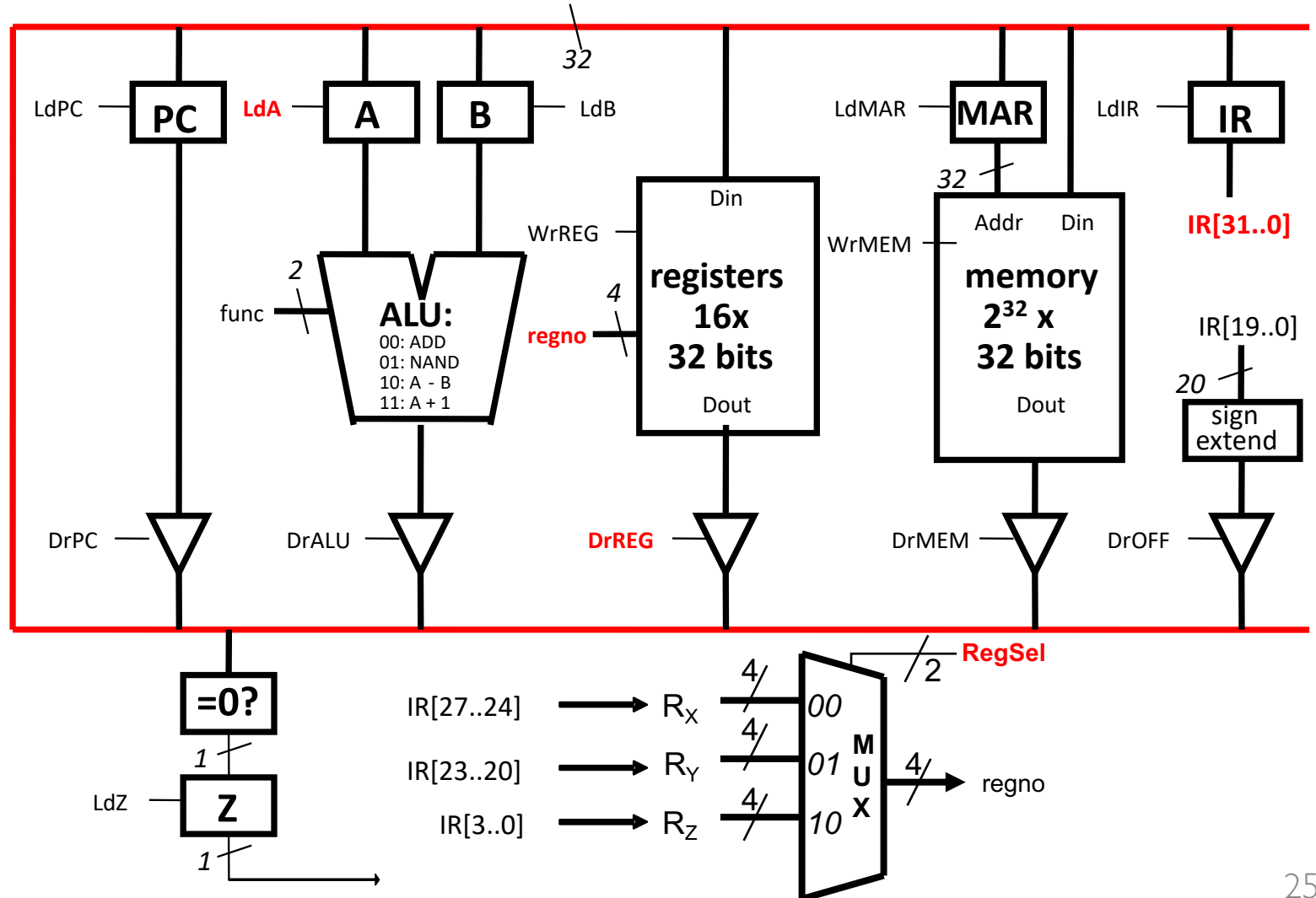


# beq l

$R_x \rightarrow A$

Control signals needed:

- Regsel=00
- DrReg
- LdA



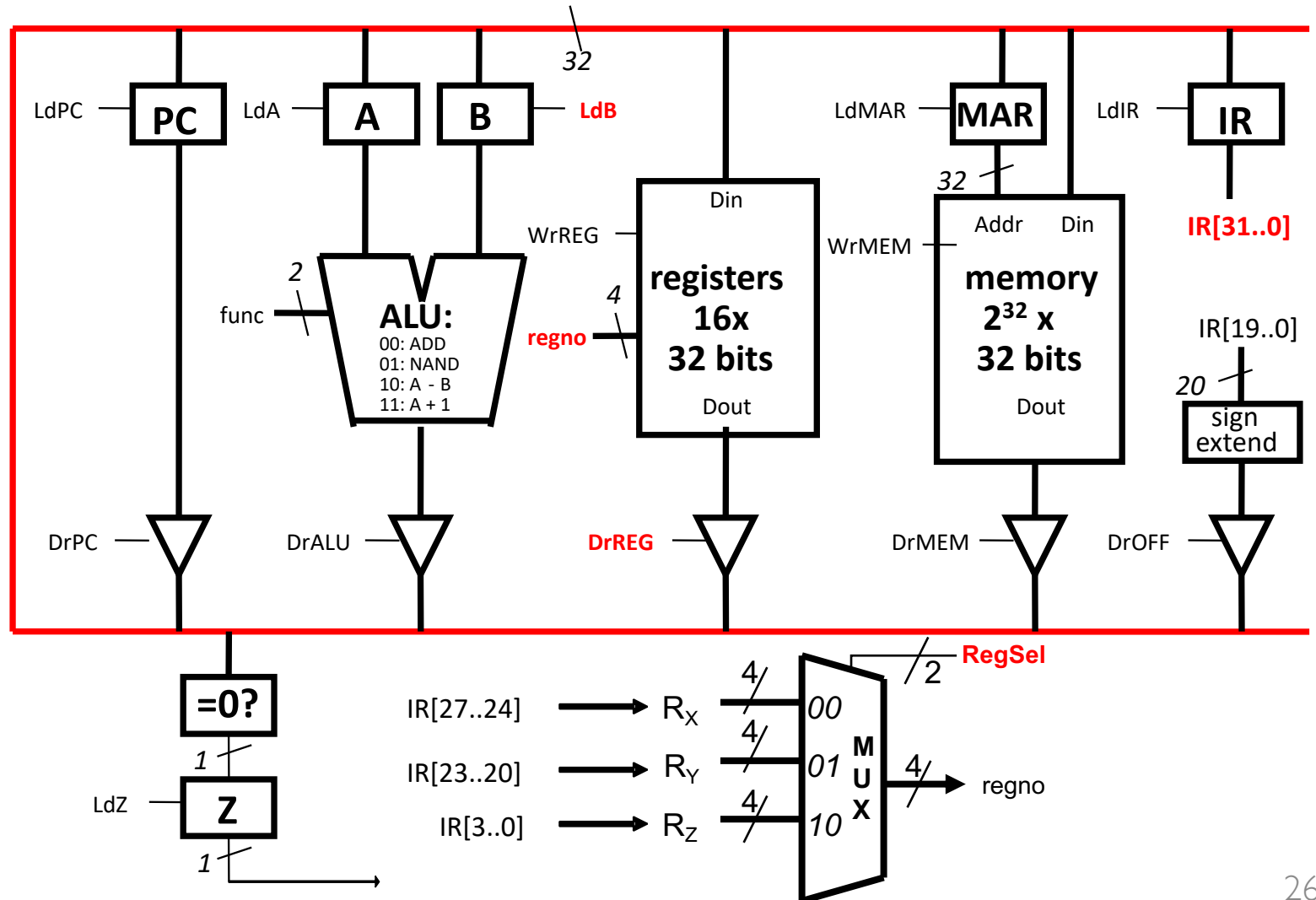


# beq2

$R_y \rightarrow B$

Control signals needed:

- Regsel=01
- DrReg
- LdB



# beq3

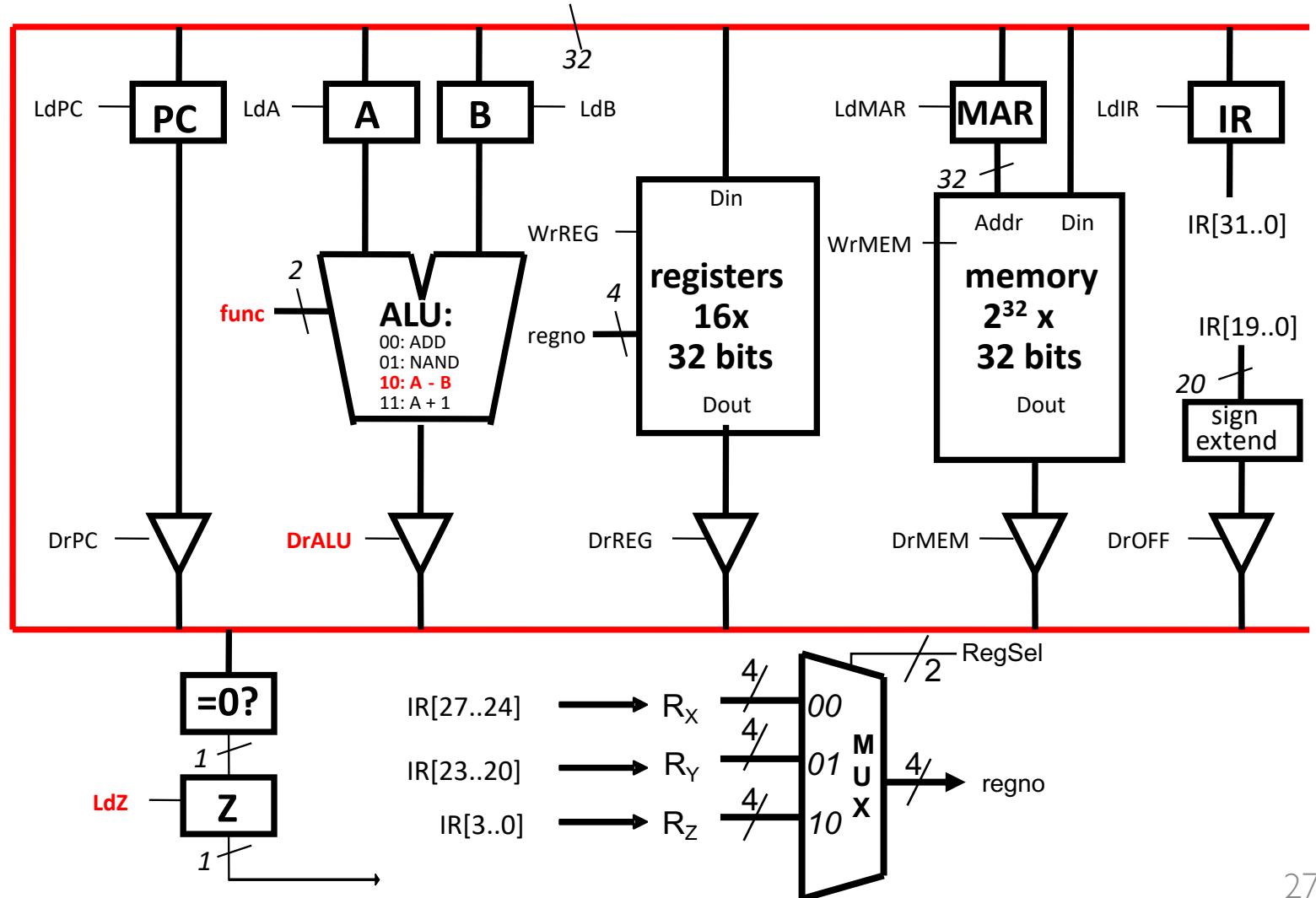
A-B

Load Z register with results of zero detect logic

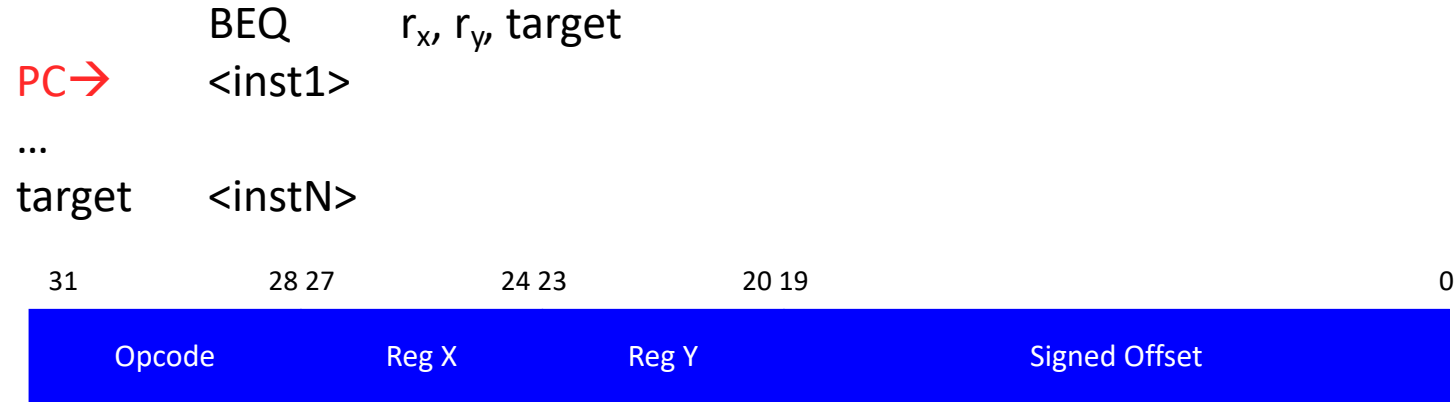
Control signals needed:

- $\text{func} = 10$
- DrALU
- LdZ

At this point,  
what do we  
know?



