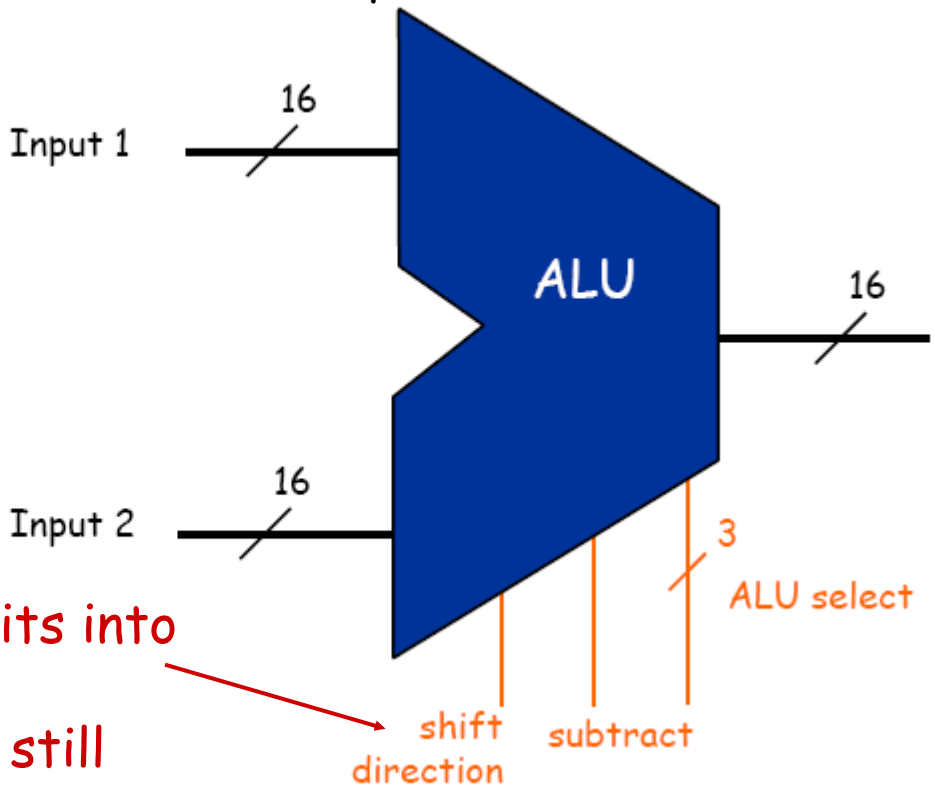
Recap: ALU

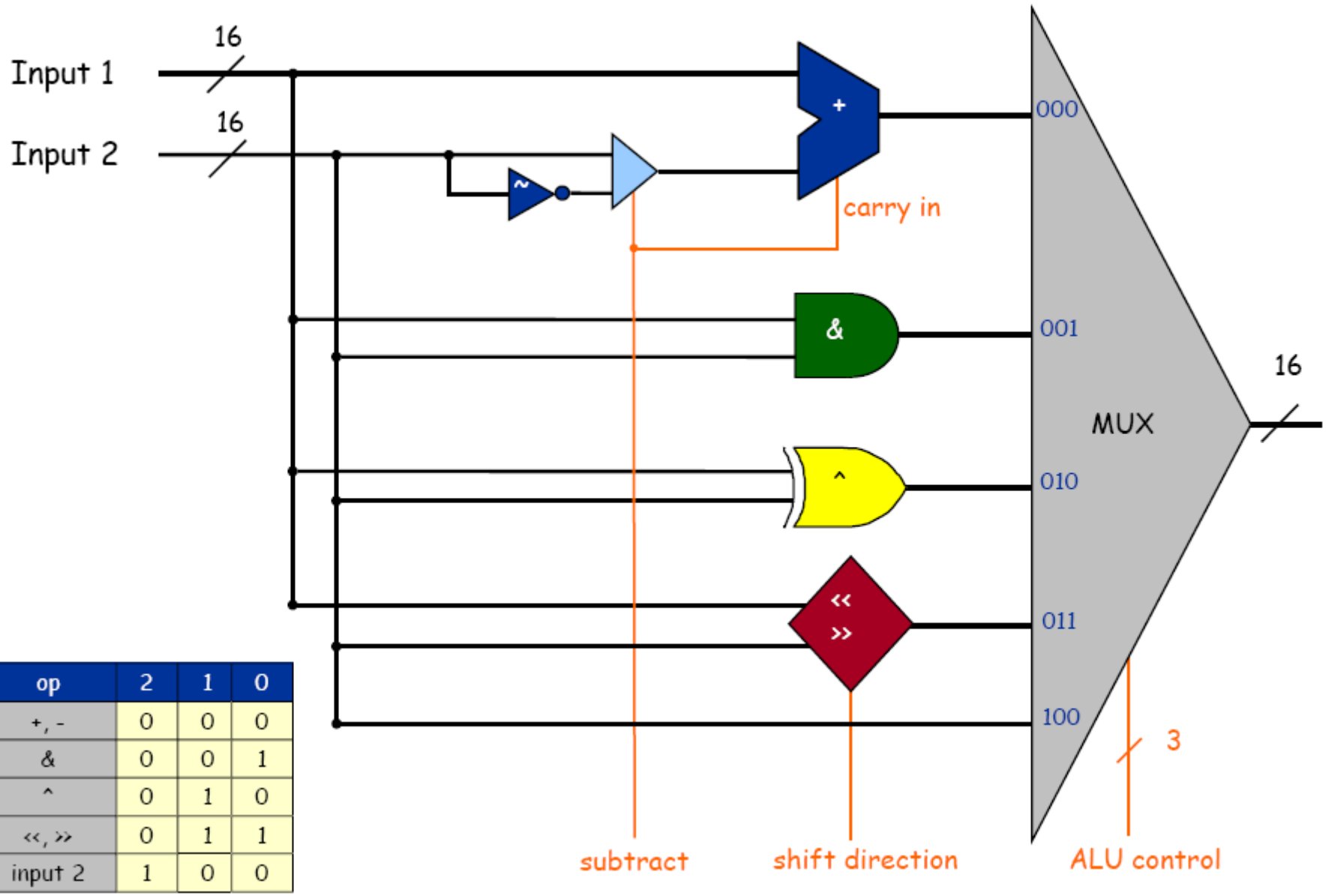
- Big combinational logic (16-bit bus)
- Add/subtract, and, xor, shift left/right, copy input 2
- A 3-bit control for 5 primary ALU operations
  - ALU performs operations in parallel
  - Control wises select which result ALU outputs

op	2	1	0
+, -	0	0	0
&	0	0	1
^	0	1	0
<<, >>	0	1	1
input 2	1	0	0



Can we combine these 5 bits into 3 bits for 7 operations?
Yes, you can. But, you will still need 5 bits at the end.

Recap: ALU

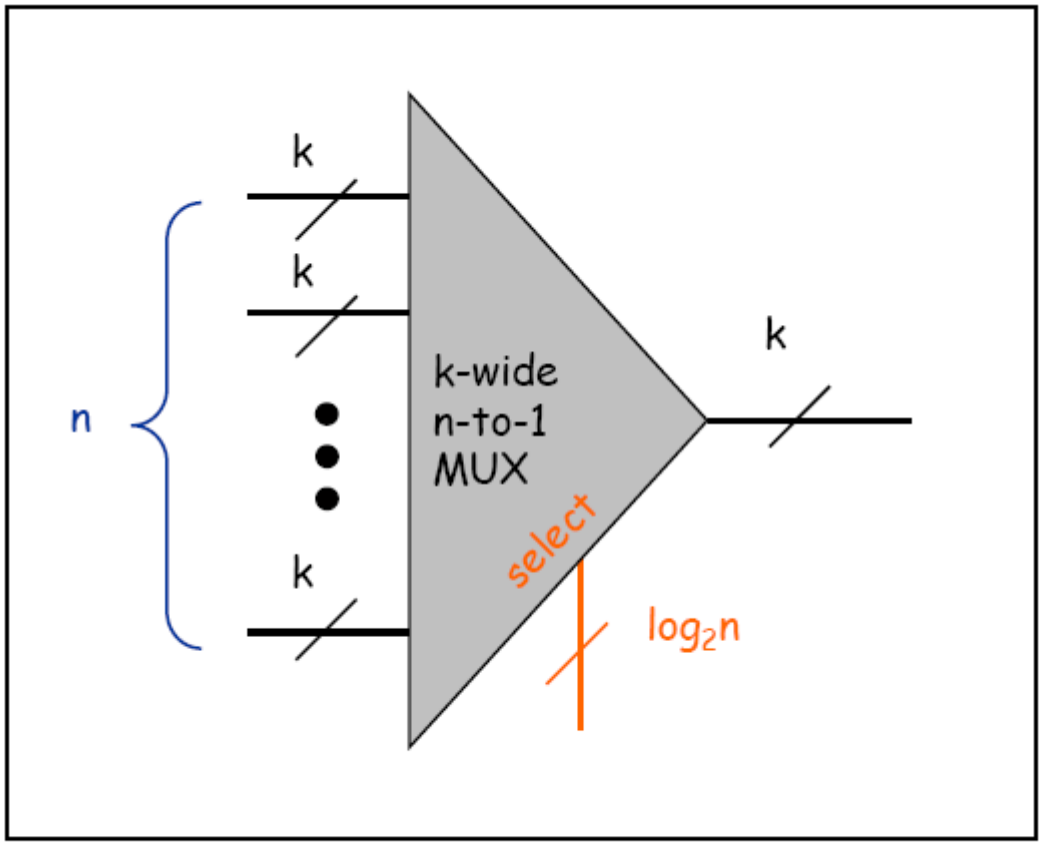


op	2	1	0
+, -	0	0	0
&	0	0	1
^	0	1	0
<<, >>	0	1	1
input 2	1	0	0

Recap: Multiplexer

Goal: select from one of n k-bit buses

- Implemented by layering k n-to-1 multiplexer

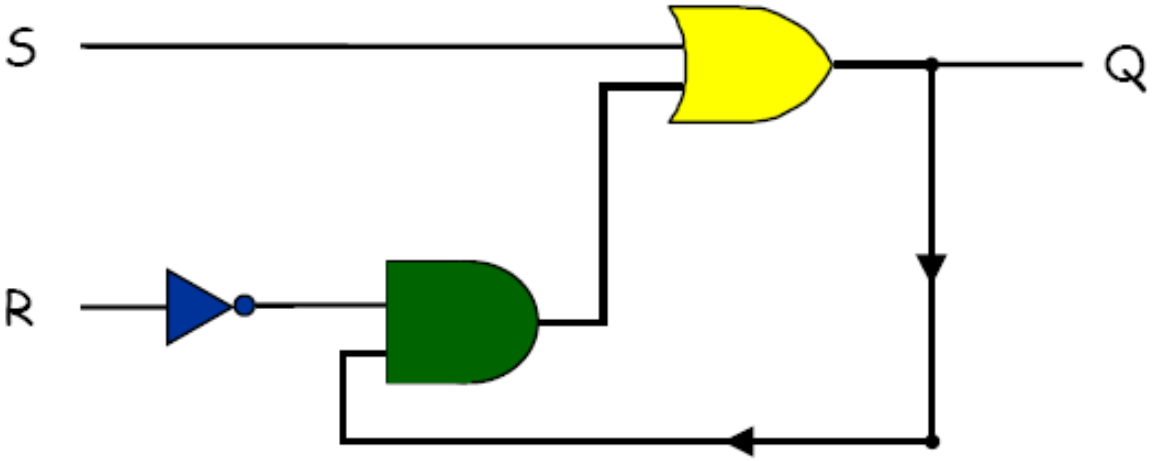


Interface

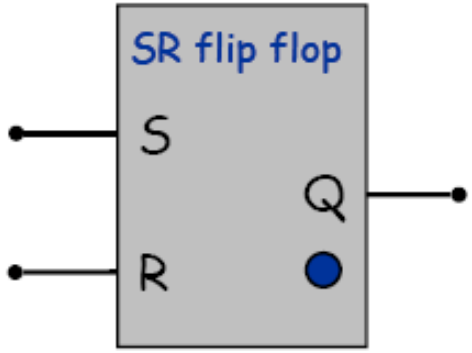
Recap: flip flop

SR Flip-Flop.

- S = 1, R = 0 (set) ⇒ Flips "bit" on.
- S = 0, R = 1 (reset) ⇒ Flops "bit" off.
- S = R = 0 ⇒ Status quo.
- S = R = 1 ⇒ Not allowed.



Implementation



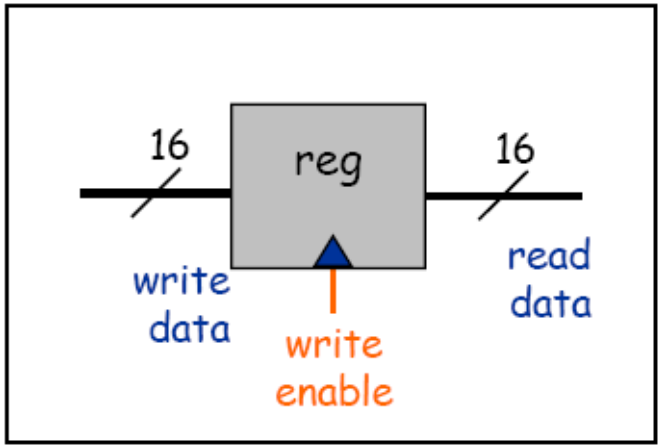
Interface

Stand-Alone Register

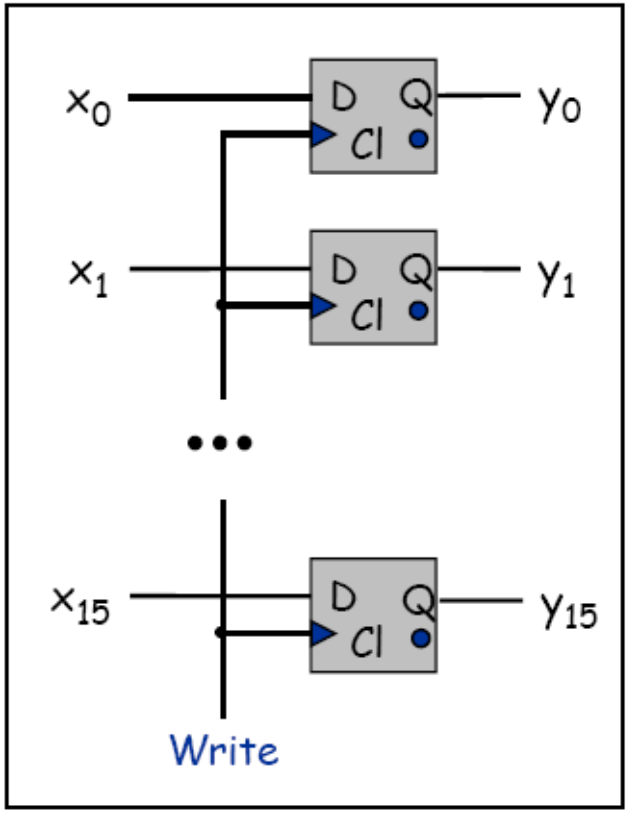
k-bit register.

- Stores k bits.
- Register contents always available on output.
- If write enable is asserted, k input bits get copied into register.

Ex: Program Counter, 16 TOY registers, 256 TOY memory locations.



16-bit Register Interface

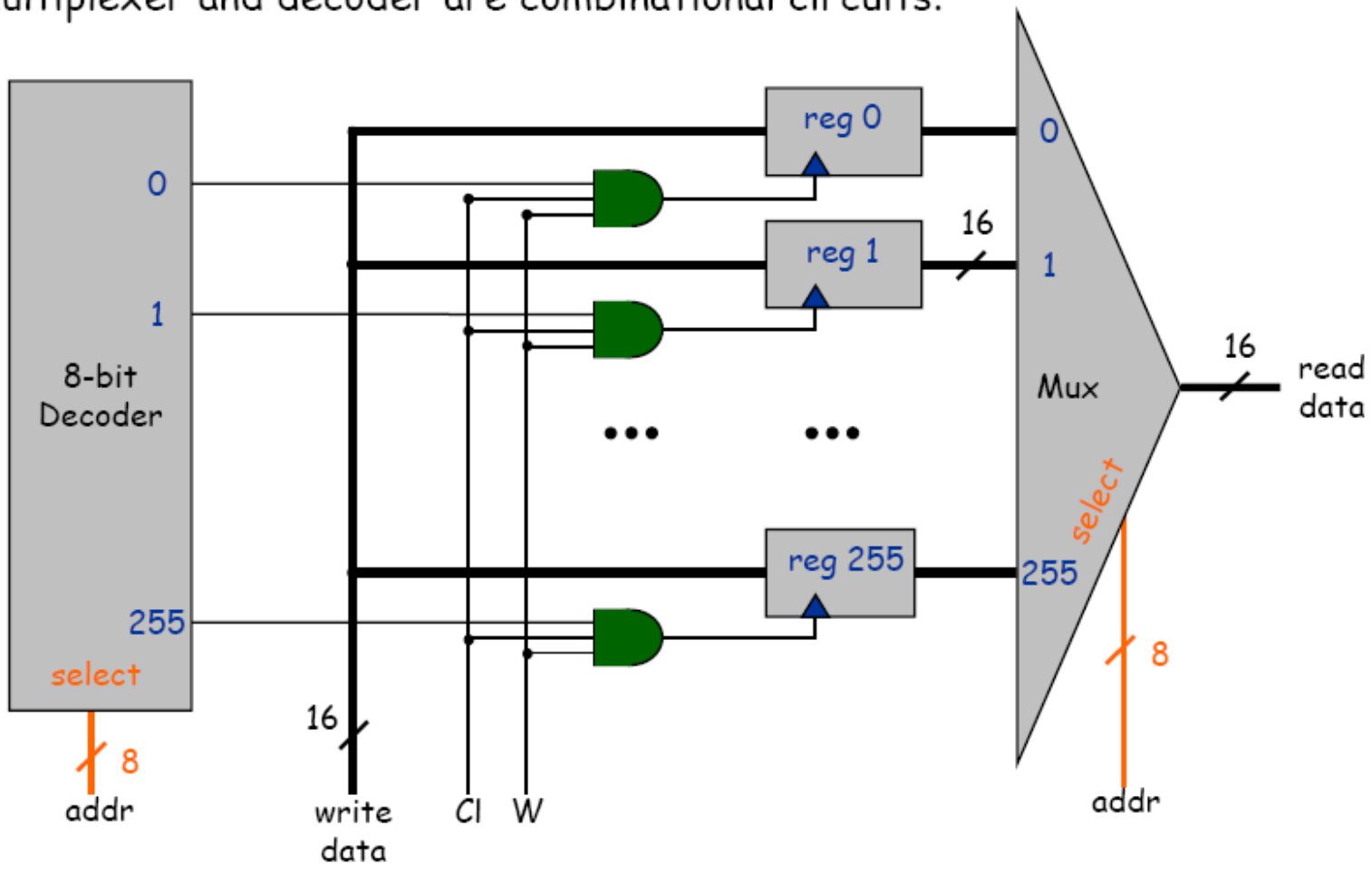


16-bit Register Implementation

Register file implementation

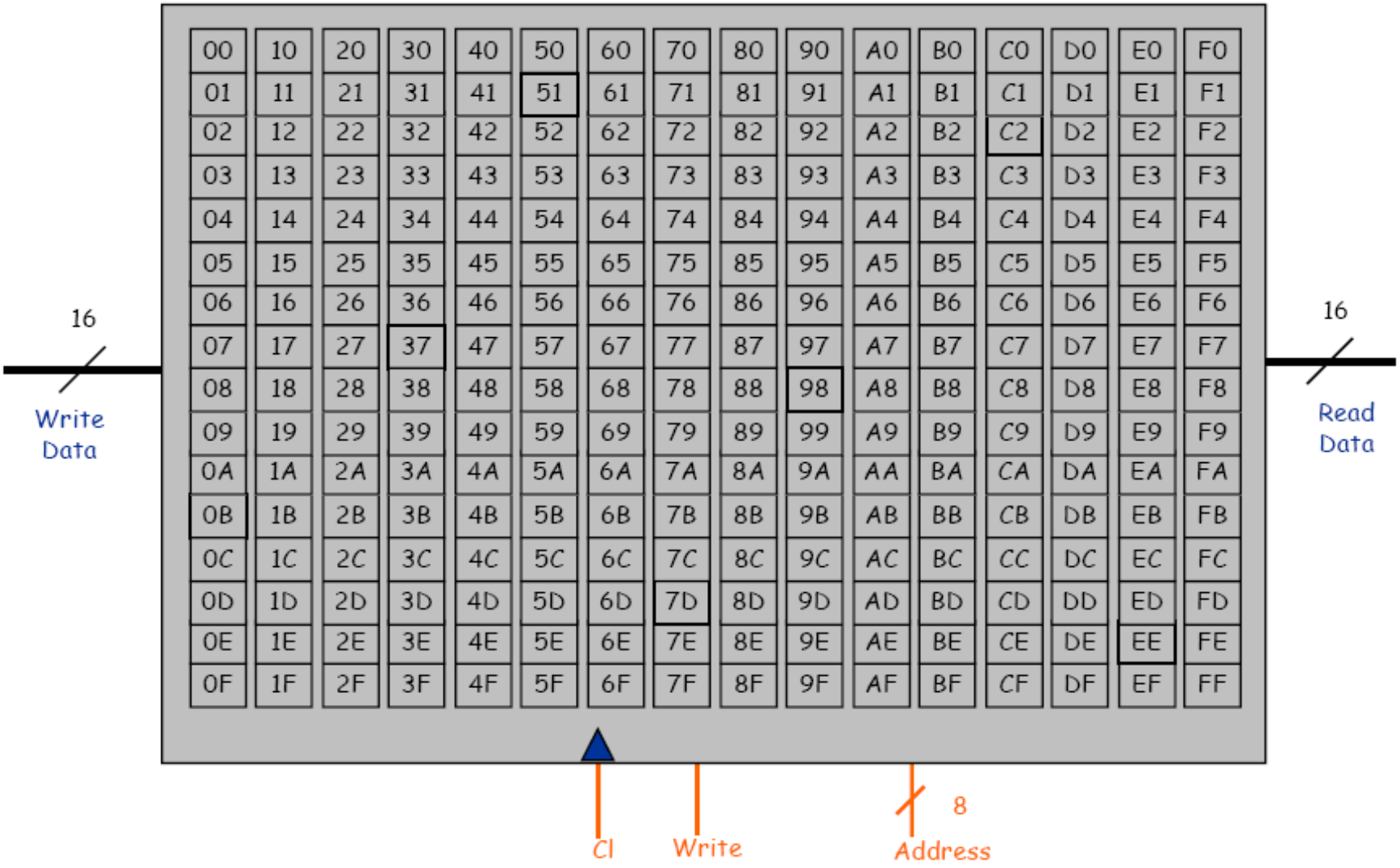
Implementation example: TOY main memory.

- Use 256 16-bit registers.
- Multiplexer and decoder are combinational circuits.



Recap: memory

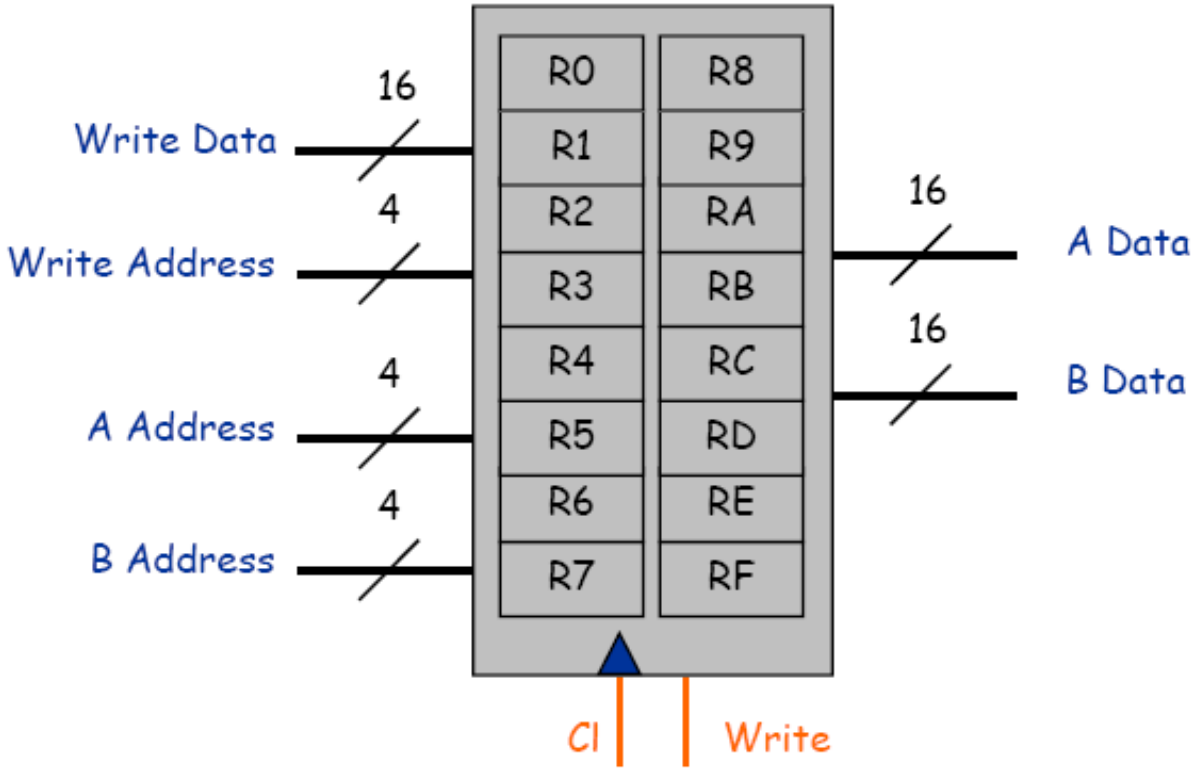
TOY main memory: 256 x 16-bit register file.



Recap: register file

TOY registers: fancy 16 x 16-bit register file.

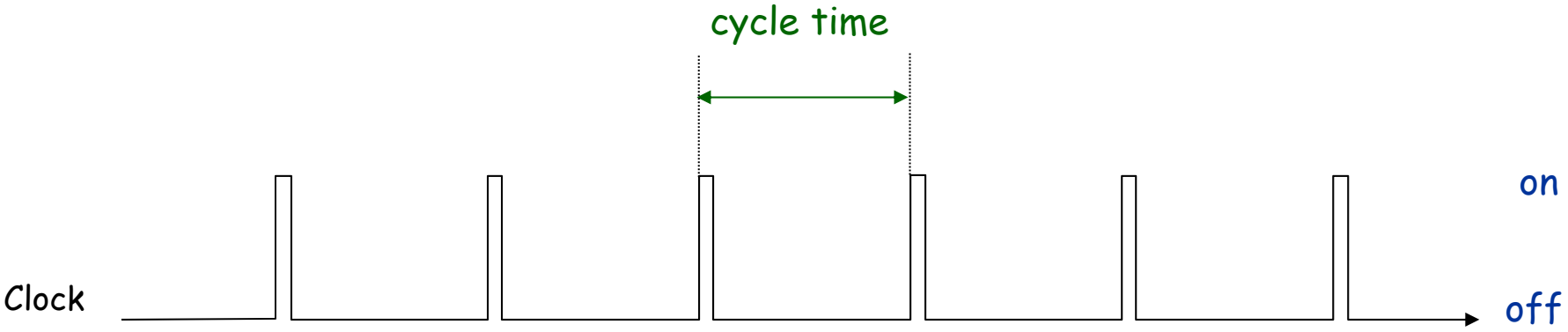
- Want to be able to read two registers, and write to a third in the same instructions: `R1 ← R2 + R3`.
- 3 address inputs, 2 data outputs.
- Add extra bank of muxes for a second read port.



Clock

Clock.