# Decision Time



BEQ has the following semantics:

if  $\text{RegX} == \text{RegY}$  then  $\text{PC} \leftarrow \text{PC} + 1 + \text{signed offset}$   
else nothing

i.e. go back to fetch  
the next instruction  
(e.g. <inst1>)

if branch is taken, next instruction  
is <instN>

# beq3

A-B

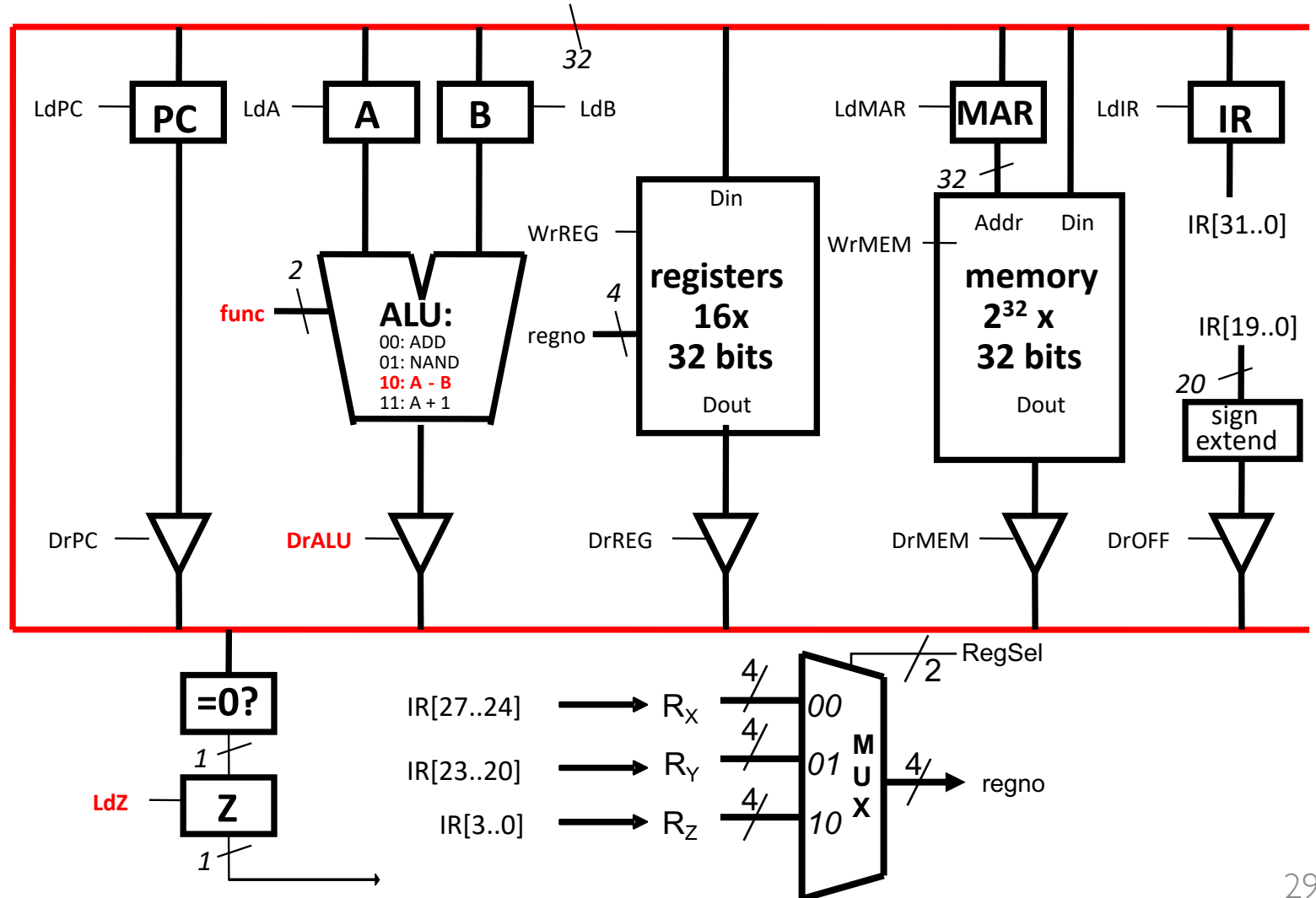
Load Z register with results of zero detect logic

Control signals needed:

- $func = 10$
- DrALU
- LdZ

If Z==1 →  
compute  
target  
address

If Z==0 → go  
to ifetch1



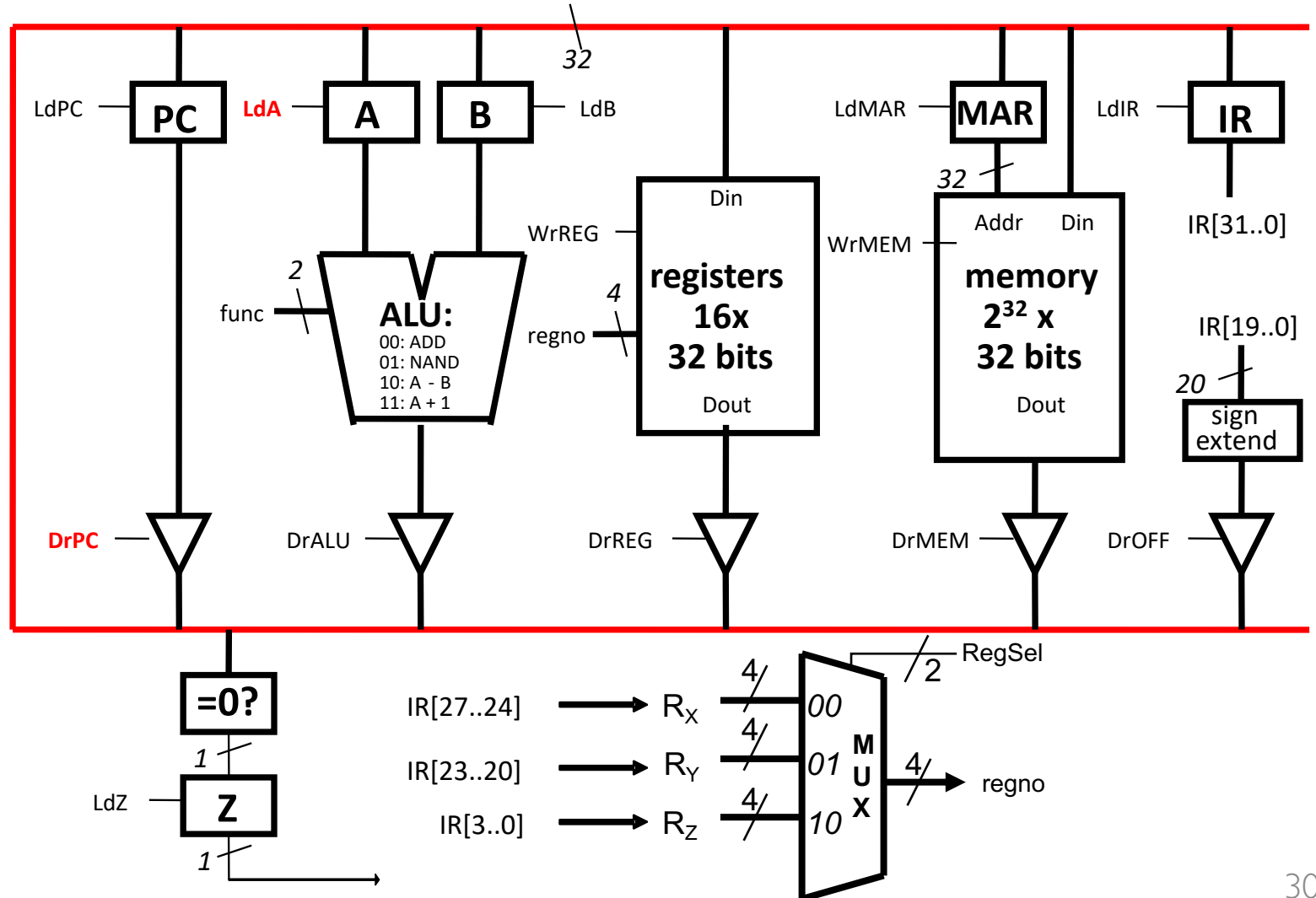
Presume  
we've taken  
the branch

beq4

PC → A

Control signals needed:

- DrPC
- LdA

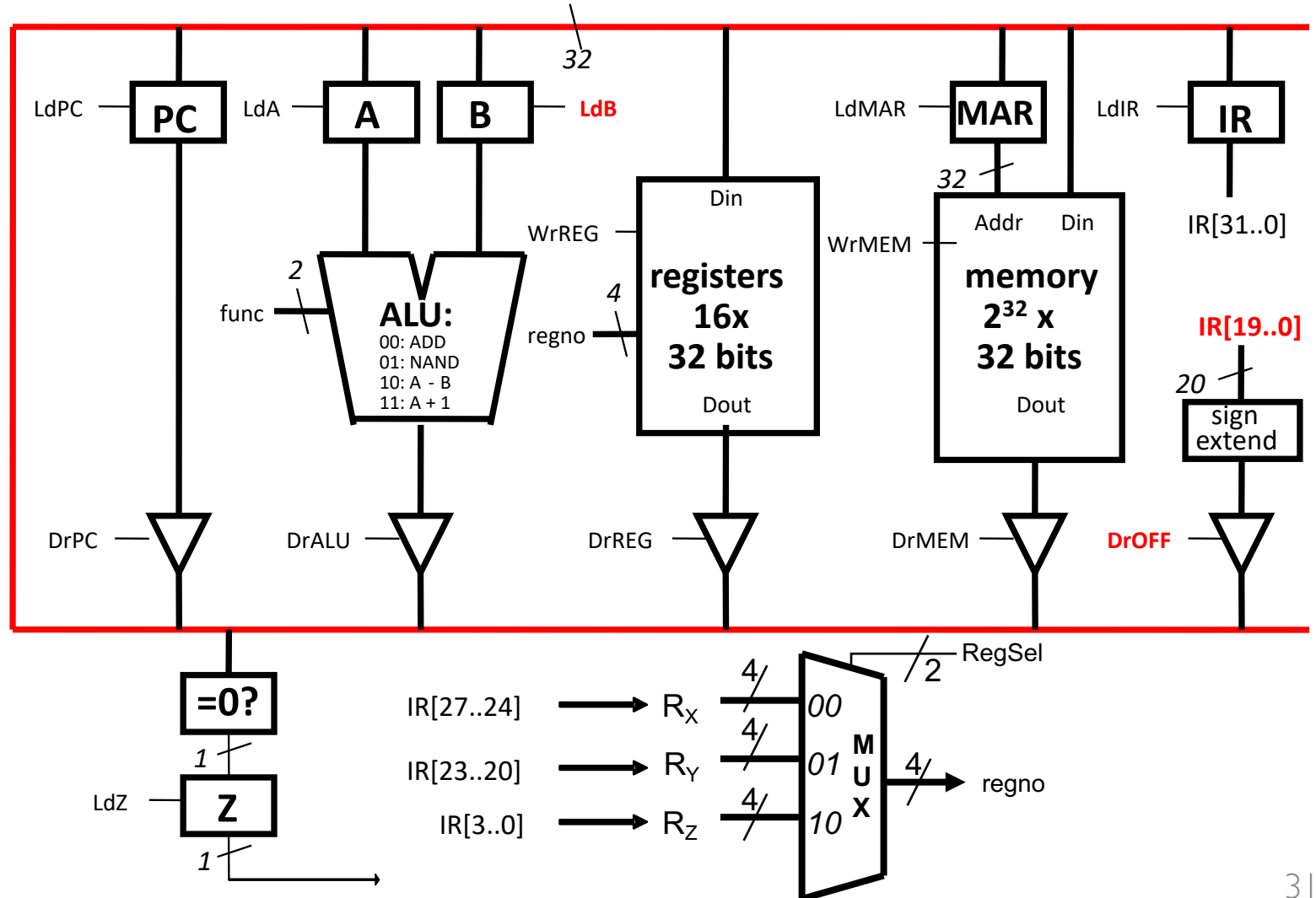


# beq5

Sign-extended offset  $\rightarrow$  B

Control signals needed:

- DrOff
- LdB



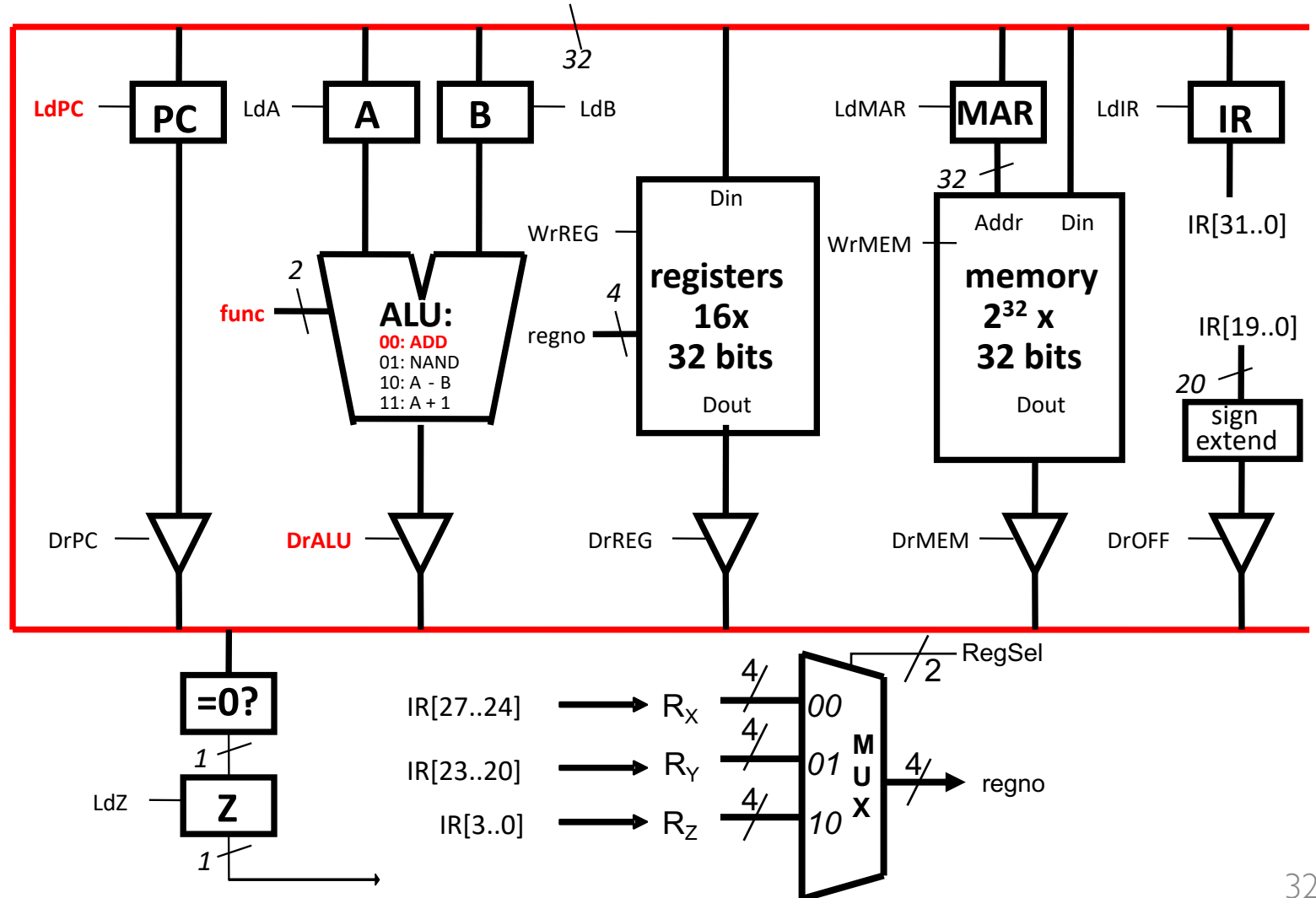
# beq6

$A + B \rightarrow PC$

Control signals needed:

- $func = 00$
- $DrALU$
- $LdPC$

Next state  
will be  
ifetch1



# And We Can Fill in Most ROM Values

	Drive Signals					Load Signals						Write Signals				
Current State	P C	ALU	Reg	MEM	OF F	PC	A	B	MAR	I R	Z	MEM	REG	func	Reg Sel	Next State
beq1			1				1								00	beq2
beq2			1					1							01	beq3
beq3		1									1				10	beq4 or ifetch1
beq4	1						1									beq5
beq5					1			1								beq6
beq6		1				1								00		ifetch1

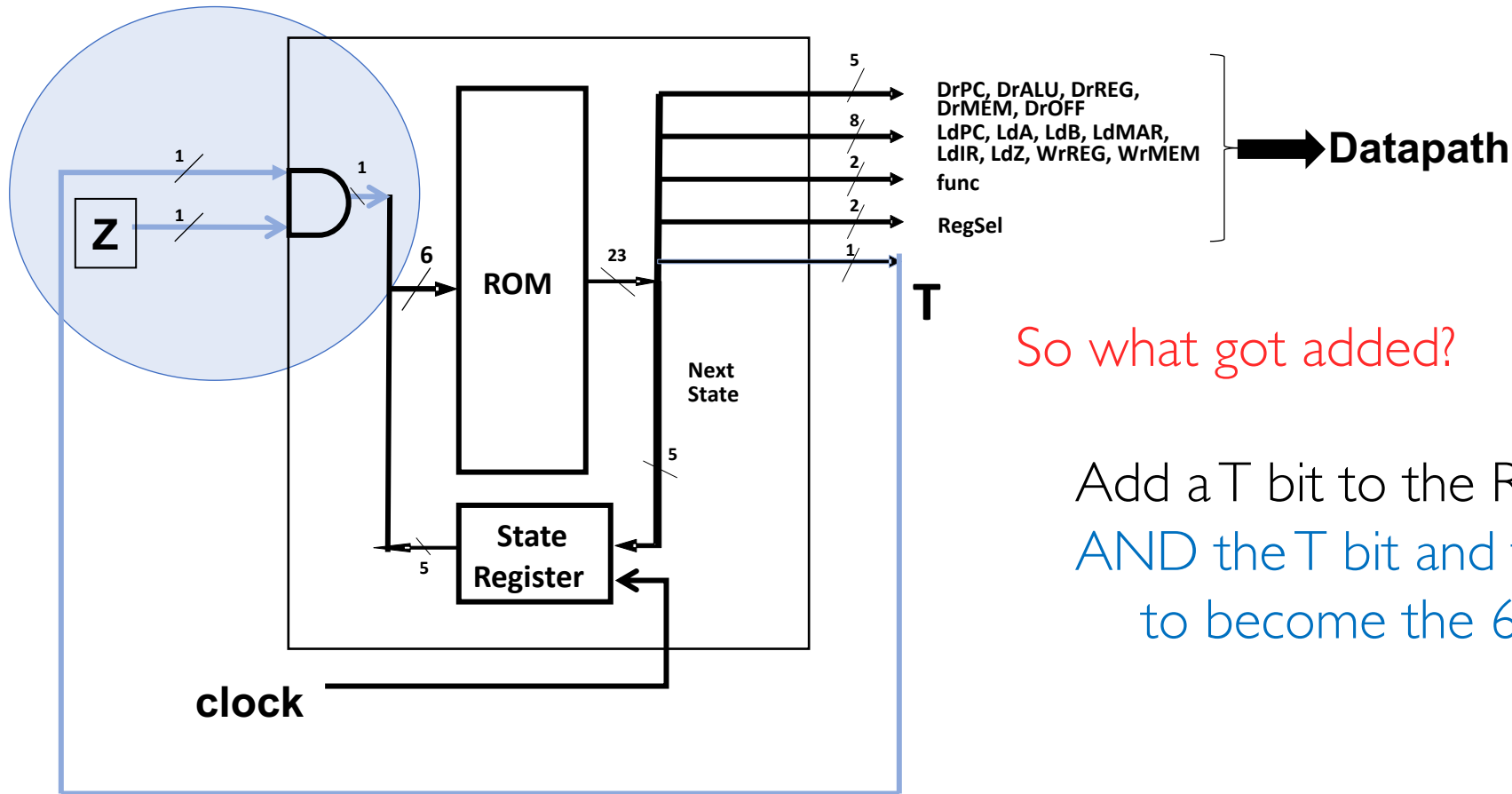


# But How Do We Handle that Decision?

---

- The steps in the ROM only allow for one “Next State”...
- So we must come up with some way to modify that Next State if we want to branch...
- What if we expand the ROM address by one bit so we can prepend a zero or one bit to the Next State if we want to test Z?
  - It doubles the ROM size, though... 2x number of words
  - For example, if Next State was 01000, we'd output 001000 UNLESS we wanted to test Z. Then we'd either output 101000 or 001000 by setting Z's value as the first bit of our next-state ROM address
- How can we do that?

# We're Going to Tweak the Control Unit



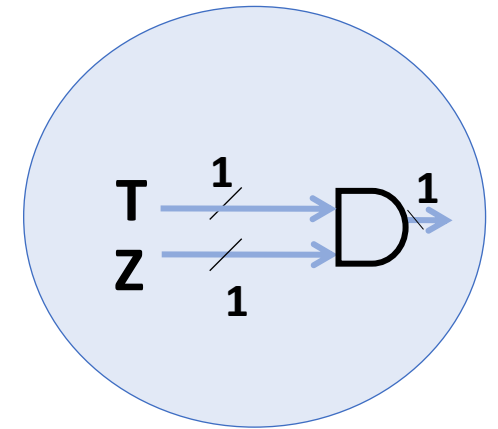
So what got added?

Add a T bit to the ROM  
AND the T bit and the Z bit  
to become the 6<sup>th</sup> address bit



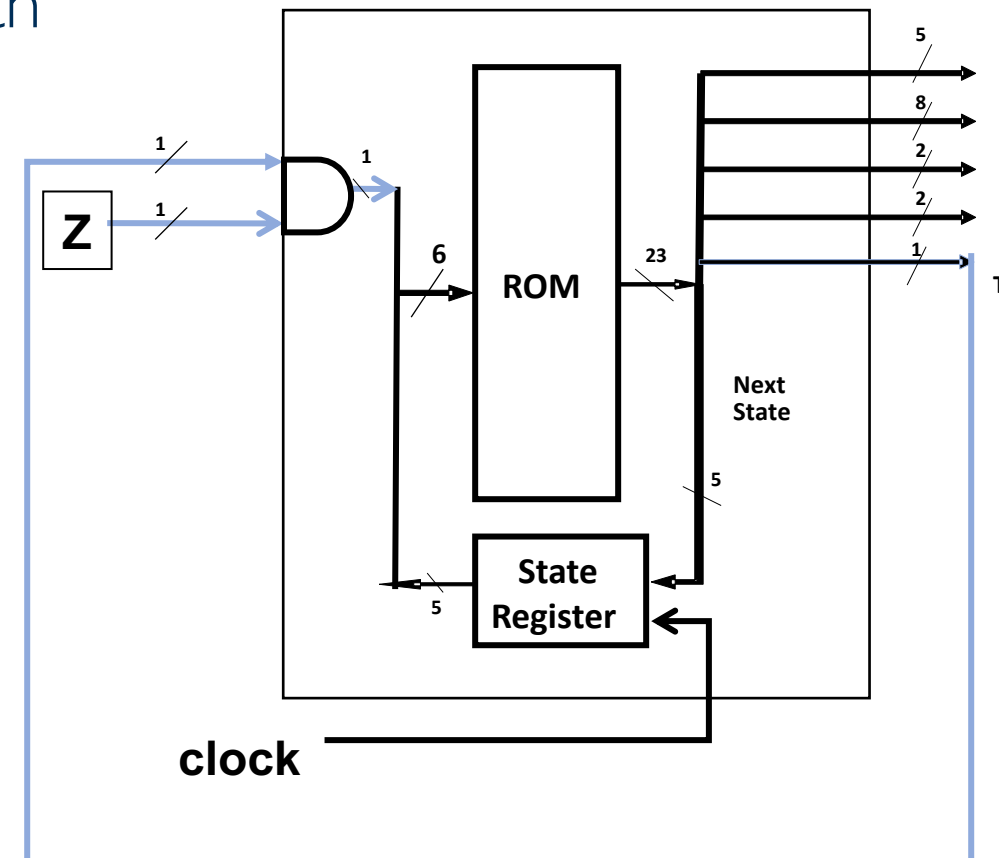
# The two-way modifier bit T is enabled...

- 0% A. At the first step of the BEQ execution
- 0% B. In the middle of the BEQ execution after operand comparison
- 0% C. Always
- 0% D. Never



# The Z bit input to the Modifier...

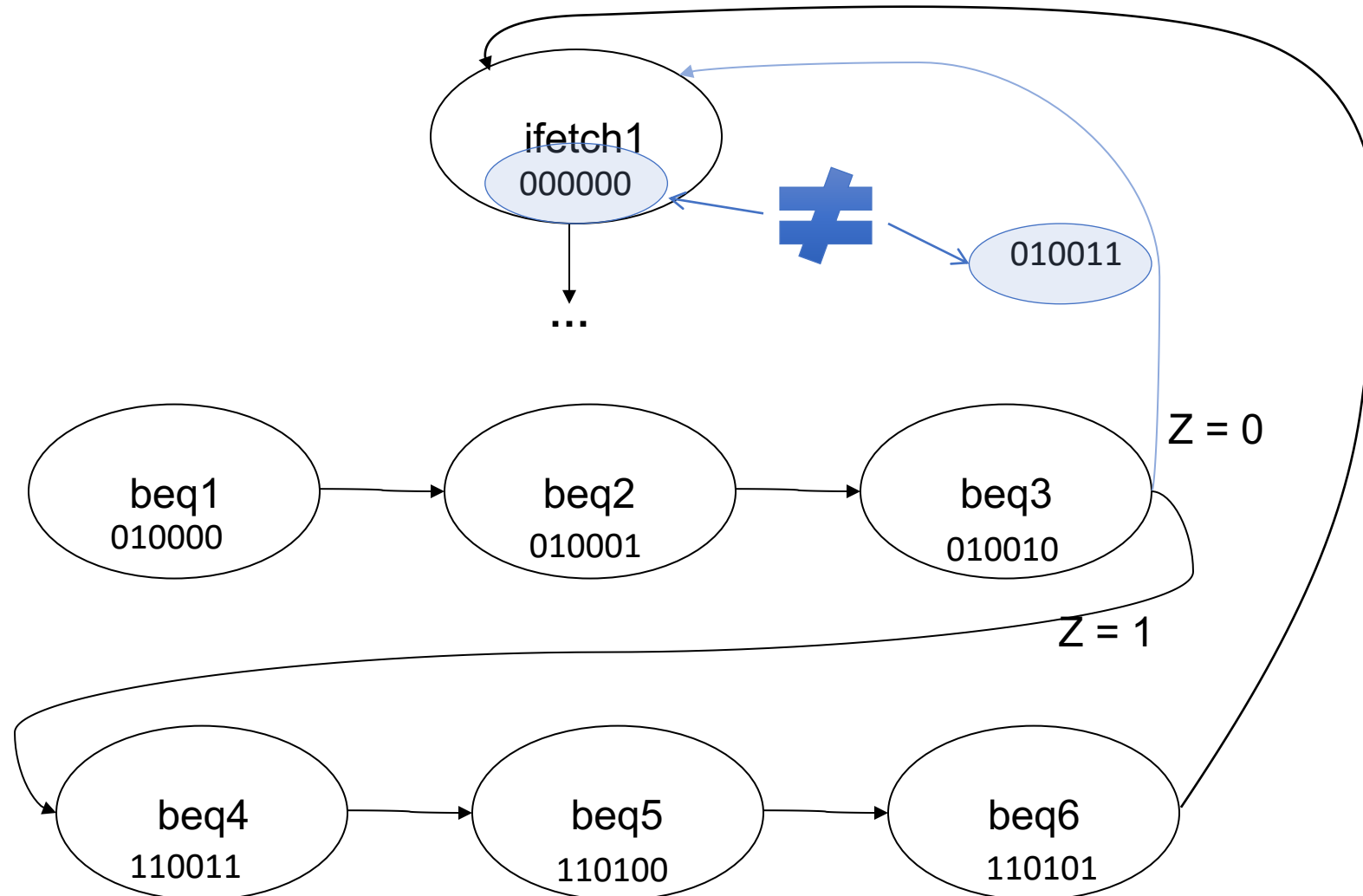
- 0% A. Is always zero
- 0% B. Is the output of the ROM itself
- 0% C. Is the output of the Z register in the datapath
- 0% D. Is 1 if the instruction is BEQ
- 0% E. No clue