- Fundamental abstraction: regular on-off pulse.
  - on: fetch phase
  - off: execute phase
- External analog device.
- Synchronizes operations of different circuit elements.
- Requirement: clock cycle longer than max switching time.



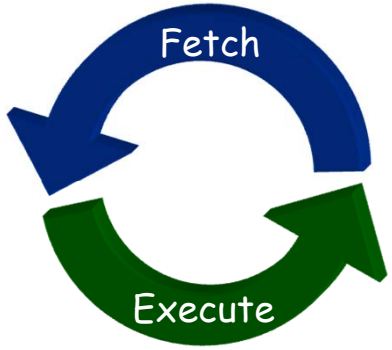
TOY Machine Architecture

The TOY Machine

Combinational circuits. ALU.
Sequential circuits. Memory.
Machine architecture. Wire components together to make computer.

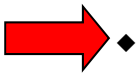
TOY machine.

- 256 16-bit words of memory.
- 16 16-bit registers.
- 1 8-bit program counter.
- 16 instruction types.



Design a processor

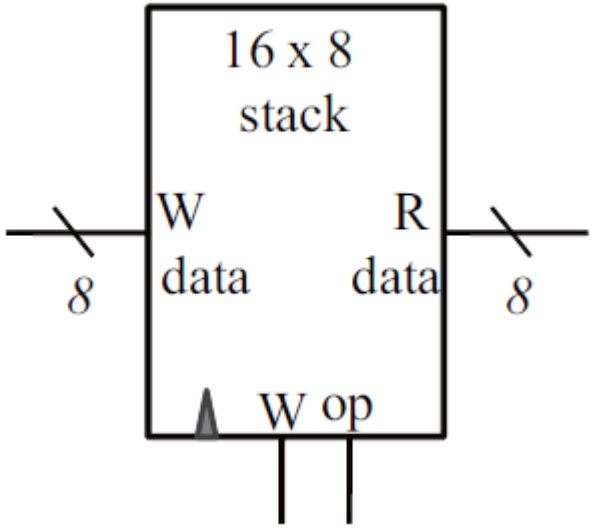
How to build a processor



- Develop instruction set architecture (ISA)
  - 16-bit words, 16 TOY machine instructions
- Determine major components
  - ALU, memory, registers, program counter
- Determine datapath requirements
  - Flow of bits
- Analyze how to implement each instruction
  - Determine settings of control signals

Practice: 4-bit counter

Practice: stack



W	op	operation	semantics
0	0	read	the content of stack[top] can be read from ”R data”
0	1	top	the content of top can be read from ”R data”
1	0	push	top++; write the ”W data” to stack[top];
1	1	pop	top−−;

Build a TOY: Interface

Instruction set architecture (ISA).

- 16-bit words, 256 words of memory, 16 registers.
- Determine set of primitive instructions.
  - too narrow ⇒ cumbersome to program
  - too broad ⇒ cumbersome to build hardware
- 16 instructions.

Instructions	
0:	halt
1:	add
2:	subtract
3:	and
4:	xor
5:	shift left
6:	shift right
7:	load address

Instructions	
8:	load
9:	store
A:	load indirect
B:	store indirect
C:	branch zero
D:	branch positive
E:	jump register
F:	jump and link

TOY Reference Card

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Format 1	opcode				dest d				source s				source t			
Format 2	opcode				dest d				addr							

#	Operation	Fmt	Pseudocode
0:	halt	1	exit(0)
1:	add	1	R[d] ← R[s] + R[t]
2:	subtract	1	R[d] ← R[s] - R[t]
3:	and	1	R[d] ← R[s] & R[t]
4:	xor	1	R[d] ← R[s] ^ R[t]
5:	shift left	1	R[d] ← R[s] << R[t]
6:	shift right	1	R[d] ← R[s] >> R[t]
7:	load addr	2	R[d] ← addr
8:	load	2	R[d] ← mem[addr]
9:	store	2	mem[addr] ← R[d]
A:	load indirect	1	R[d] ← mem[R[t]]
B:	store indirect	1	mem[R[t]] ← R[d]
C:	branch zero	2	if (R[d] == 0) pc ← addr
D:	branch positive	2	if (R[d] > 0) pc ← addr
E:	jump register	1	pc ← R[t]
F:	jump and link	2	R[d] ← pc; pc ← addr

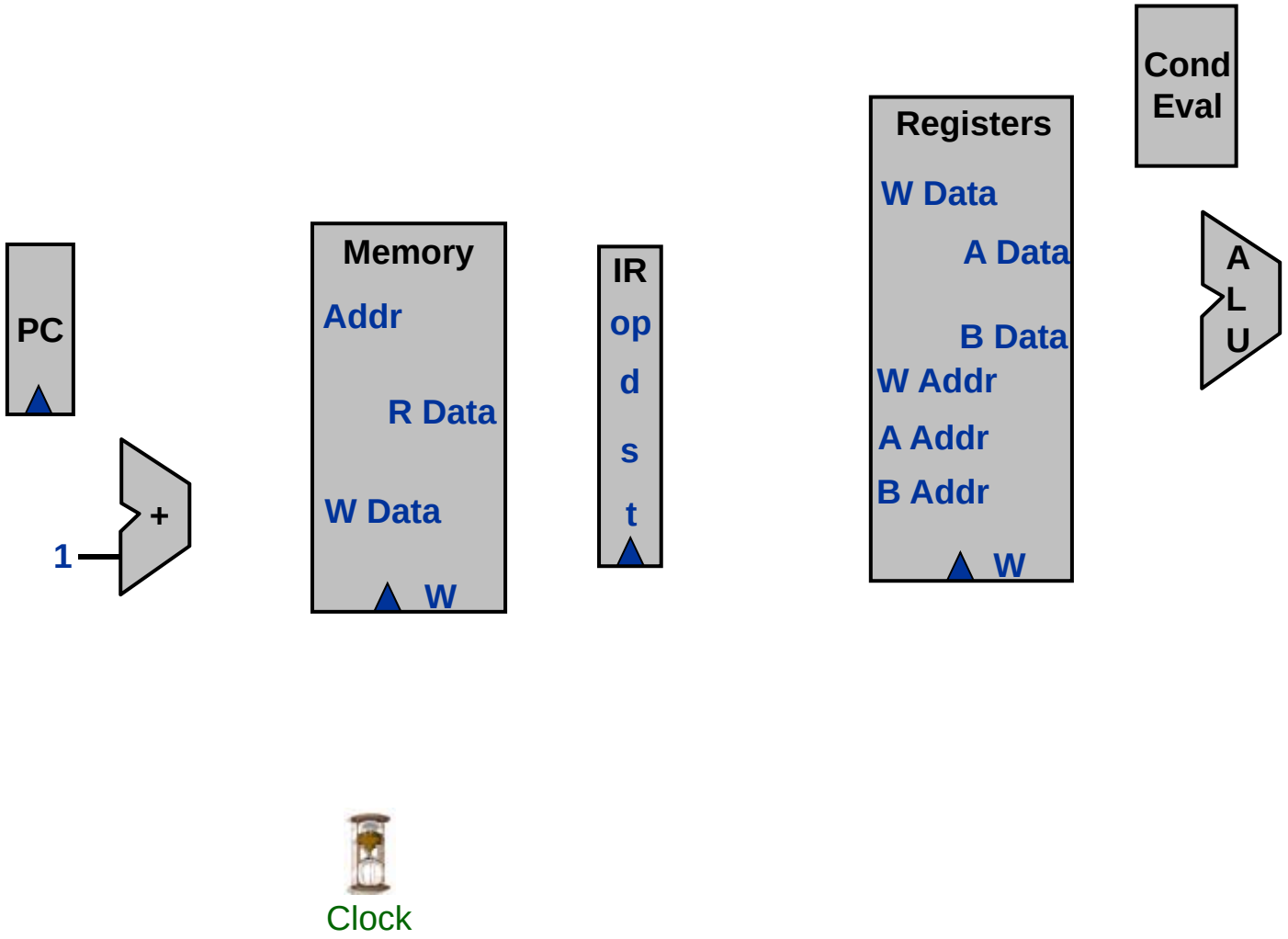
Register 0 always 0.
Loads from mem[FF] from stdin.
Stores to mem[FF] to stdout.

Design a processor

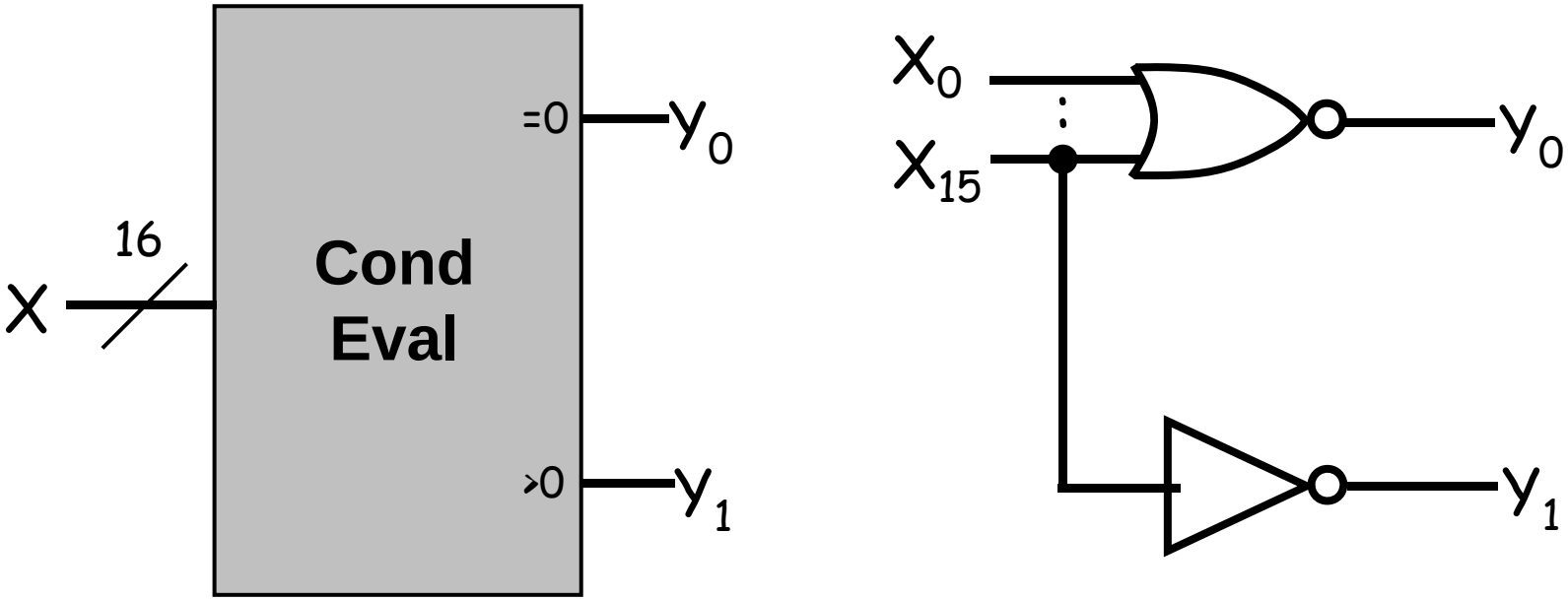
How to build a processor

- Develop instruction set architecture (ISA)
  - 16-bit words, 16 TOY machine instructions
- Determine major components
  - ALU, memory, registers, program counter
- Determine datapath requirements
  - Flow of bits
- Analyze how to implement each instruction
  - Determine settings of control signals

Components


PC
+
1
Memory
Addr
R Data
W Data
W
IR
op
d
s
t
Registers
W Data
A Data
B Data
W Addr
A Addr
B Addr
W
Cond
Eval
A
L
U
Clock

Cond. Eval.

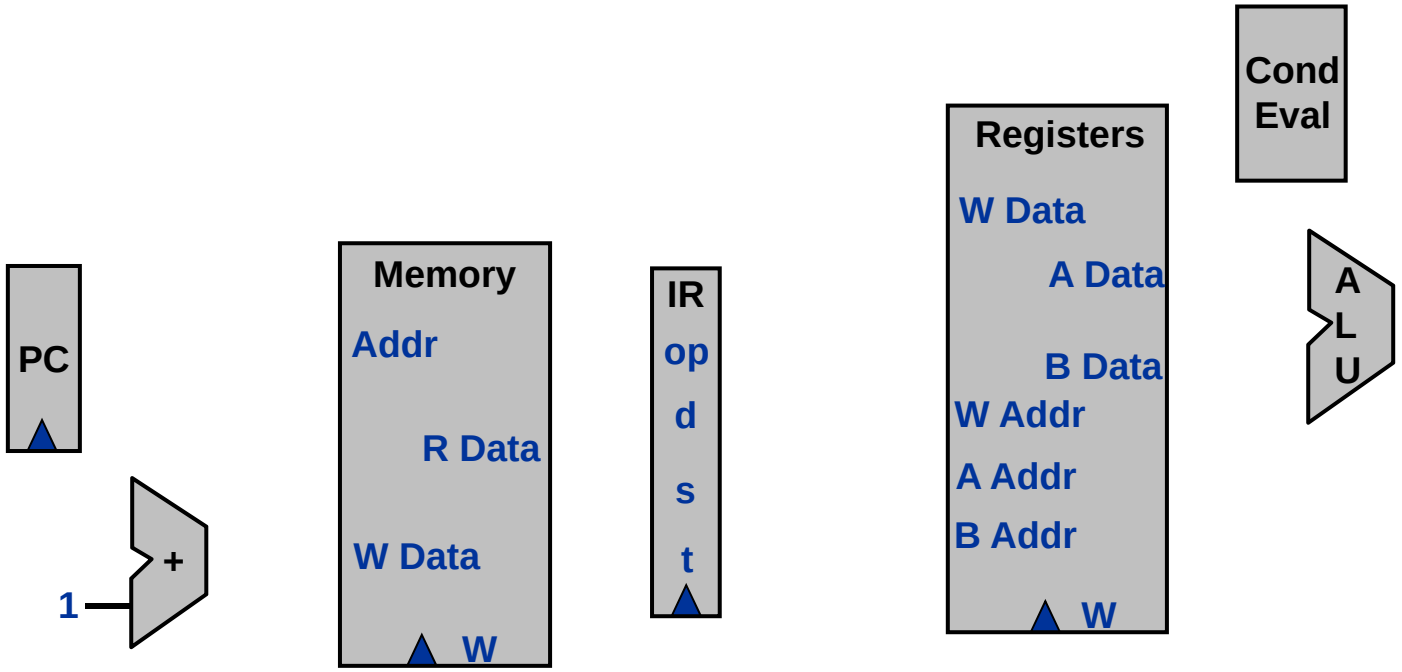


Design a processor

How to build a processor

- Develop instruction set architecture (ISA)
  - 16-bit words, 16 TOY machine instructions
- Determine major components
  - ALU, memory, registers, program counter
- Determine datapath requirements
  - Flow of bits
- Analyze how to implement each instruction
  - Determine settings of control signals

Datapath



1-6 `R[d] ← R[s] ALU R[t]`

9 `mem[addr] ← R[d]`

CD `if (R[d]?) pc ← addr`

7 `R[d] ← addr`

A `R[d] ← mem[R[t]]`

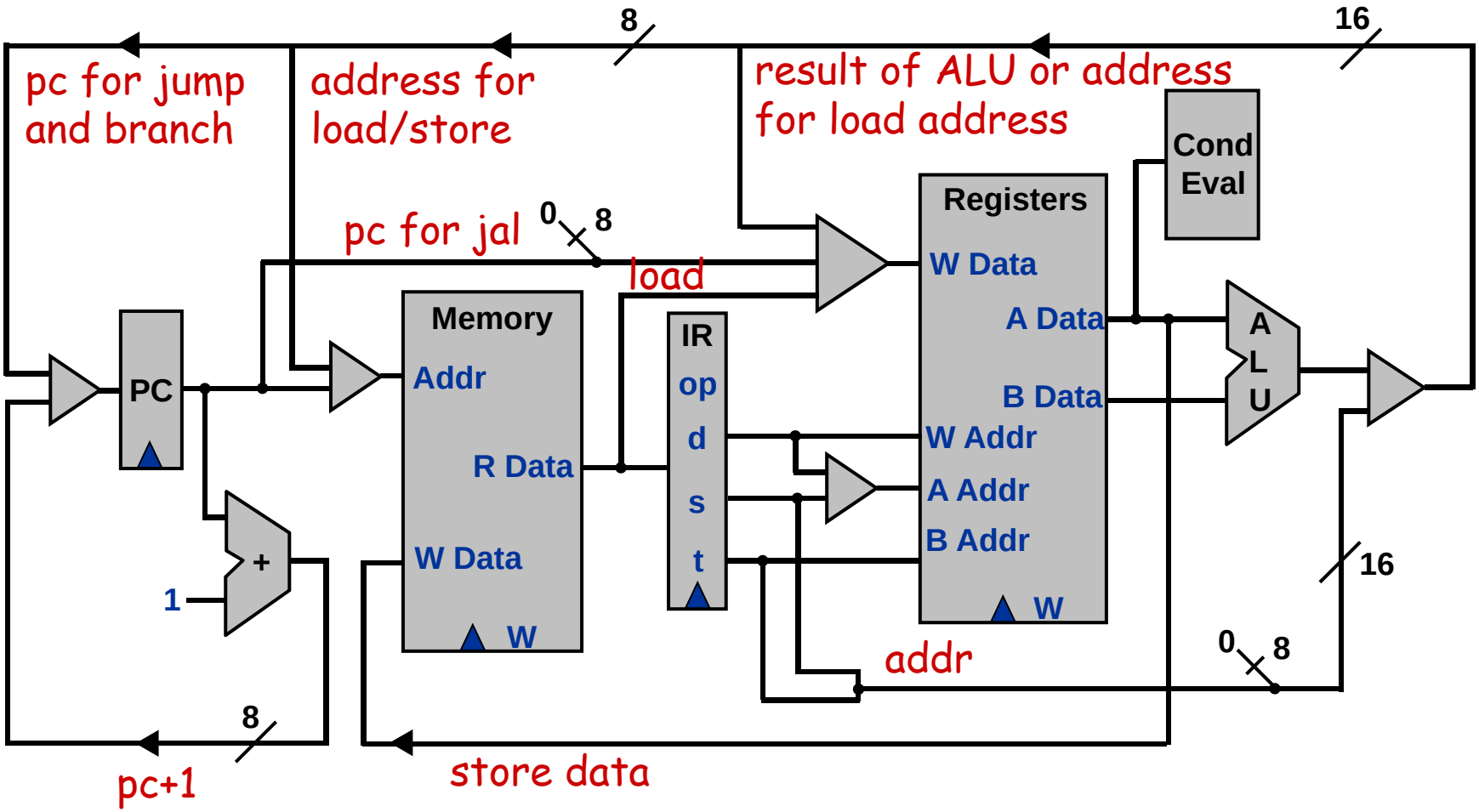
E `pc ← R[t]`

8 `R[d] ← mem[addr]`

B `mem[R[t]] ← R[d]`

F `R[d] ← pc; pc ← addr`

Datapath

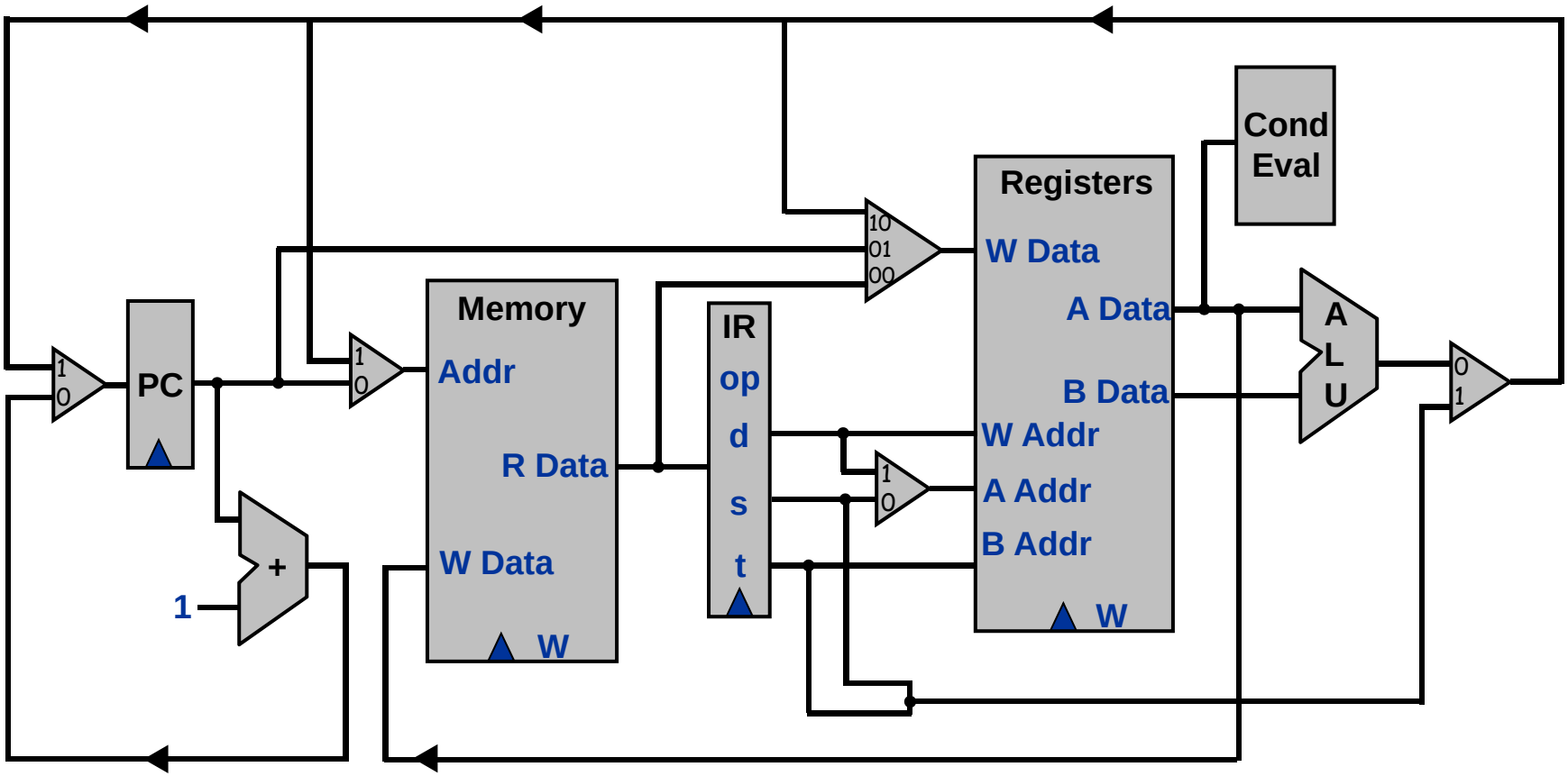

8
16
pc for jump
and branch
address for
load/store
result of ALU or address
for load address
Cond
Eval
Registers
W Data
A Data
B Data
W Addr
A Addr
B Addr
W
pc for jal
0
8
load
Memory
Addr
R Data
W Data
W
IR
op
d
s
t
PC
A
L
U
+
1
16
0
8
addr
8
pc+1
store data

Design a processor

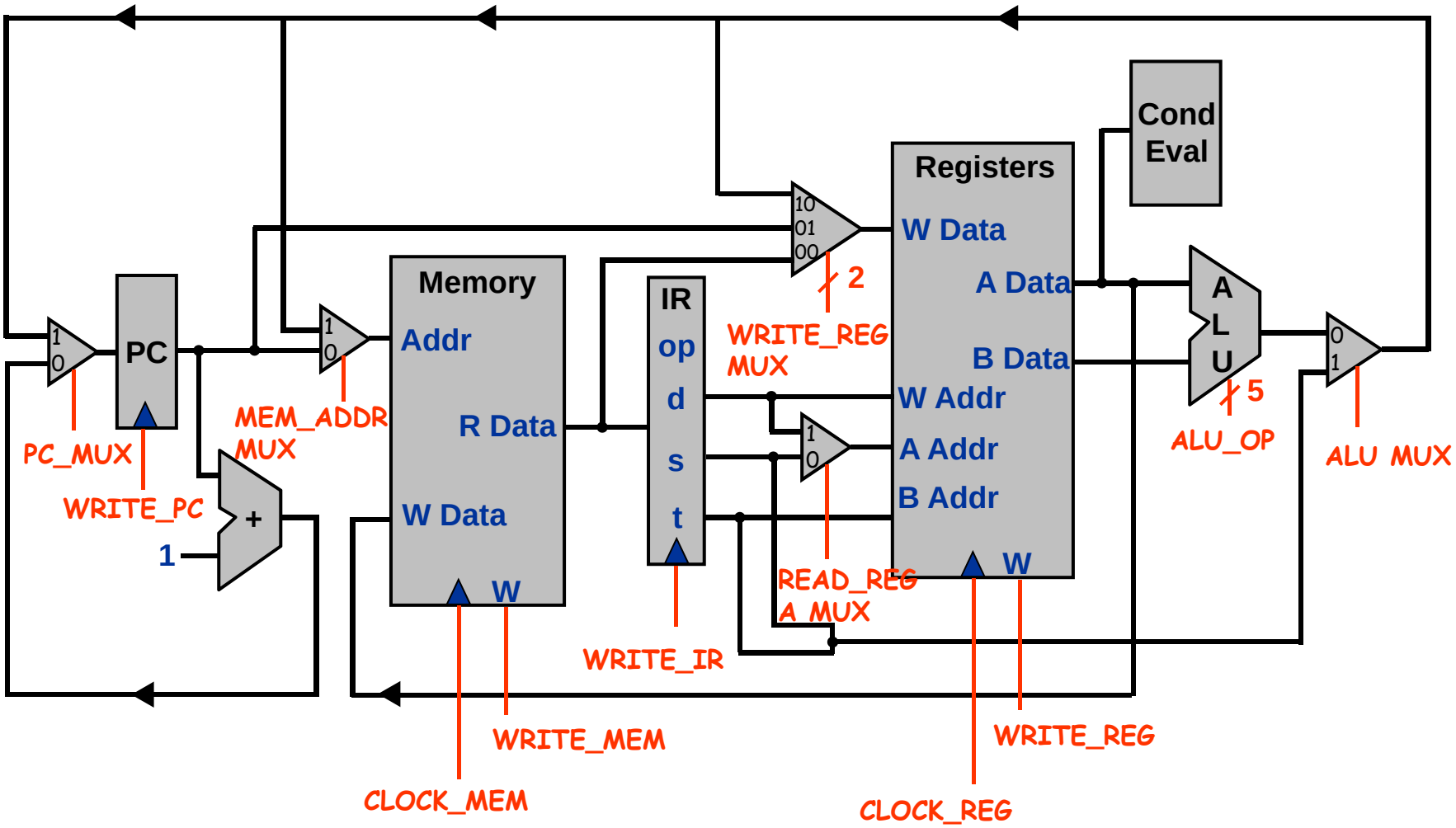
How to build a processor

- Develop instruction set architecture (ISA)
  - 16-bit words, 16 TOY machine instructions
- Determine major components
  - ALU, memory, registers, program counter
- Determine datapath requirements
  - Flow of bits
- Analyze how to implement each instruction
  - Determine settings of control signals