# So We Need to Set T in the Microcode

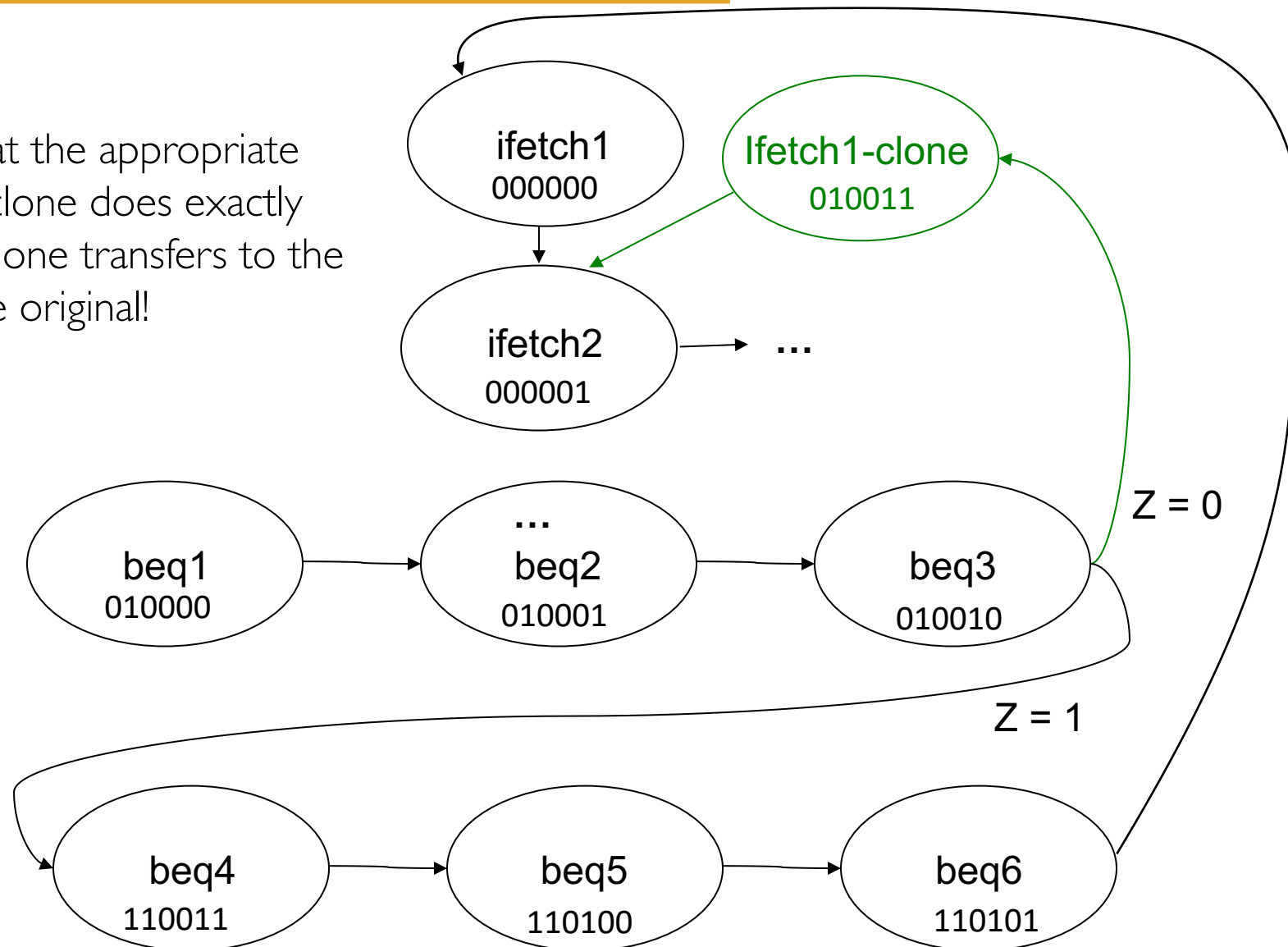
	Drive Signals					Load Signals						Write Signals					
Current State	PC	ALU	Reg	MEM	OFF	PC	A	B	MAR	IR	Z	MEM	REG	func	Reg Sel	T	Next State
010000			1				1								00		10001
010001			1					1							01		10010
010010		1									1				10	1	10011
110011	1						1									1	10100
110100					1			1								1	10101
110101		1				1								00			00000
010011	Here we need to fill in the <i>contents</i> of ROM location ifetch1. Why?																

# What's the Problem with the Z Branch?



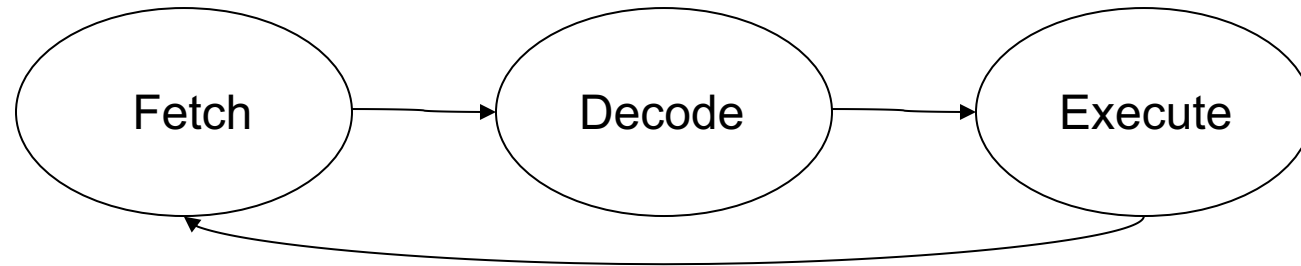
# An Old Microcode Trick!

We duplicate ifetch1 at the appropriate ROM address (so its clone does exactly the same thing). The clone transfers to the same next-state as the original!



# Back to basic State Diagram


---





# FETCH macro-state

---

- Need to do
    - We need to send PC to the memory
    - Read the memory contents
    - Bring the memory contents read into the IR
    - Increment the PC
    - (And decode the opcode by branching to the right execution state)
  - Microstates to accomplish
    - ifetch1
      - $PC \rightarrow MAR$
    - ifetch2
      - $MEM[MAR] \rightarrow IR$
    - ifetch3
      - $PC \rightarrow A$
    - ifetch4
      - $A+1 \rightarrow PC$
- Simplify
- 
- ifetch1
    - $PC \rightarrow MAR$
    - $PC \rightarrow A$
  - ifetch2
    - $MEM[MAR] \rightarrow IR$
  - ifetch3
    - $A+1 \rightarrow PC$

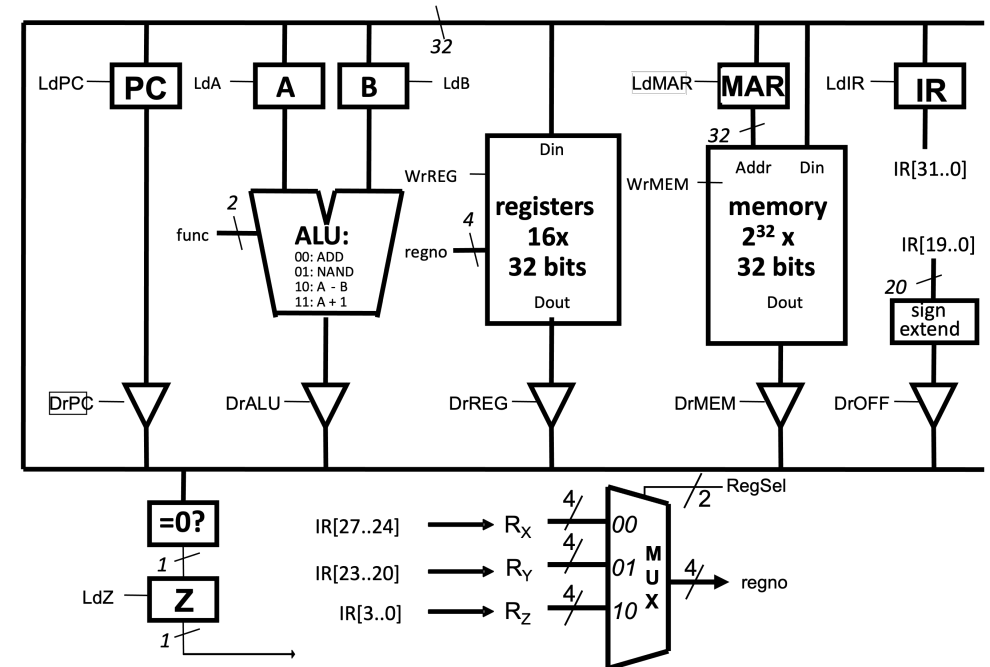
# FETCH state: Adding in control signals

- ifetch1

- PC → MAR
- PC → A
- Control signals needed:
  - DrPC
  - LdMAR
  - LdA

- ifetch2

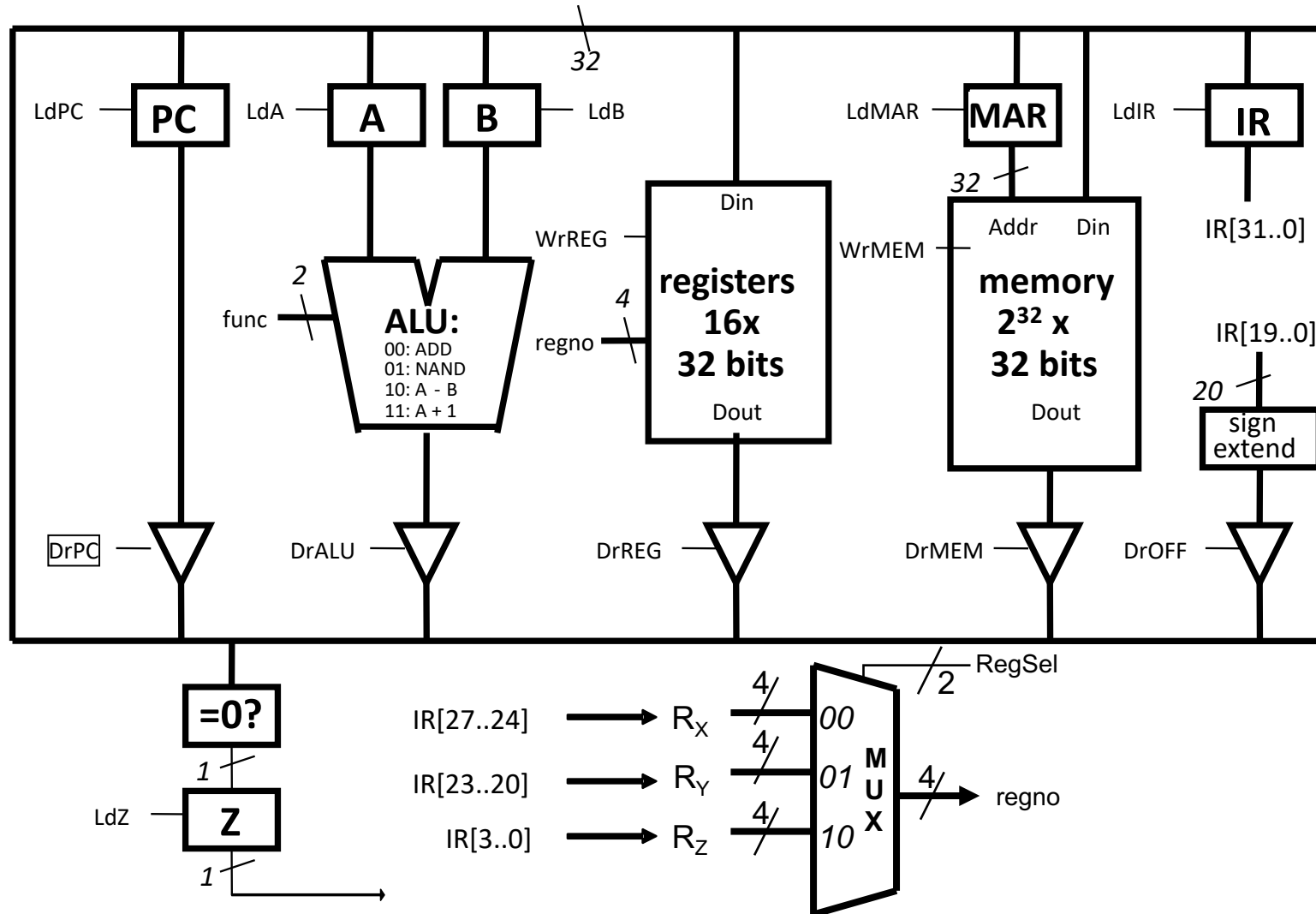
- MEM[MAR] → IR
- Control signals needed:
  - DrMEM
  - LdIR