Datapath

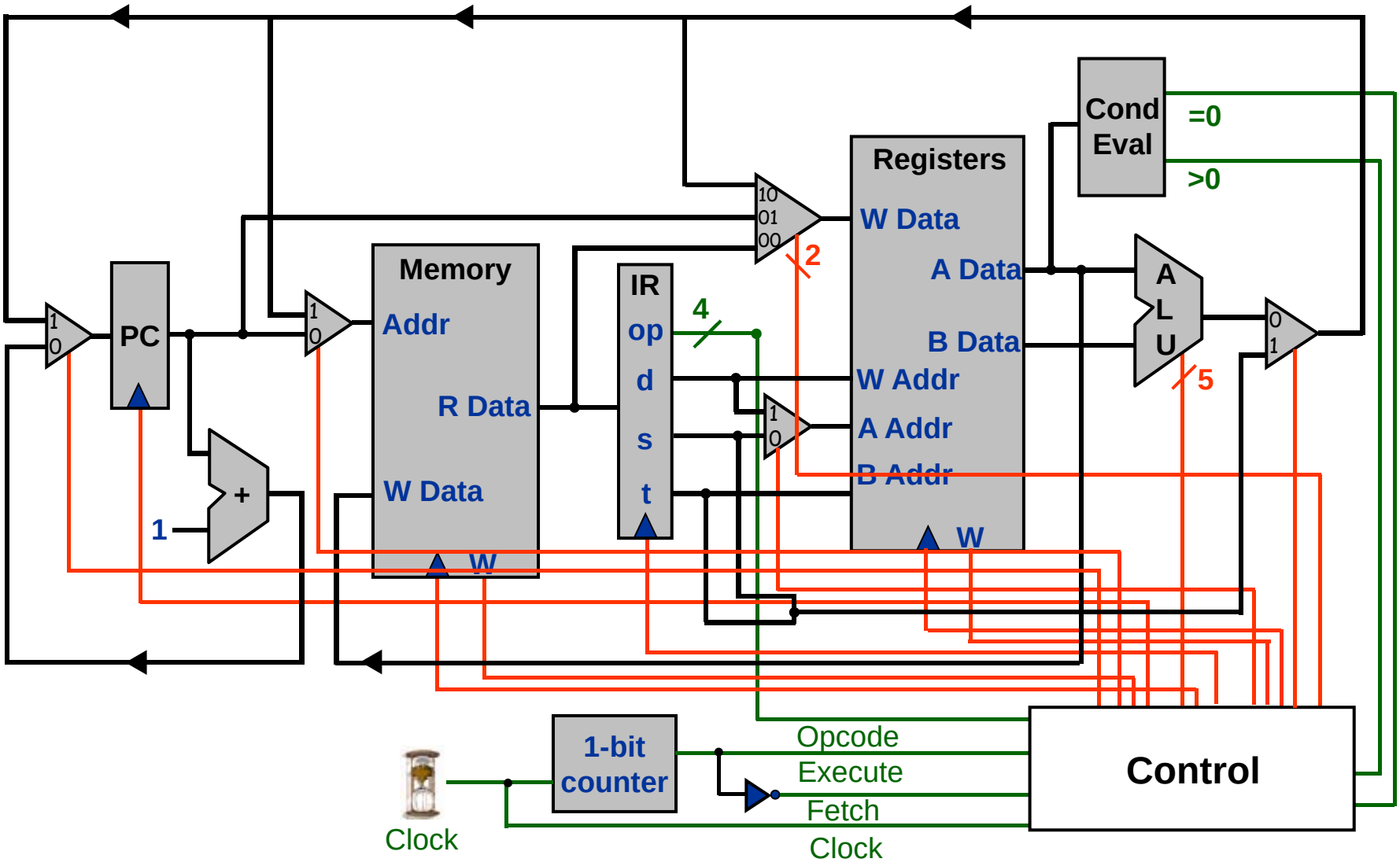

Cond
Eval
Registers
W Data
A Data
B Data
W Addr
A Addr
B Addr
W
10
01
00
Memory
Addr
R Data
W Data
W
IR
op
d
s
t
PC
1
0
+
1
A
L
U
0
1

Control

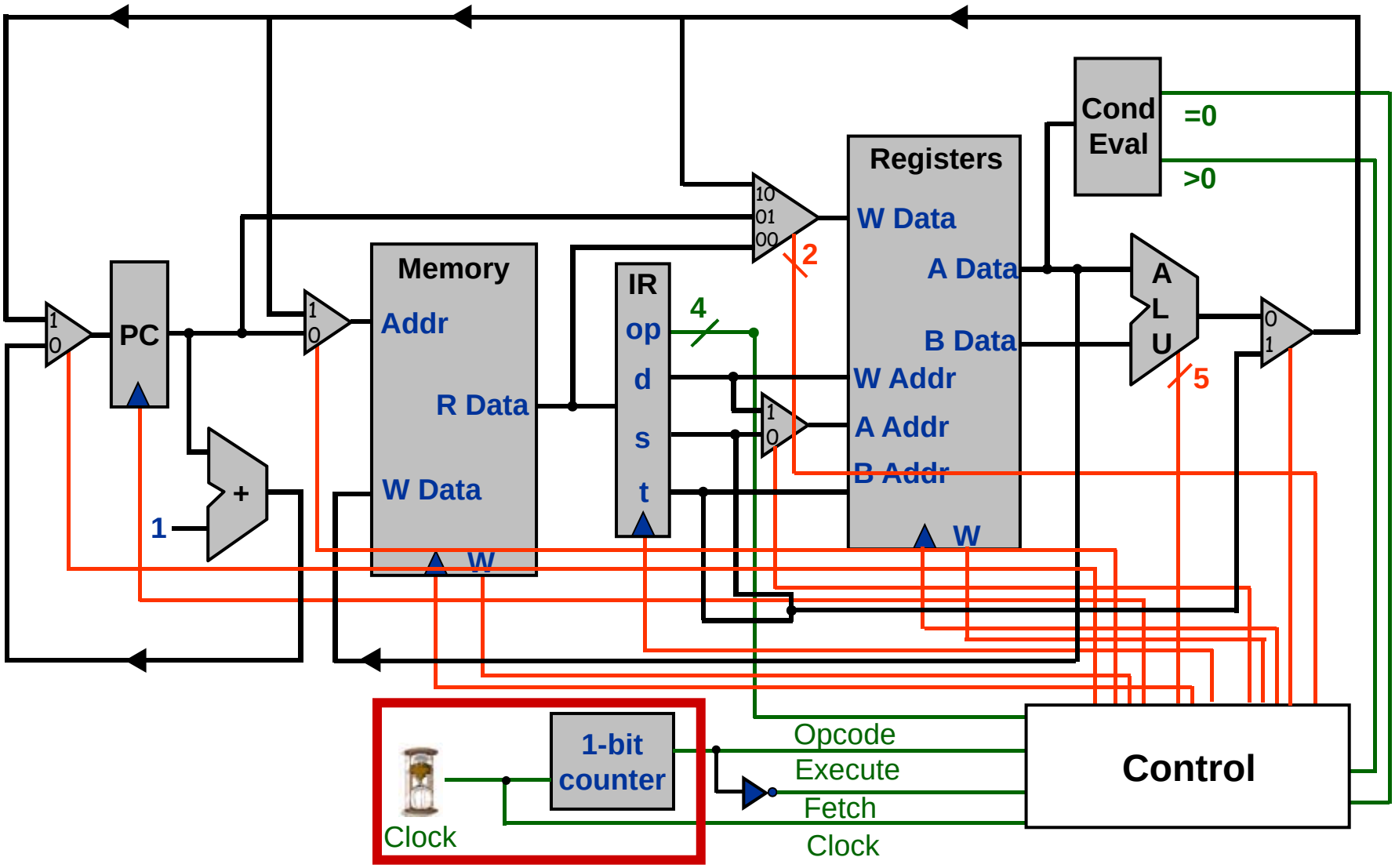


A total of 17 control signals

TOY architecture

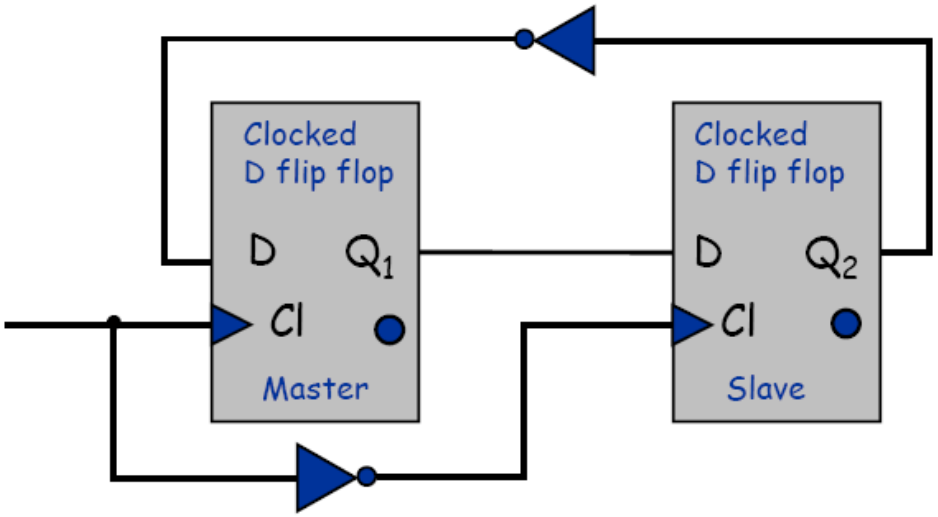


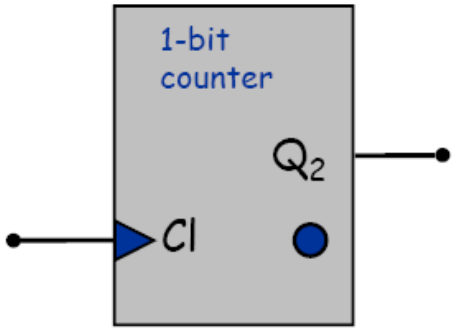
Clock


Cond Eval
=0
>0
Registers
W Data
A Data
B Data
W Addr
A Addr
B Addr
W
10
01
00
2
Memory
Addr
R Data
W Data
W
IR
op
d
s
t
4
A
L
U
5
PC
1
0
+
1
1-bit counter
Clock
Opcode
Execute
Fetch
Clock
Control

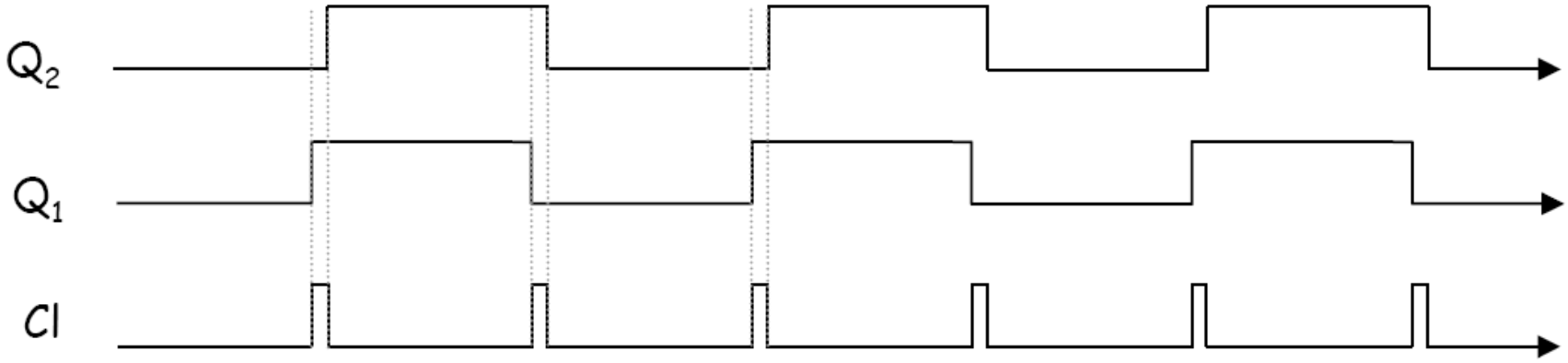
1-bit counter

- Circuit that oscillates between 1 and 0.





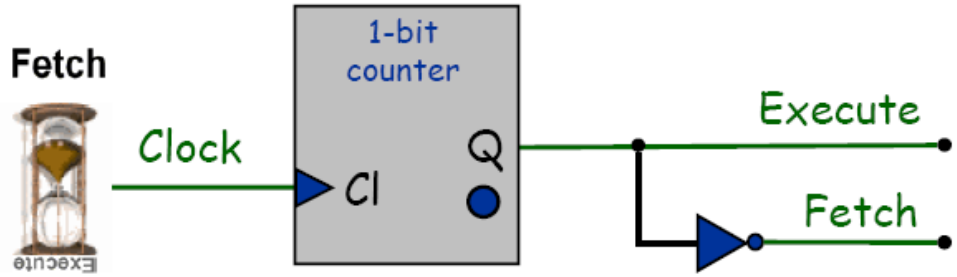
Interface

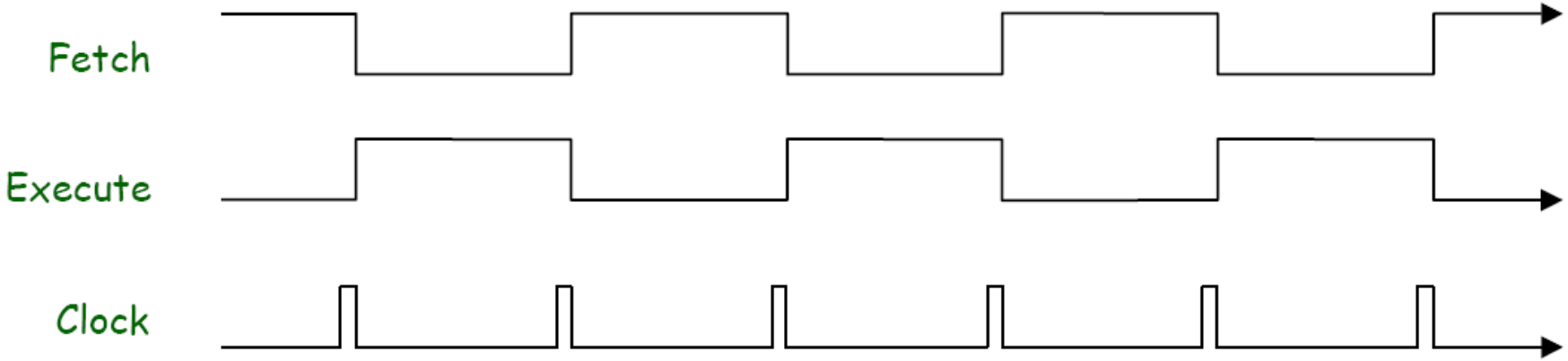


Clock

Two cycle design (fetch and execute)

- Use 1-bit counter to distinguish between 2 cycles.
- Use two cycles since fetch and execute phases each access memory and alter program counter.