- ifetch3

- A + I → PC
- Control signals needed:
  - func = 11
  - DrALU
  - LdPC

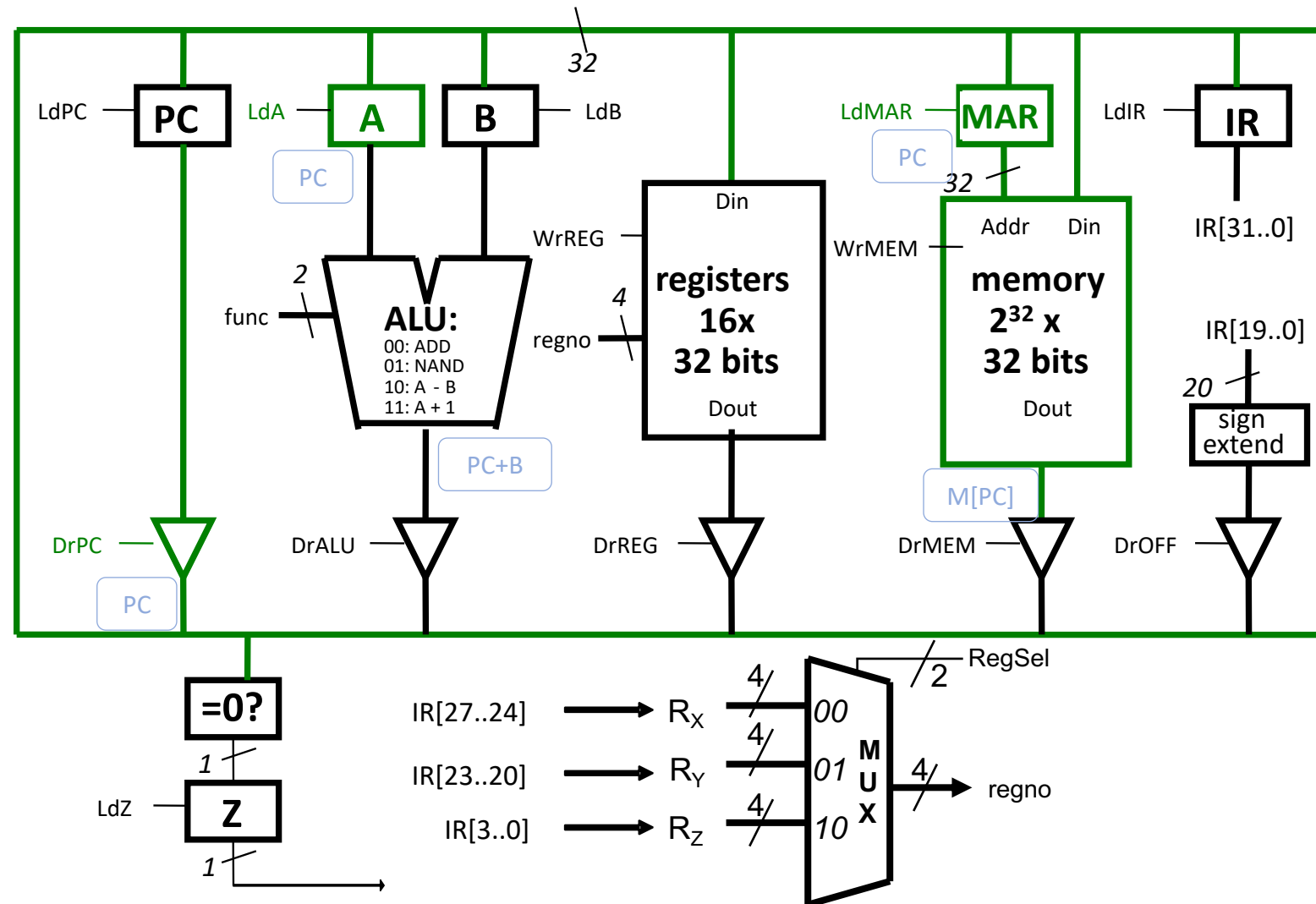
# Before ifetch I



# Implementing ifetch1 (end of clock 1)

- $PC \rightarrow MAR$
- $PC \rightarrow A$
- Control signals needed:

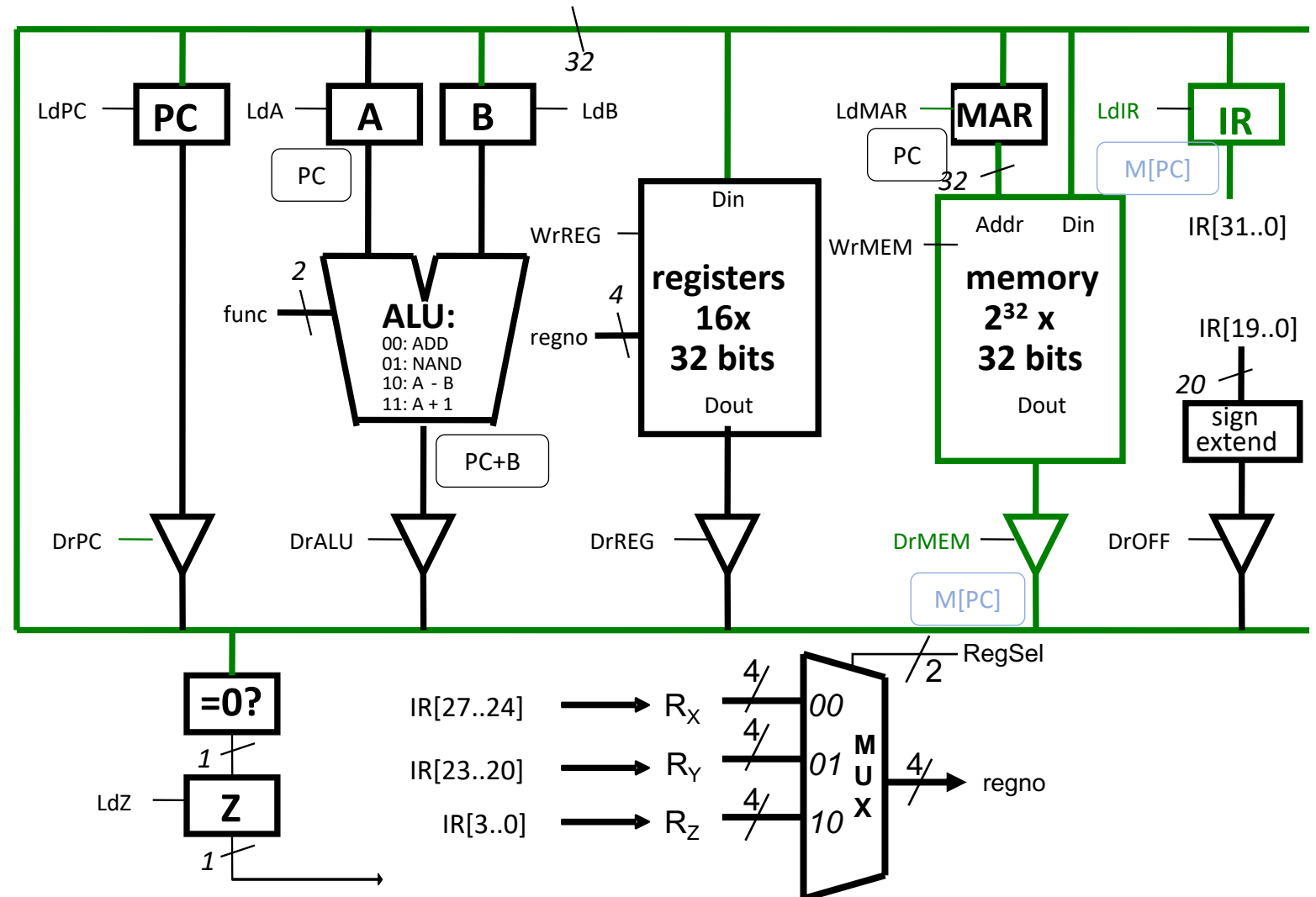
DrPC  
LdMAR  
LdA  
Others=0



# Implementing ifetch2 (end of clock 2)

- MEM[MAR] → IR
- Control signals needed:

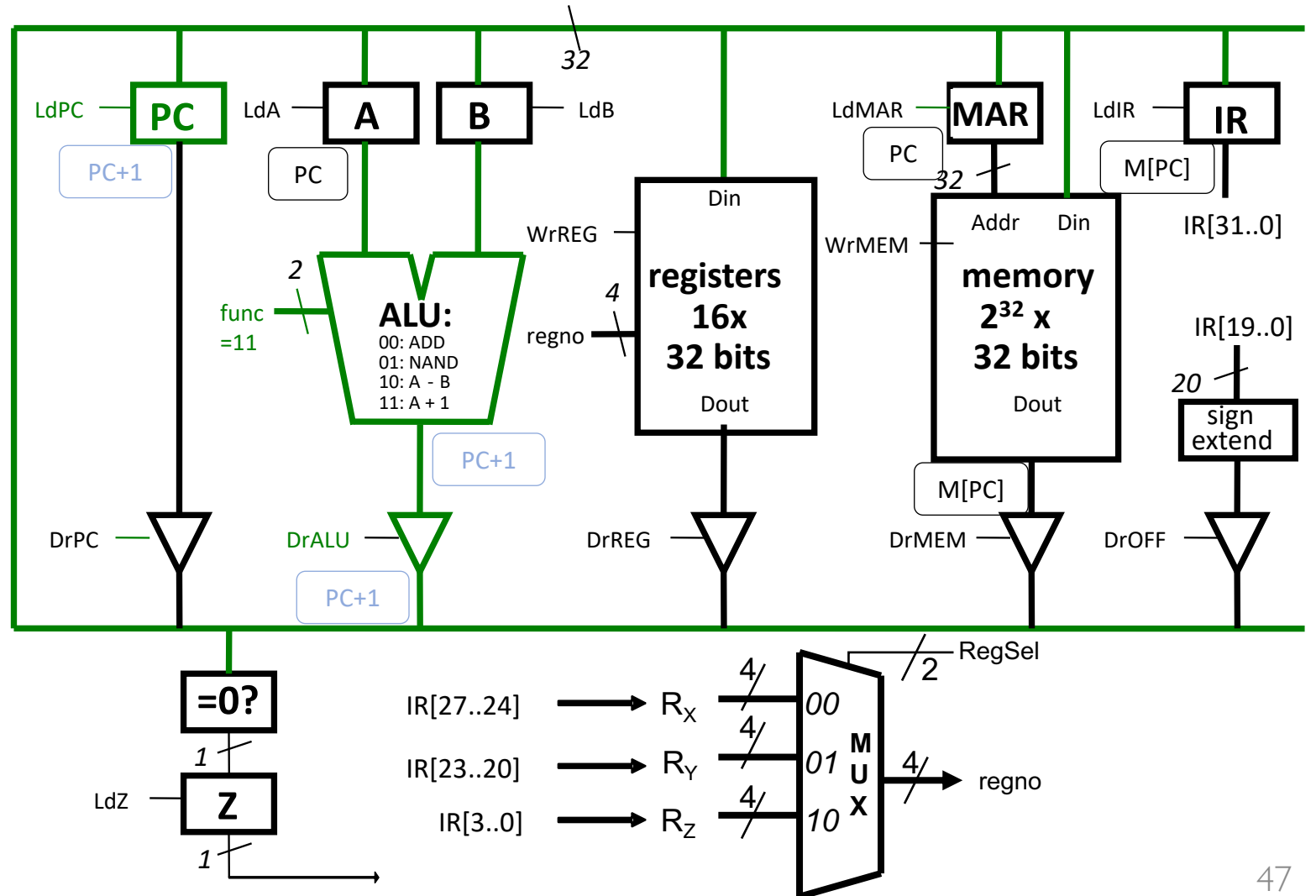
DrMem  
LdIR  
Others=0



# Implementing ifetch3 (end of clock 3)

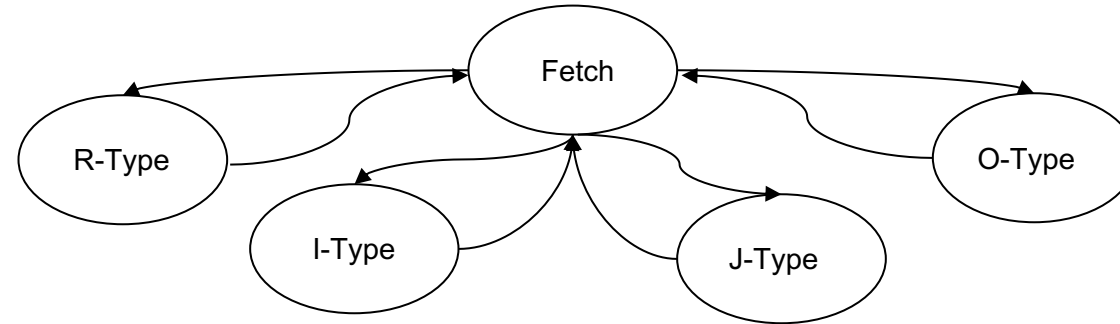
- $A + I \rightarrow PC$
- Control signals needed:

func=11  
DrALU  
LdPC  
Others=0



# DECODE State

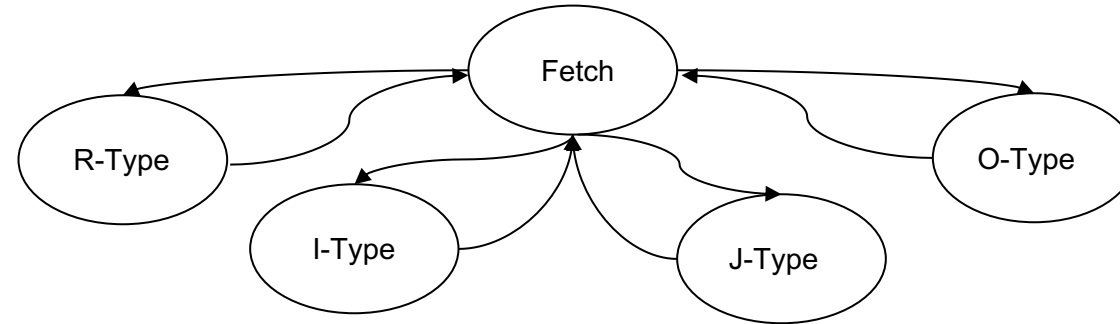
---



- Decode is a MULTIWAY branch!
- We can't encode this in Next State!
- Actually, we can... let's reuse the same trick we applied for BEQ!

# DECODE State

---



- On the last step of ifetch, we'll set the top 4 bits of our ROM address to the opcode that's in  $IR[31:28]$ !
- OK. How do we do that?

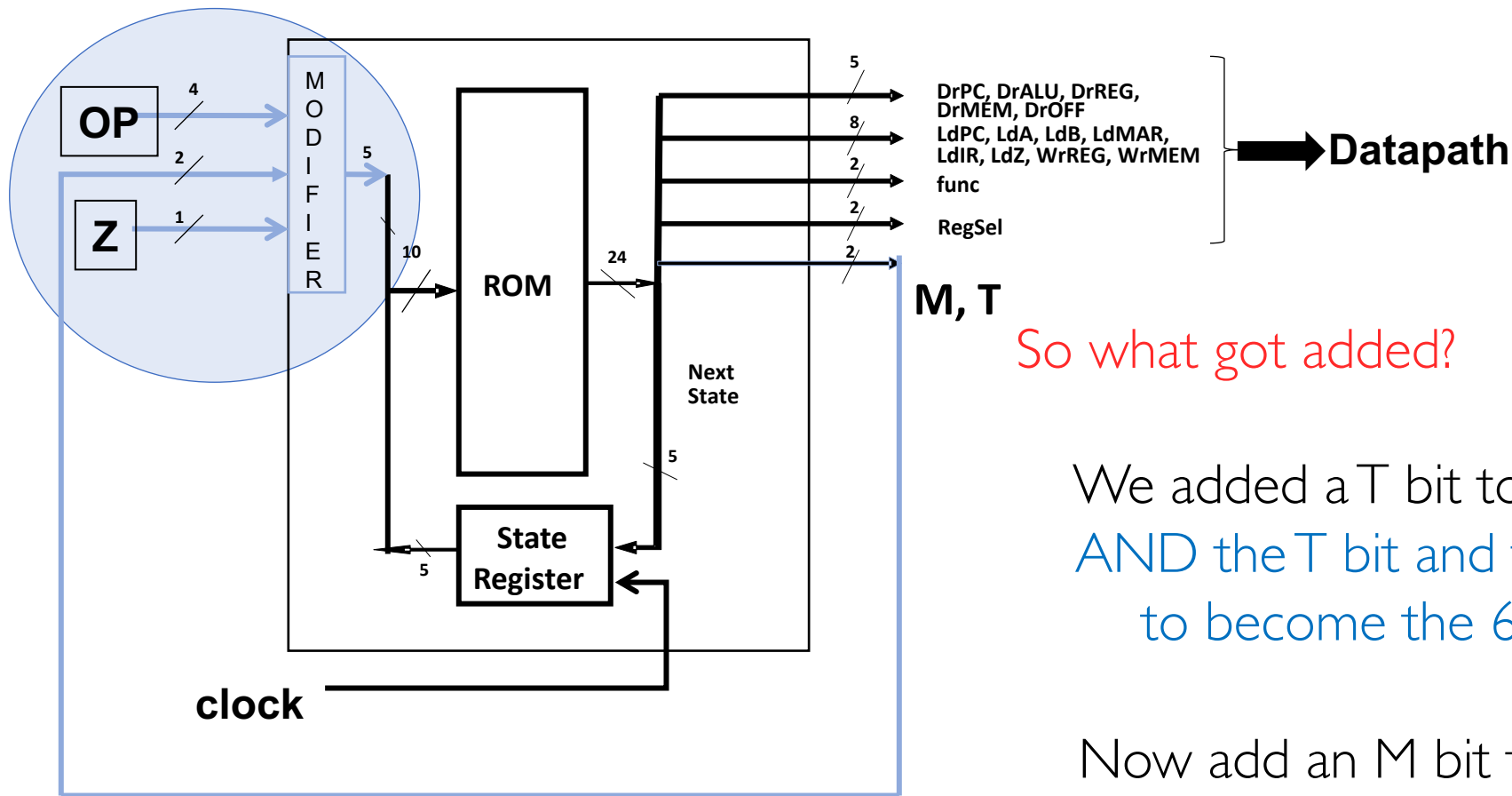


# Let's extend the Control Unit again

---

- We expanded the ROM address from 5 to 6 bits for BEQ
- Let's extend it from 6 to 10 bits and use the opcode as the top 4 bits of the ROM address
- That means if the opcode is 0010 and next-state is 000011, then if we use the opcode bits, we would use 0010 000011 as the next state so the microcode to execute 0010 would start at that address
- This gives us a many-way branch!

# Updated Control Unit

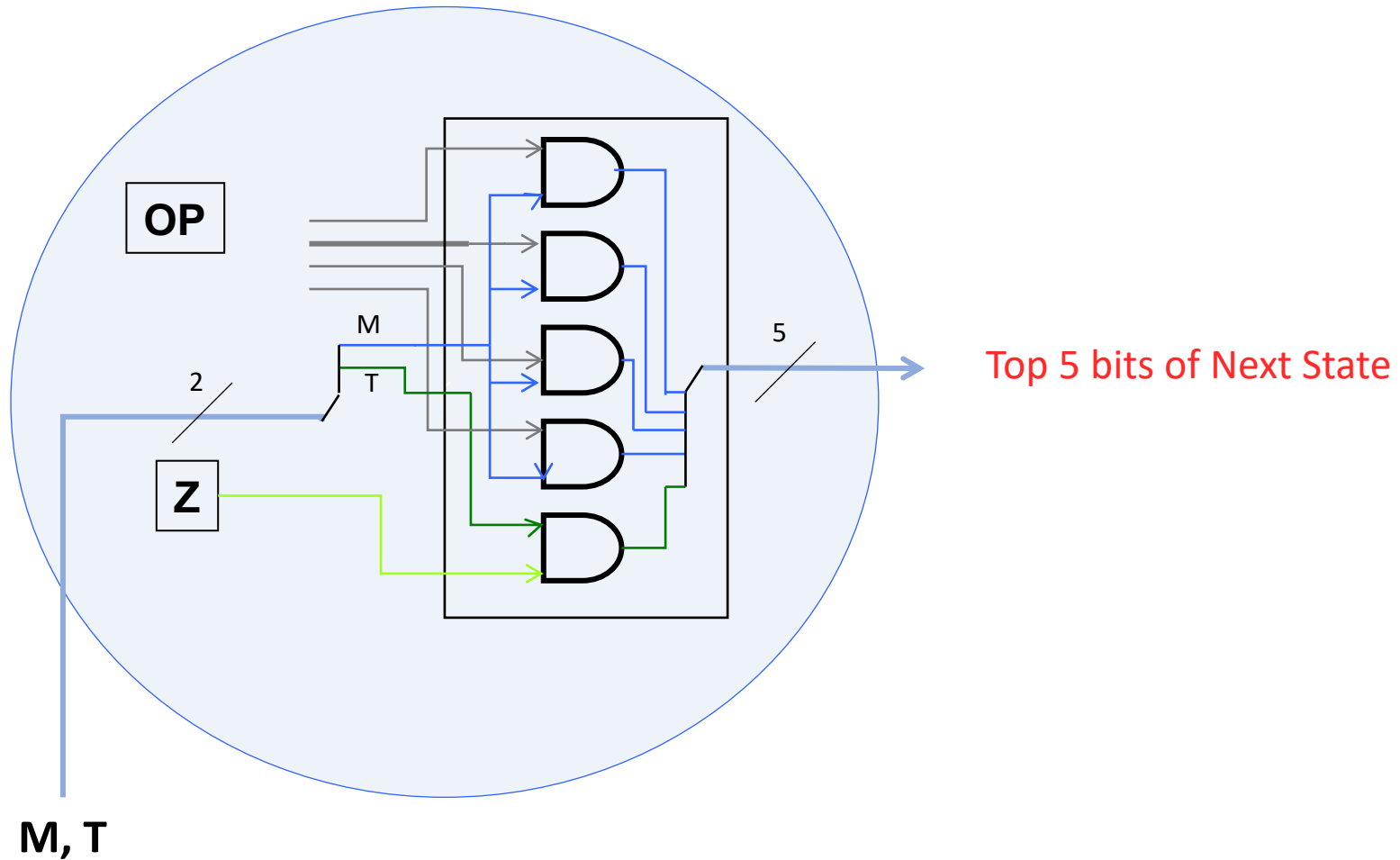


So what got added?

We added a T bit to the ROM for BEQ  
AND the T bit and the Z bit  
to become the 6<sup>th</sup> address bit

Now add an M bit to the ROM  
AND the M bit and the OP from bits IR[31:28]  
to become the 10<sup>th</sup>-7<sup>th</sup> address bits