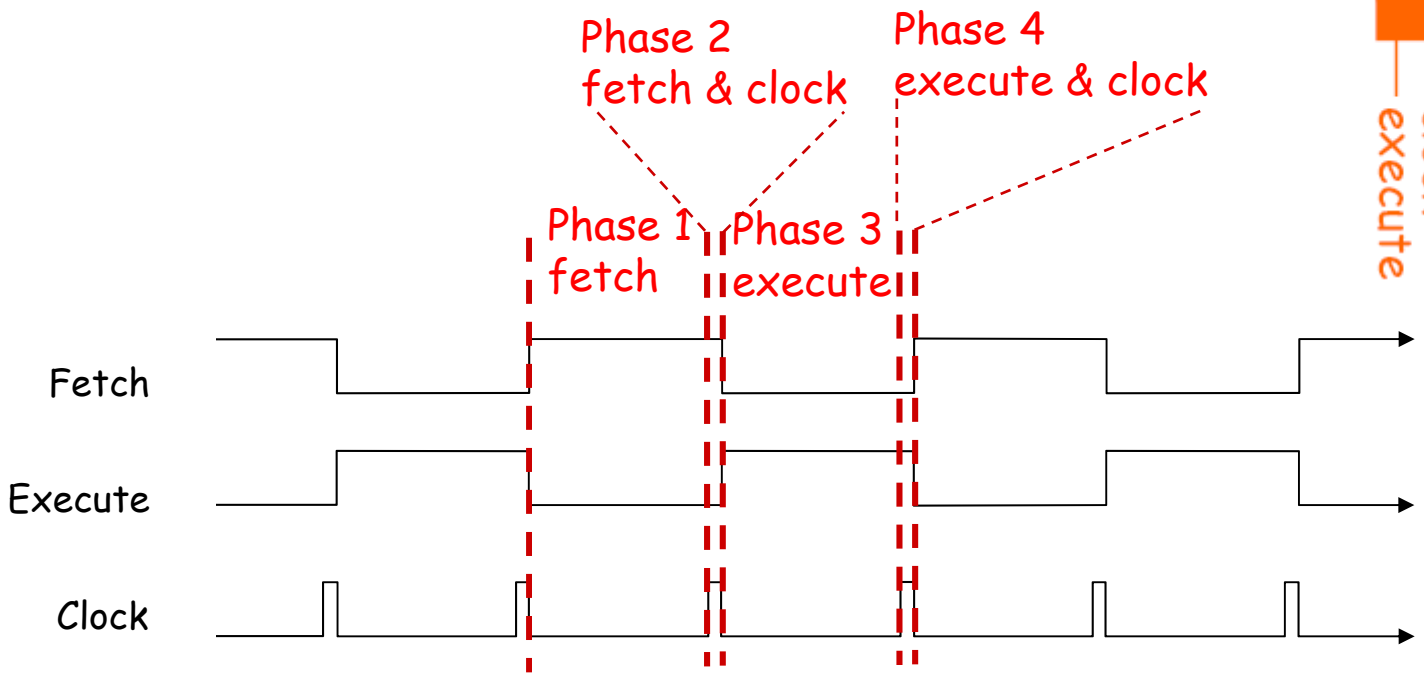




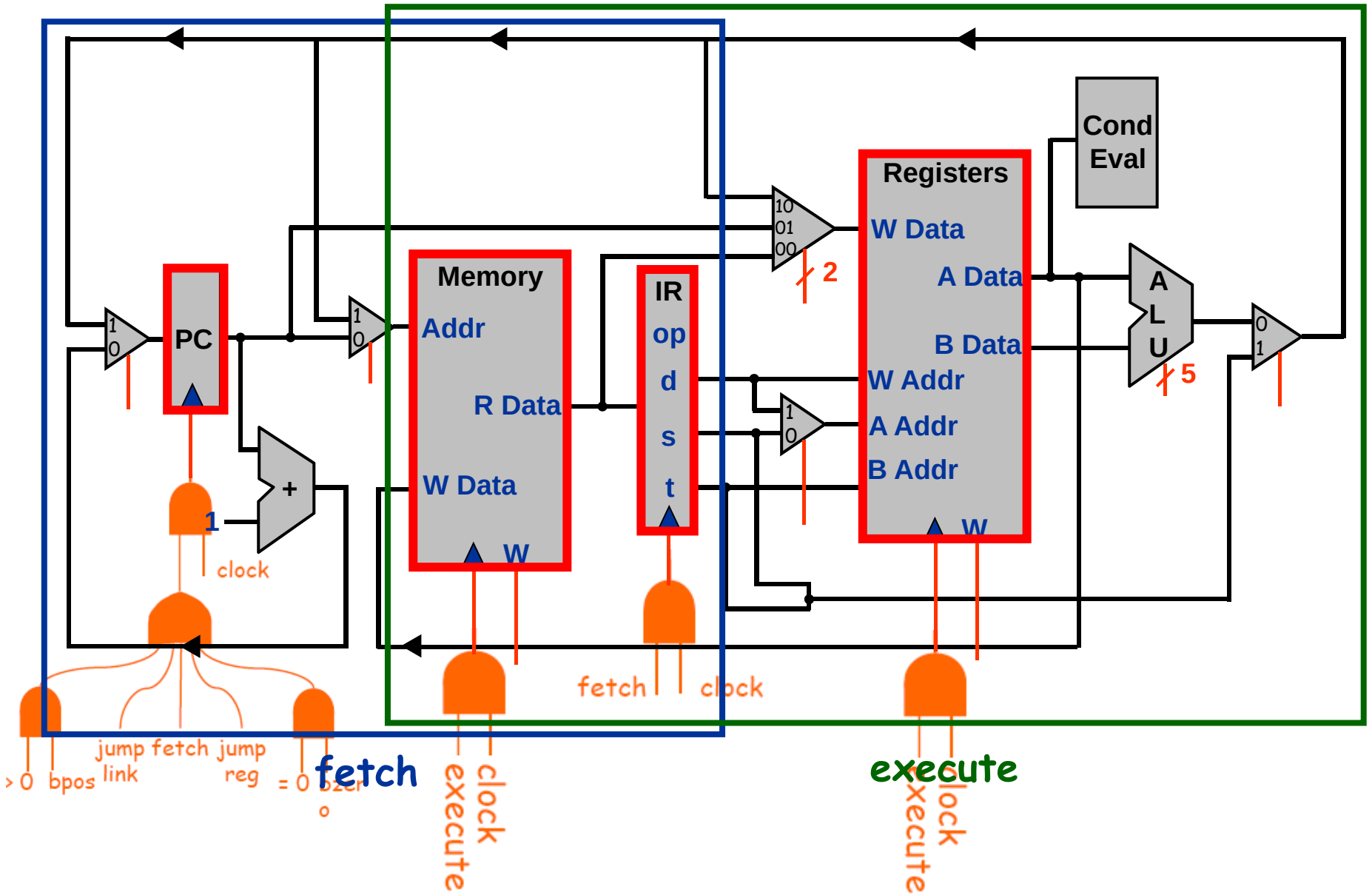
Clocking Methodology

Two-cycle design.

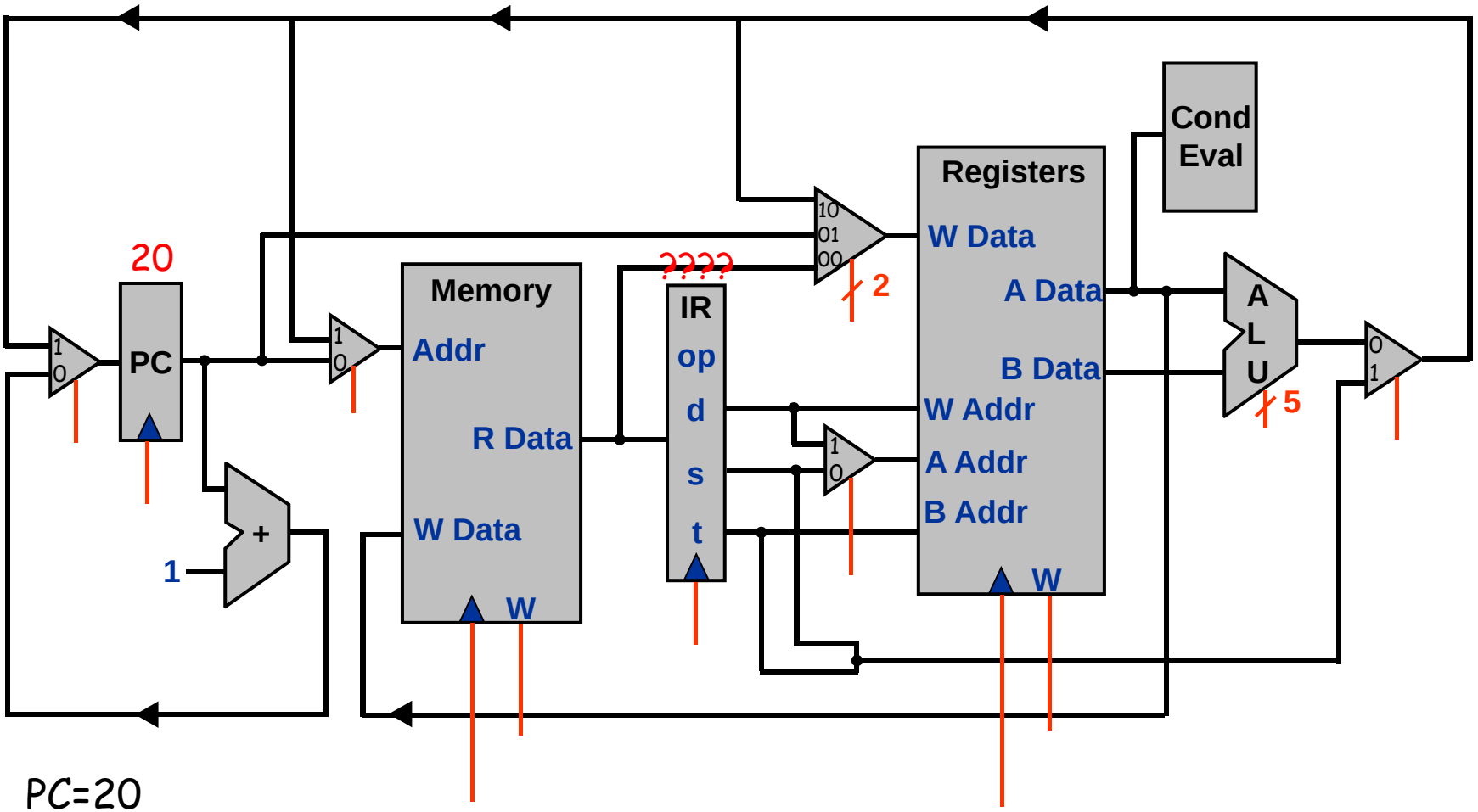
- Each control signal is in one of four epochs.
  - fetch [set memory address from pc]
  - fetch and clock [write instruction to IR]
  - execute [set ALU inputs from registers]
  - execute and clock [write result of ALU to registers]