# What's in MODIFIER?

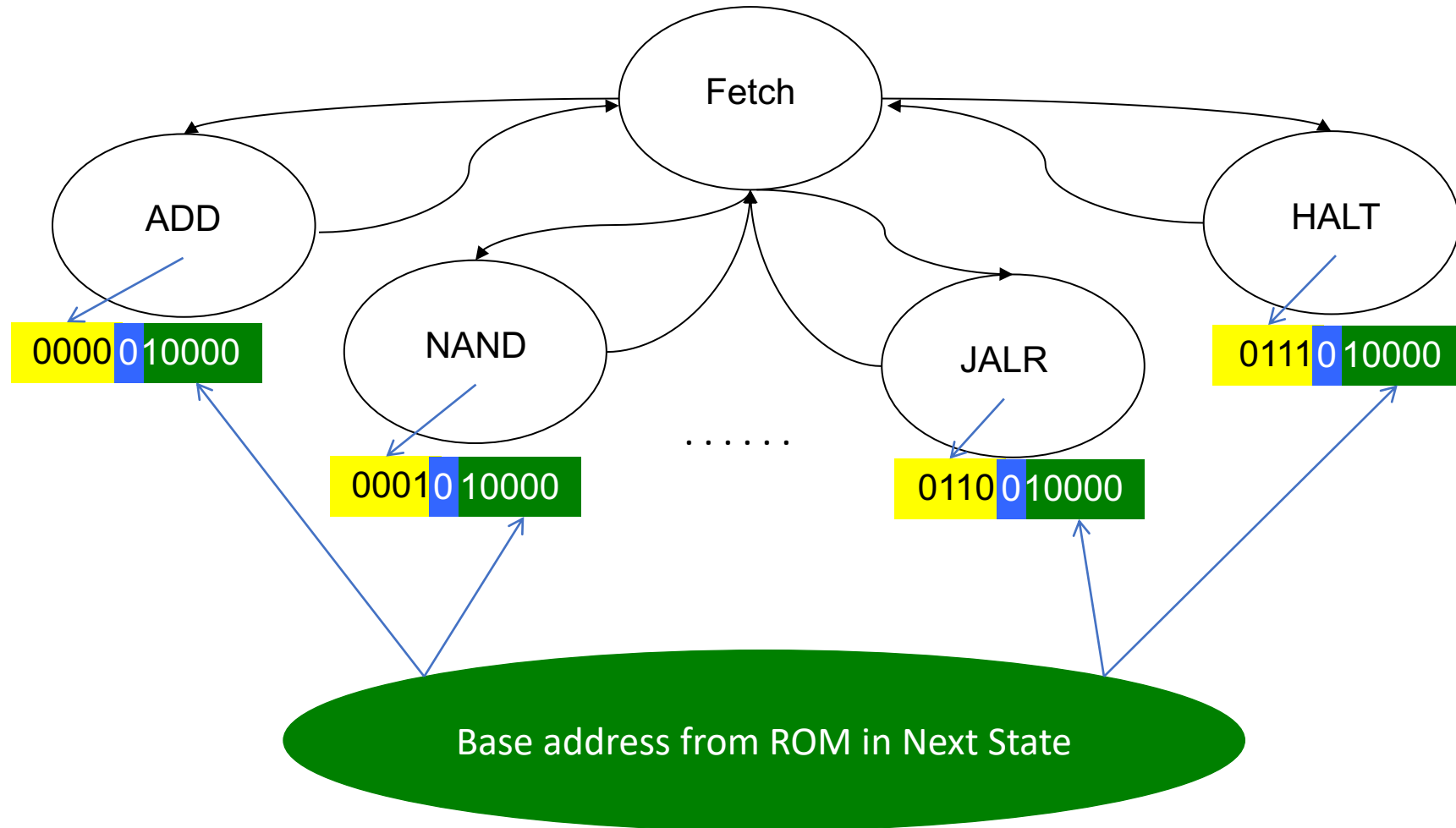


# Let's Encode the 3 ifetch States

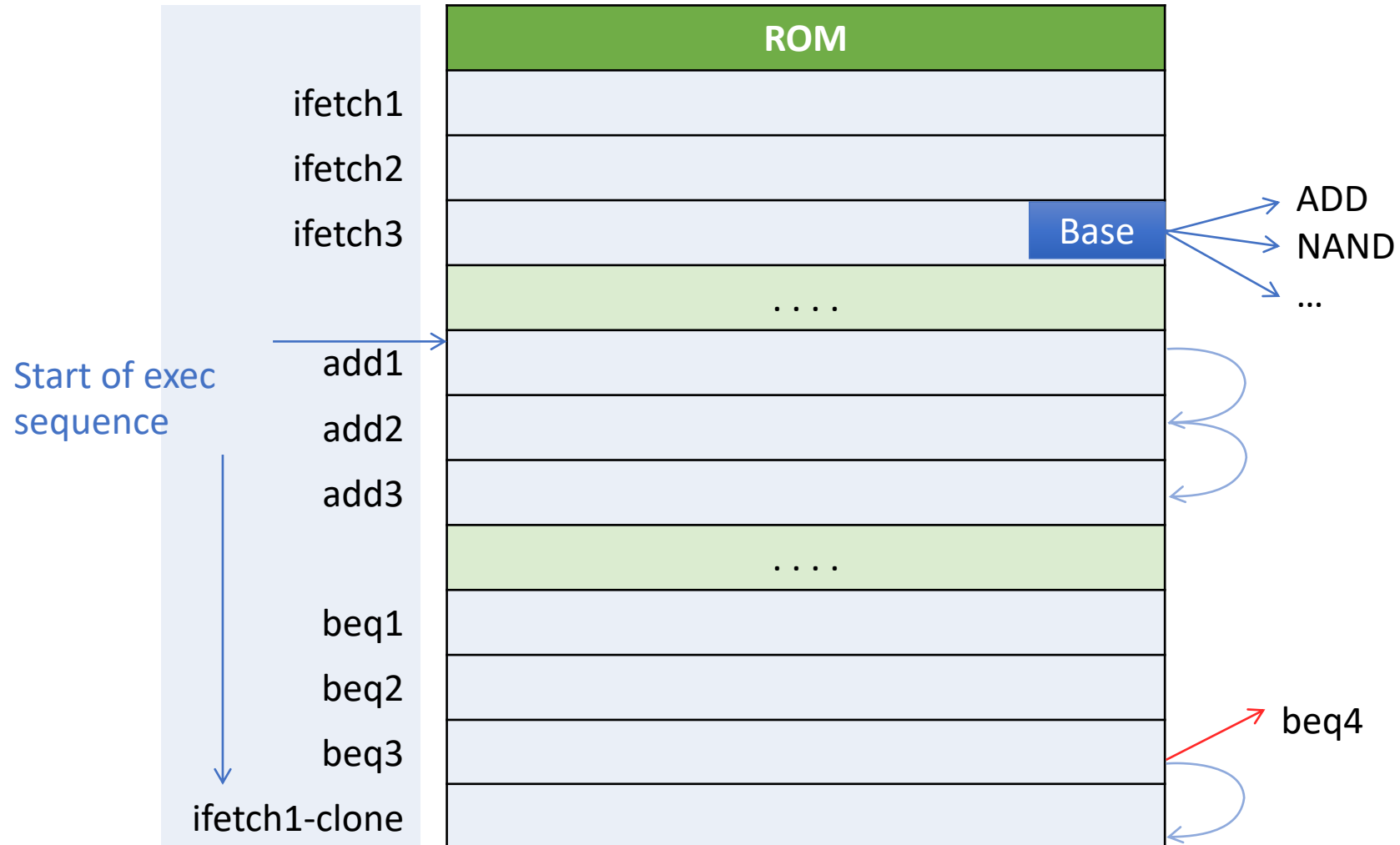
	Drive Signals					Load Signals						Write Signals						
Current State	PC	ALU	Reg	MEM	OFF	PC	A	B	MAR	IR	Z	MEM	REG	func	Reg Sel	<b>M</b>	T	Next State
000000000	1						1		1									00001
000000001				1						1								00010
000000010		1				1								11		1		10000

- So how do we make it take that multi-way branch?
- Just set the M bit at 0000000010!

# Next State After the Last State of Fetch



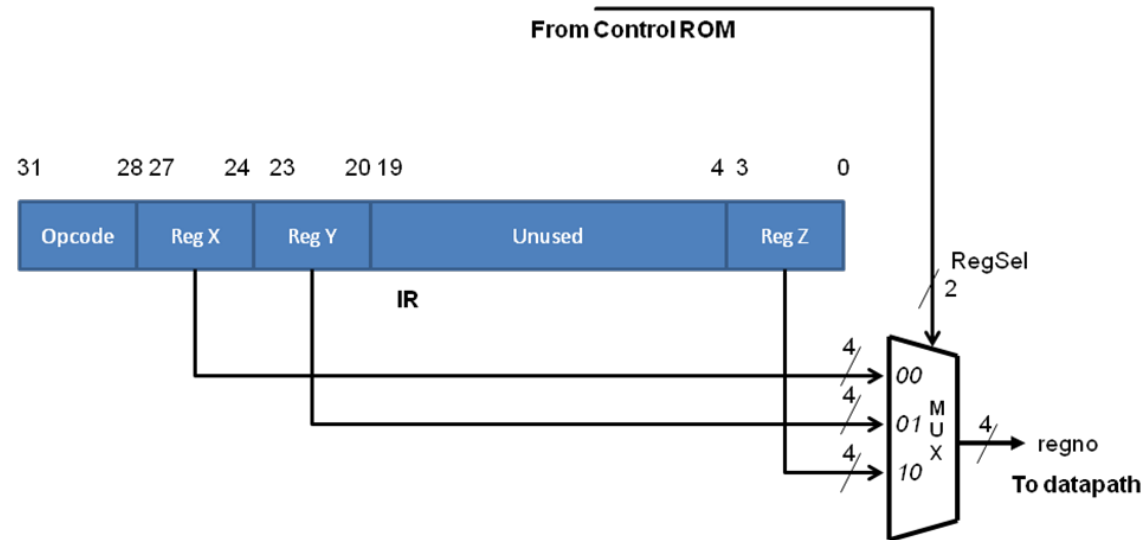
# ROM Contents



# EXECUTE state: ADD instruction



$$R_X \leftarrow R_Y + R_Z$$



# EXECUTE state: ADD instruction

add1

$R_y \rightarrow A$

Control signals needed:

RegSel = 01

DrREG

LdA

add2

$R_z \rightarrow B$

Control signals needed:

RegSel = 10

DrREG

LdB

add3

$A+B \rightarrow R_x$

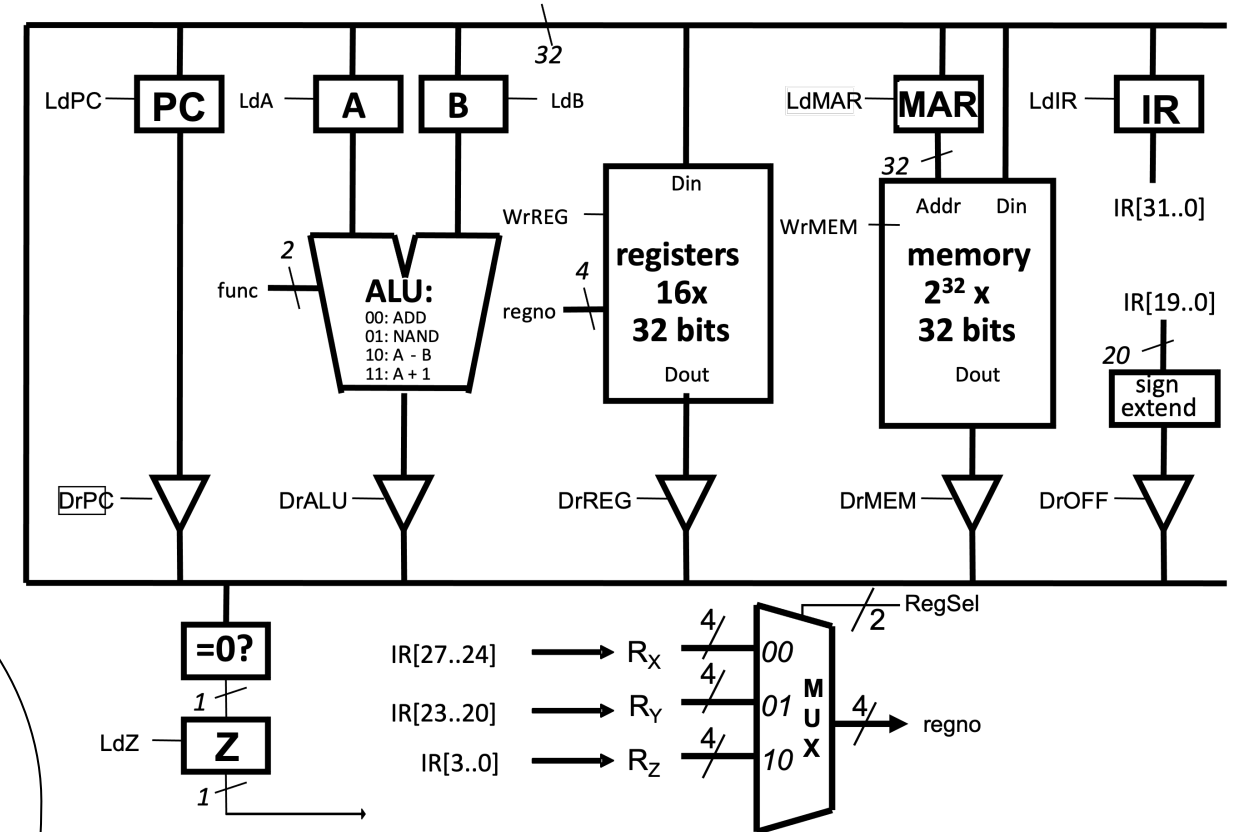
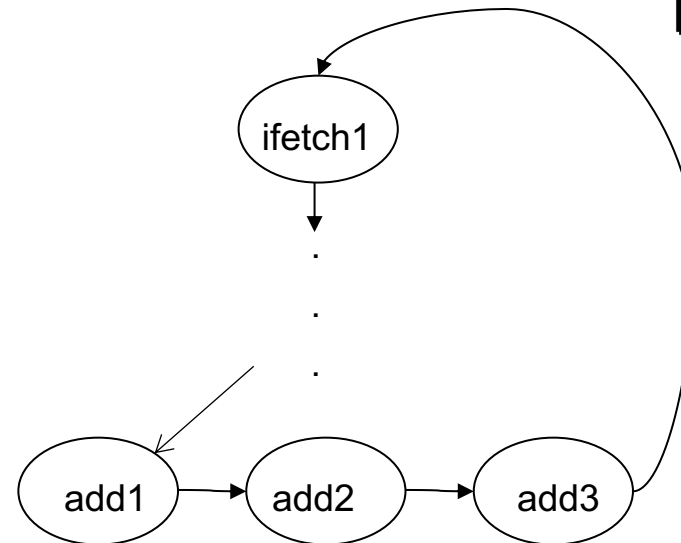
Control signals needed:

func = 00

DrALU

RegSel = 00

WrREG





# EXECUTE state: ADD instruction

add1

$R_y \rightarrow A$

Control signals needed:

RegSel = 01

DrREG

LdA

add2

$R_z \rightarrow B$

Control signals needed:

RegSel = 10

DrREG

LdB

add3

$A+B \rightarrow R_x$

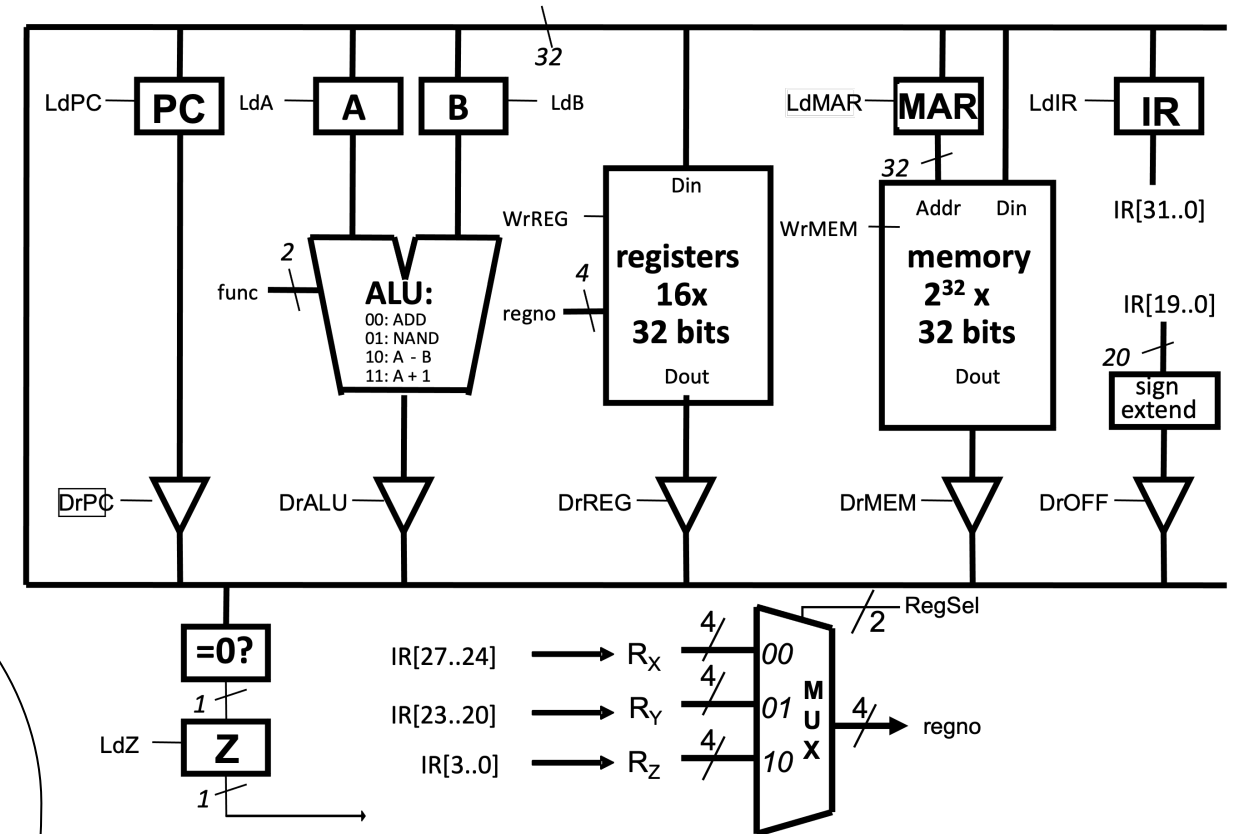
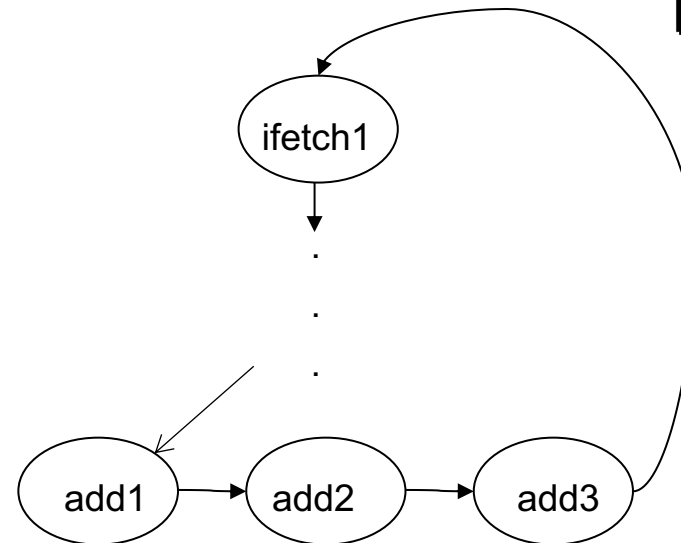
Control signals needed:

func = 00

DrALU

RegSel = 00

WrREG



What must be changed in ADD to implement NAND?