Clocking Methodology

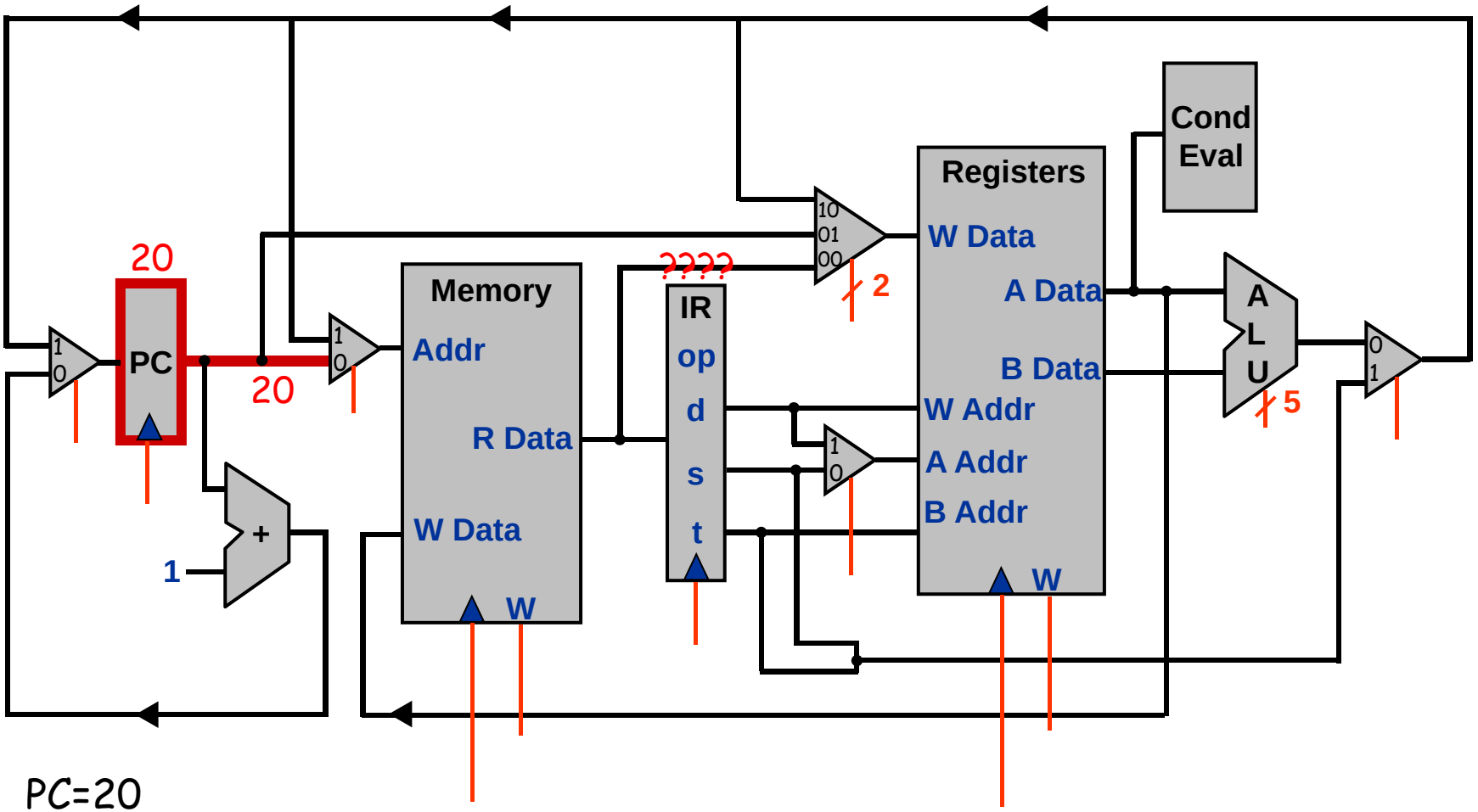


Example: ADD



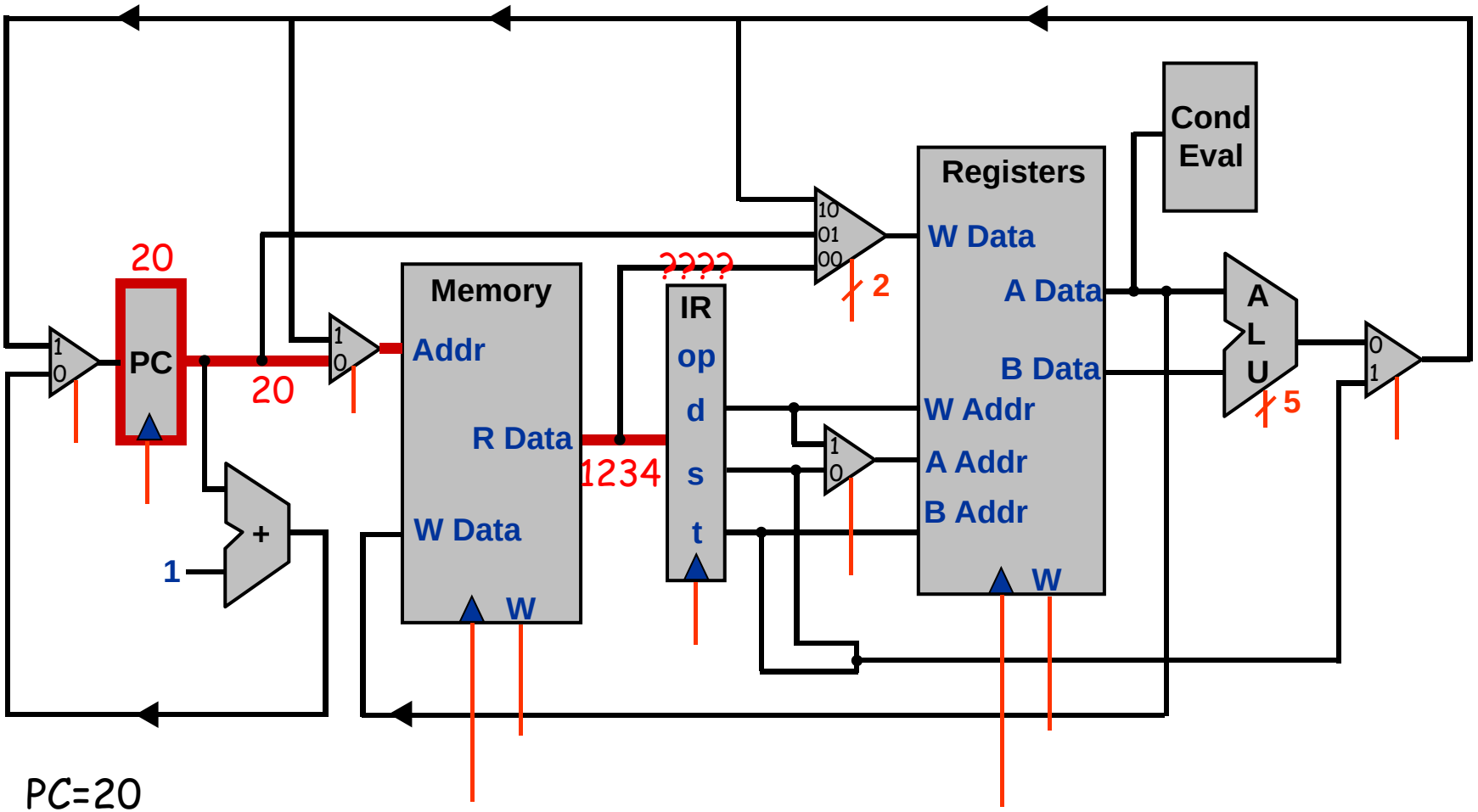
PC=20
Mem[20]=1234
R[3]=0028 R[4]=0064

Example: ADD (fetch)



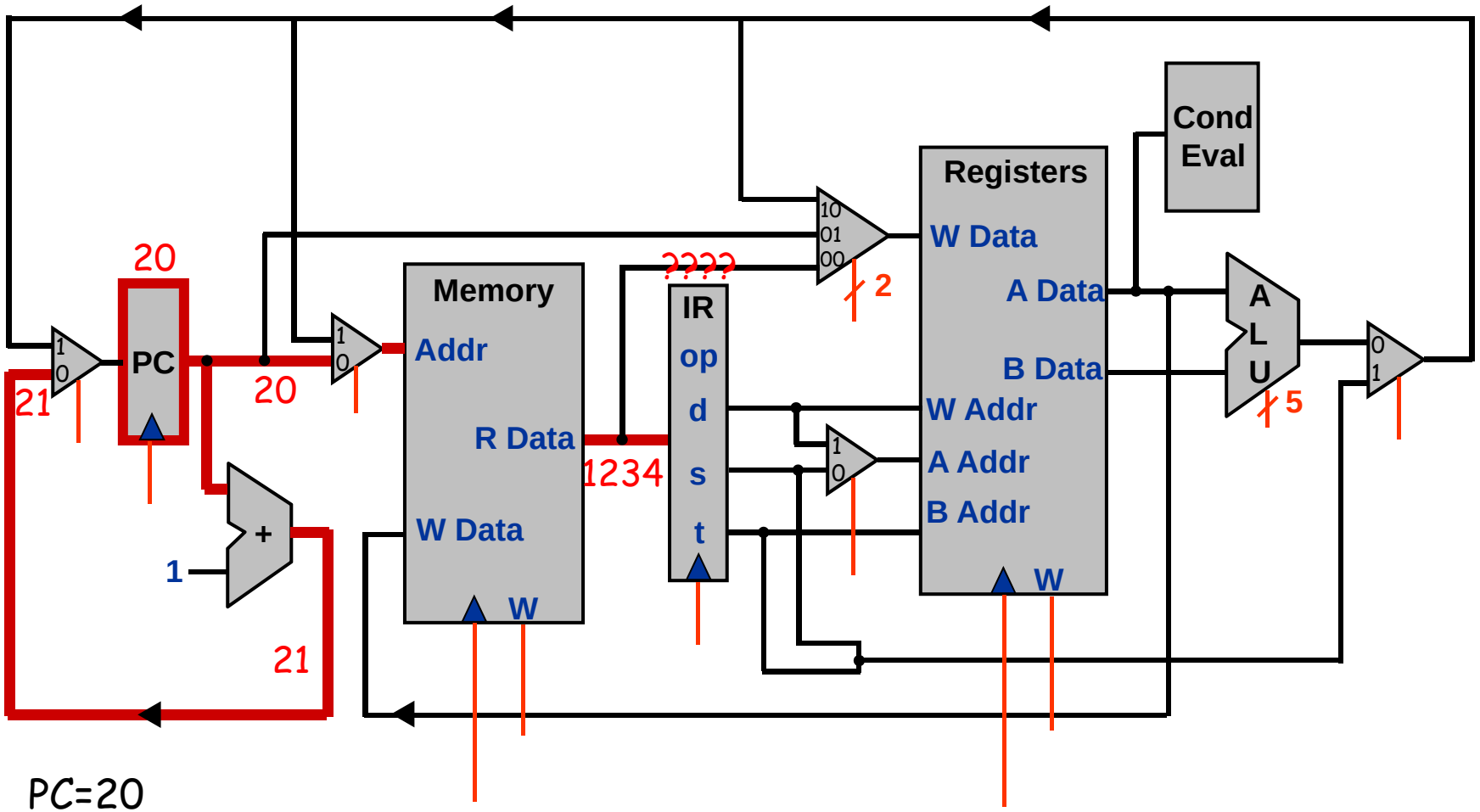
PC=20
Mem[20]=1234
R[3]=0028 R[4]=0064

Example: ADD (fetch)

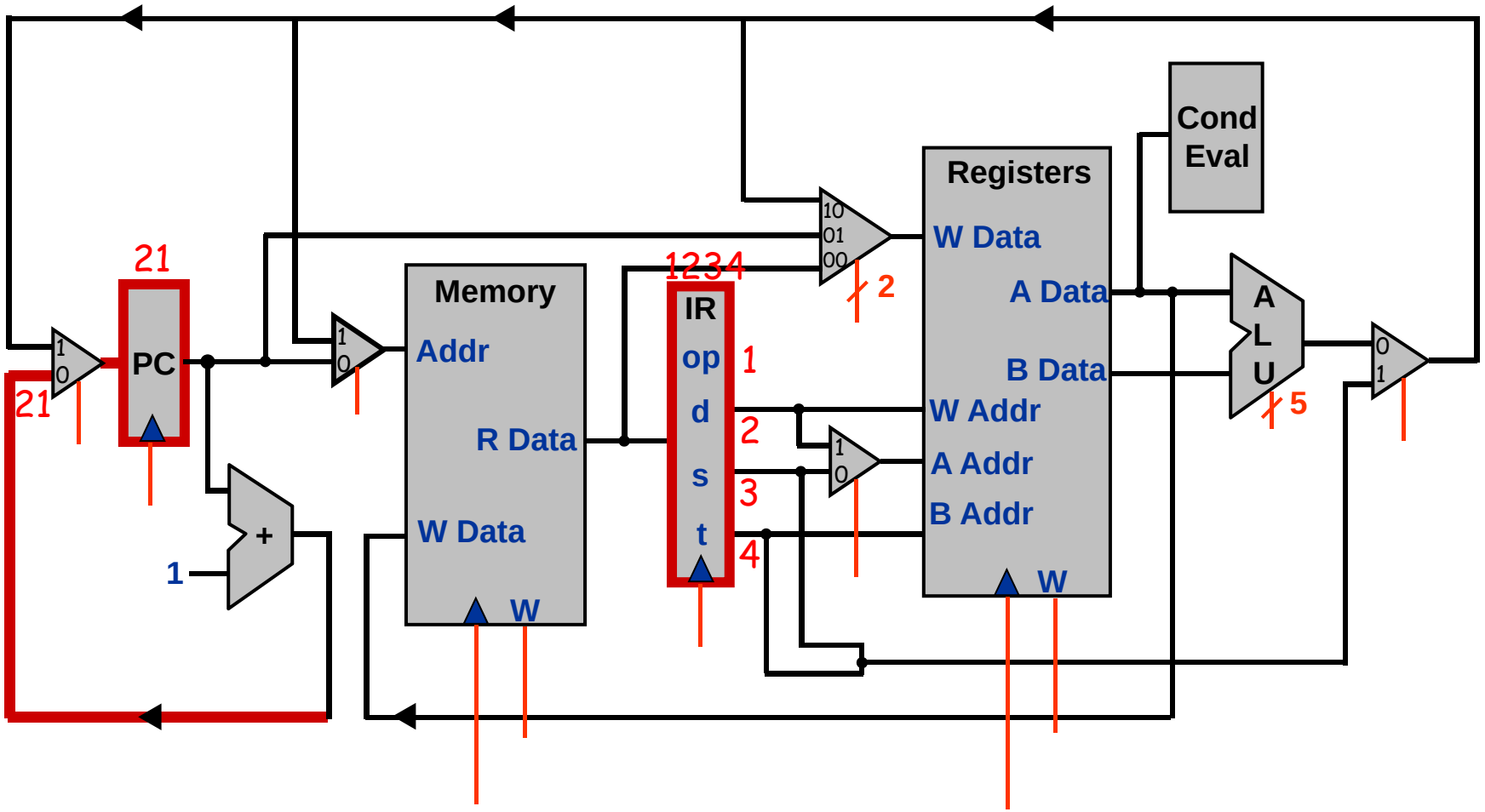


PC=20
Mem[20]=1234
R[3]=0028 R[4]=0064

Example: ADD (fetch)