# EXECUTE state: JALR instruction

JALR instruction does the following:

$$R_Y \leftarrow PC + 1$$

$$PC \leftarrow R_X$$

jalr1

$$PC \rightarrow R_Y$$

Control signals needed:

DrPC

RegSel = 01

WrREG

jalr2

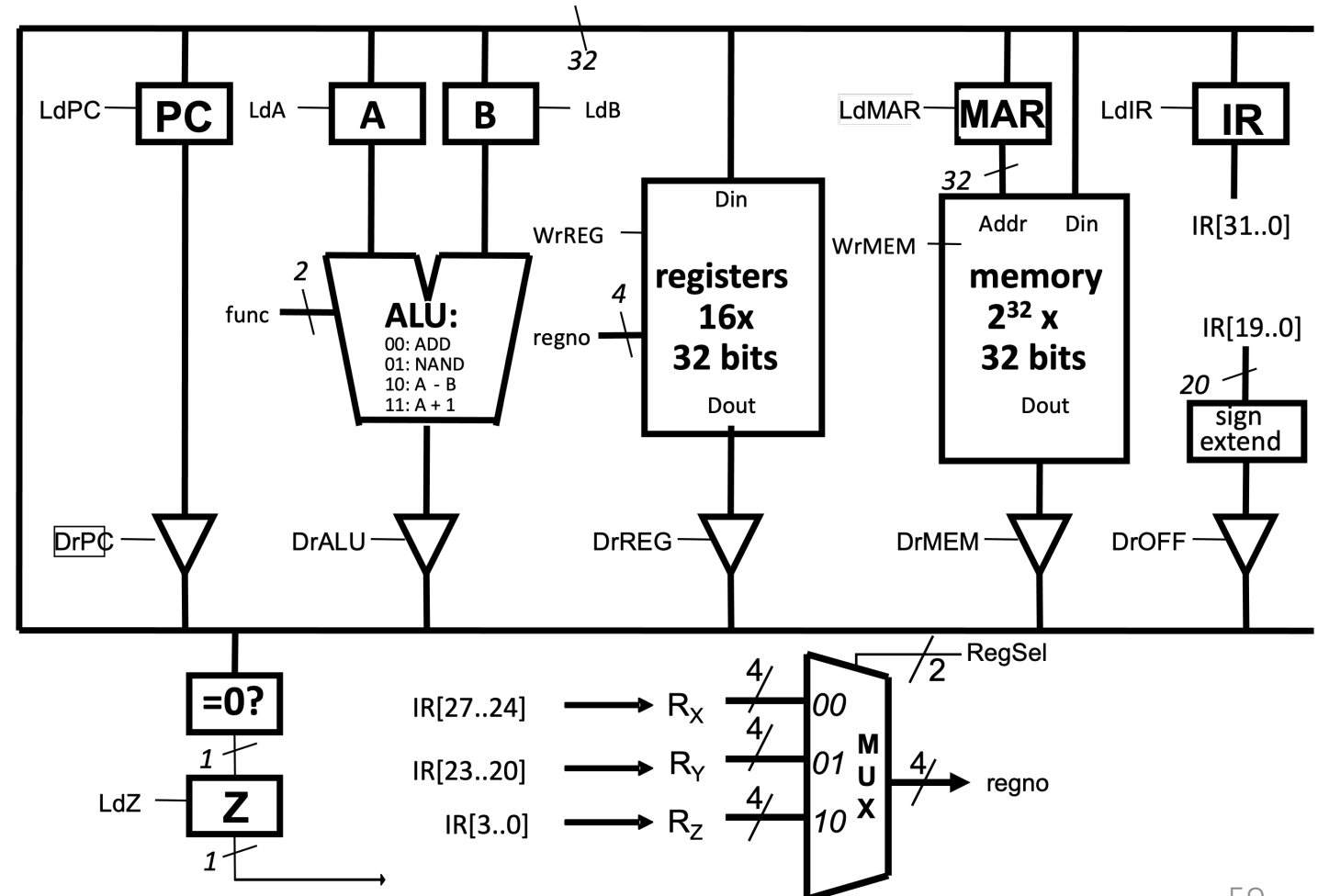
$$R_X \rightarrow PC$$

Control signals needed:

RegSel = 00

DrREG

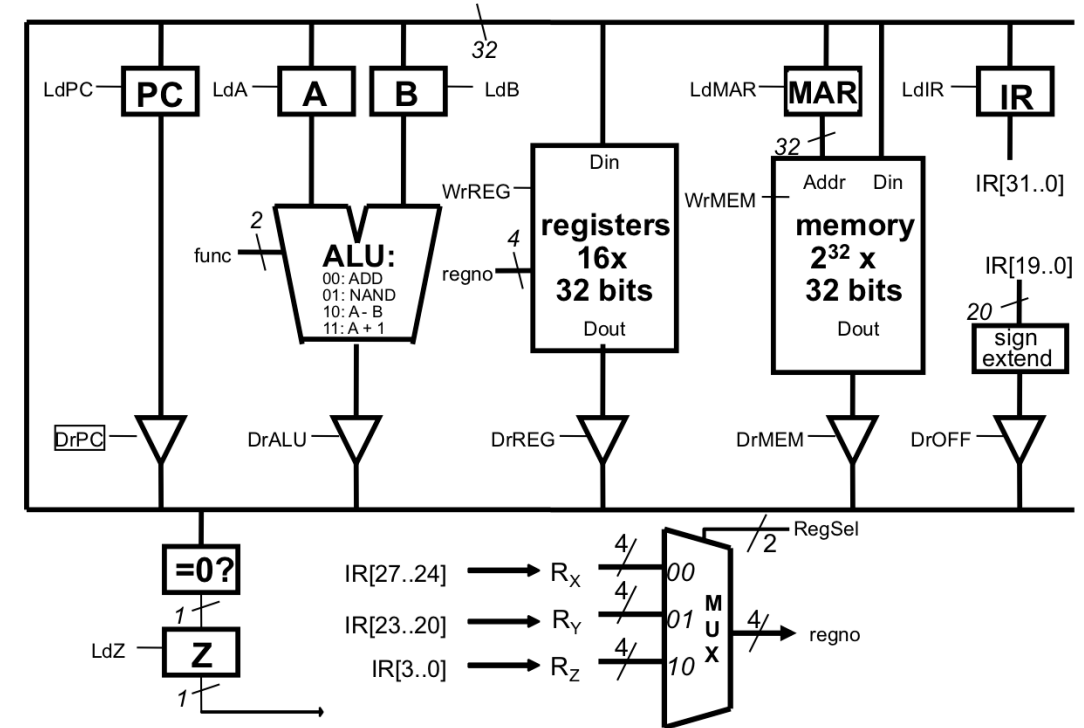
LdPC





# Question

When all of the control signals are zero in the LC-2200 datapath, what value is being presented by the ALU to DrALU?



- 0% A.  $A + B$
- 0% B. The value of one of the registers
- 0% C. Zero
- 0% D. Floating

# Alternative Style of Control Unit Design

---

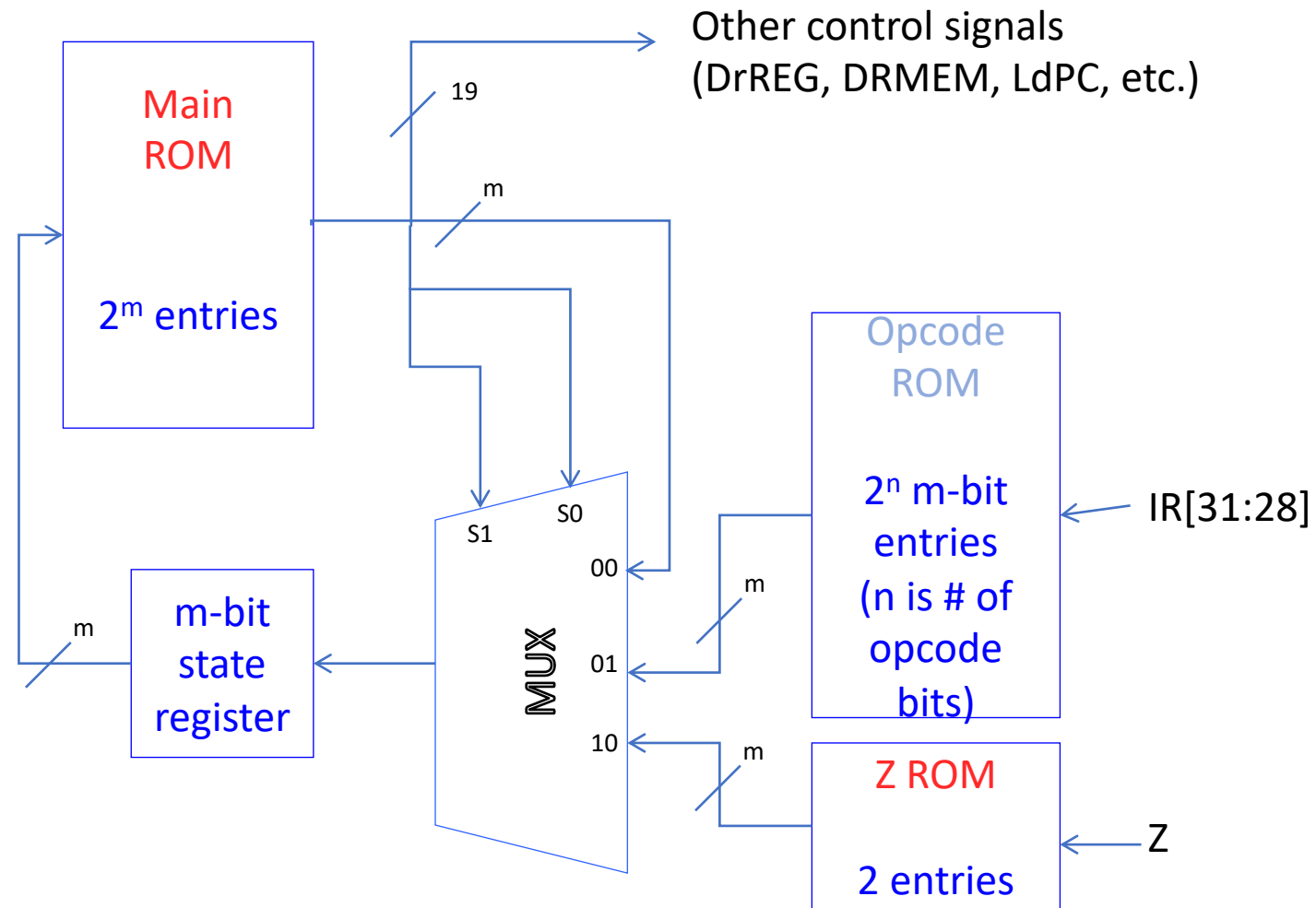
A number of different approaches may be used to implement the Control Unit

# Microprogrammed Control

---

- As presented our design works
- Problem: Too slow
  - Solution: Pre-fetch the next microinstruction
- Problem: Too much memory required
  - Solution: OR the opcode with the next state value 10000 instead of pre-pending it
  - Solution: Use more than one ROM and more sophisticated Decode/BEQ logic
    - One set of ROMs for which state comes next
    - One for what the control outputs should be in the state

# 3-ROM Microsequencer



# Space/Time Tradeoff

---

- Flat ROM
  - More space (since we increased the ROM by a factor of 32 for the occasional address modifiers, but have extra ROM space)
  - Faster since only one ROM access in each microinstruction
- Micro sequencer (3-ROM control unit)
  - Less space (main ROM much smaller than Flat ROM)
  - Slower since additional ROM access in every clock cycle

# Hardwired Control

---

- State machine can be represented as sequential logic truth table
- Thus can be implemented using normal combinational logic or FPGA
- Can produce boolean function for each control signal
  - E.g.,  $DrPC = ifetchI + jalrI + beq4 + \dots$



Control Regime	Pros	Cons	Comment	When to use	Examples
Micro-programmed	Simplicity, maintainability, flexibility Rapid prototyping	Potential for space and time inefficiency	Space inefficiency may be mitigated with vertical microcode Time inefficiency may be mitigated with prefetching	For complex instructions, and for quick non-pipelined prototyping of architectures	PDP 11 series, IBM 360 and 370 series, Motorola 68000, complex instructions in Intel x86 architecture
Hardwired	Amenable for pipelined implementation Potential for higher performance	Potentially harder to change the design Longer design time	Maintainability can be increased with the use of structured hardware such as PLAs and FPGAs	For High performance pipelined implementation of architectures	Most modern processors including Intel Pentium series, IBM PowerPC, MIPS