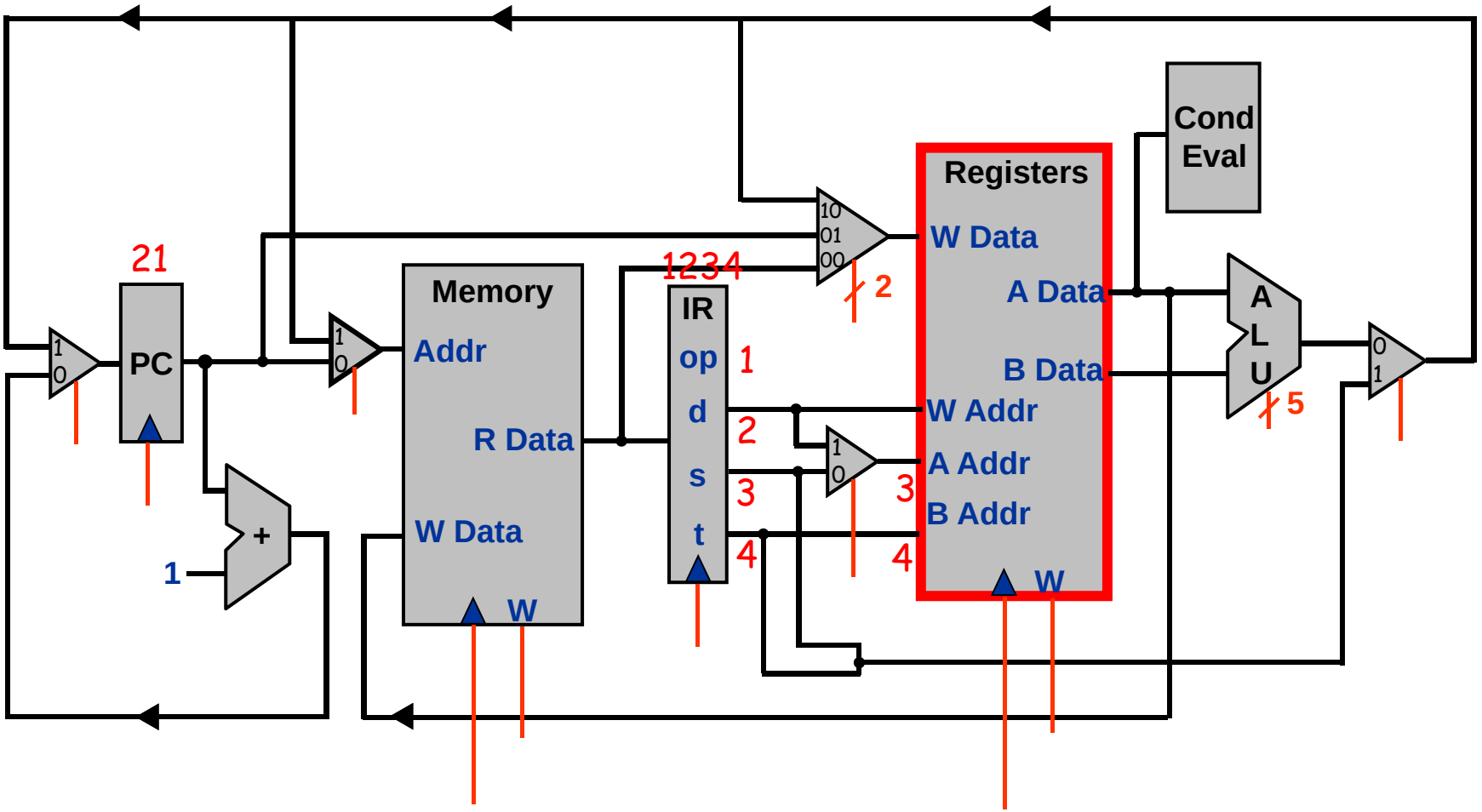
Example: ADD (fetch and clock)



Mem[20]=1234
R[3]=0028 R[4]=0064

PC=21
IR=1234

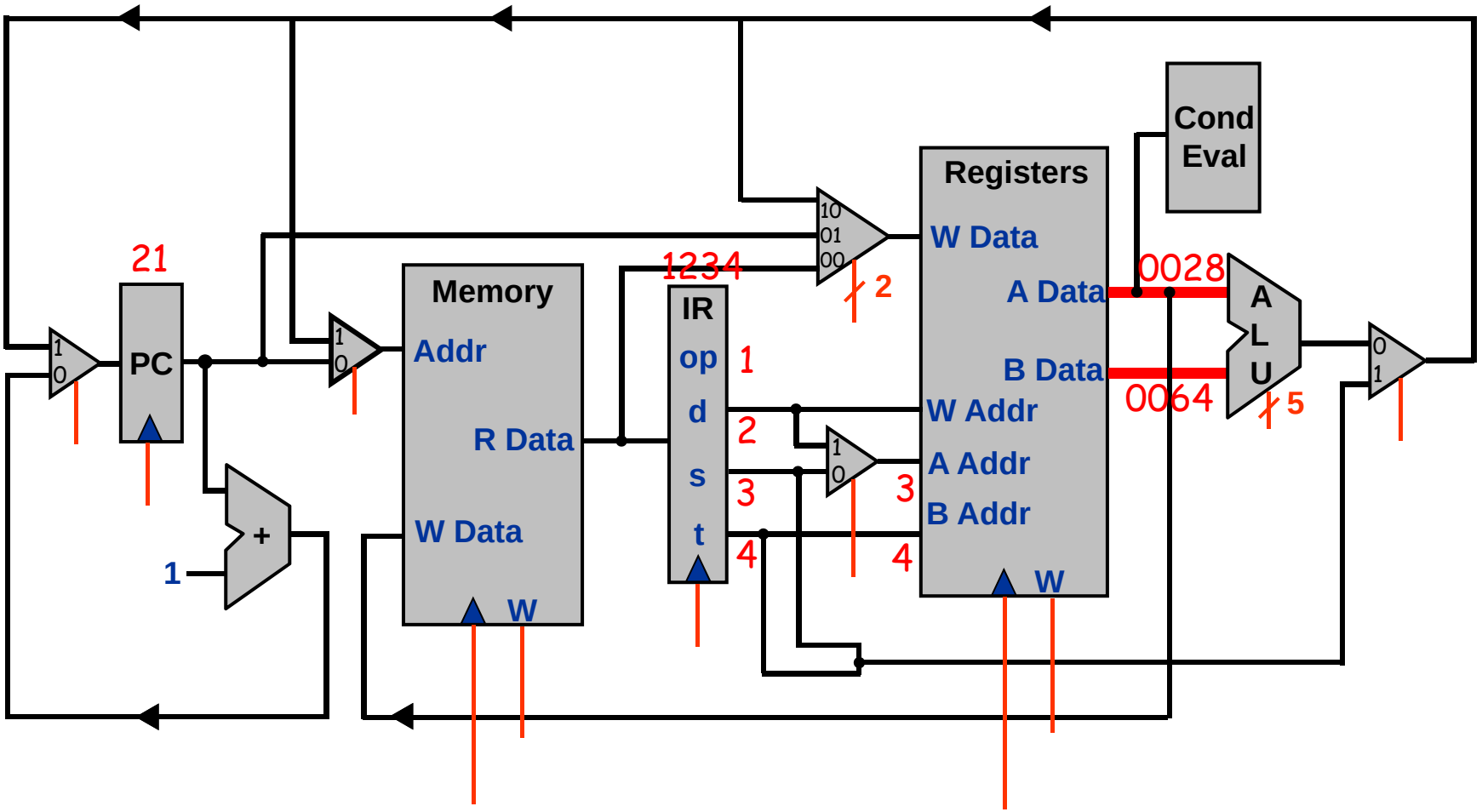
Example: ADD (execute)



Mem[20]=1234
R[3]=0028 R[4]=0064

PC=21
IR=1234

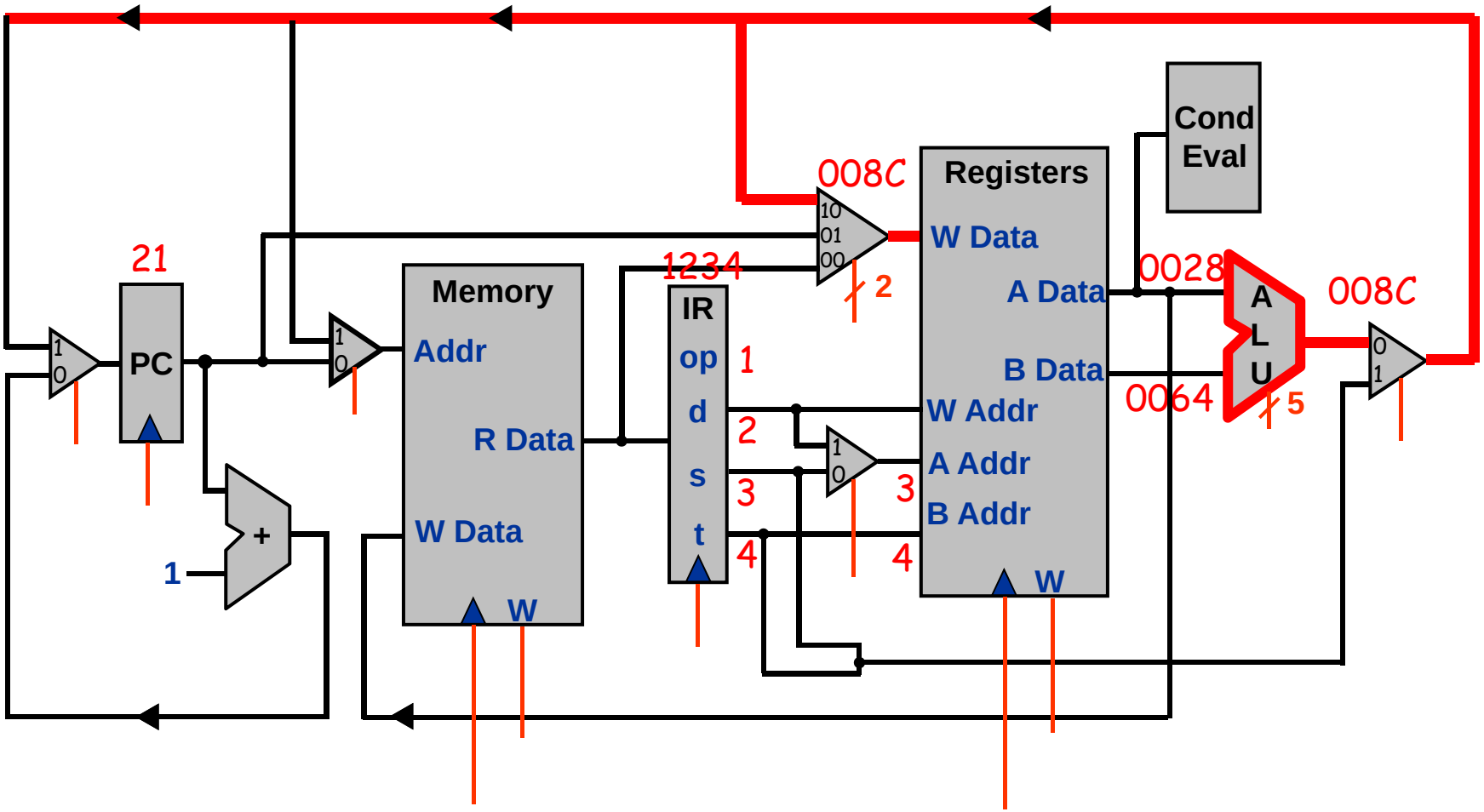
Example: ADD (execute)



Mem[20]=1234
R[3]=0028 R[4]=0064

PC=21
IR=1234

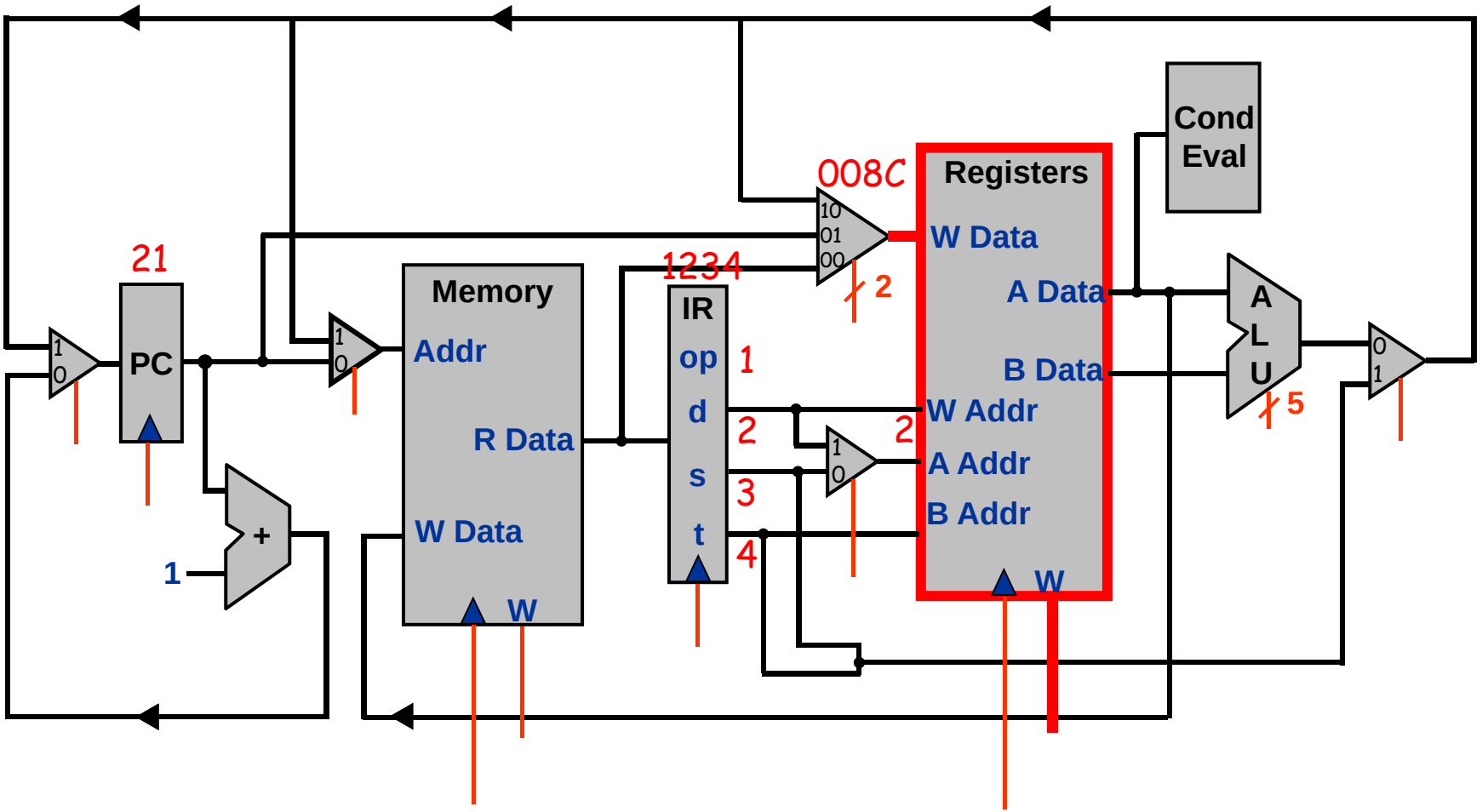
Example: ADD (execute)



Mem[20]=1234
R[3]=0028 R[4]=0064

PC=21
IR=1234

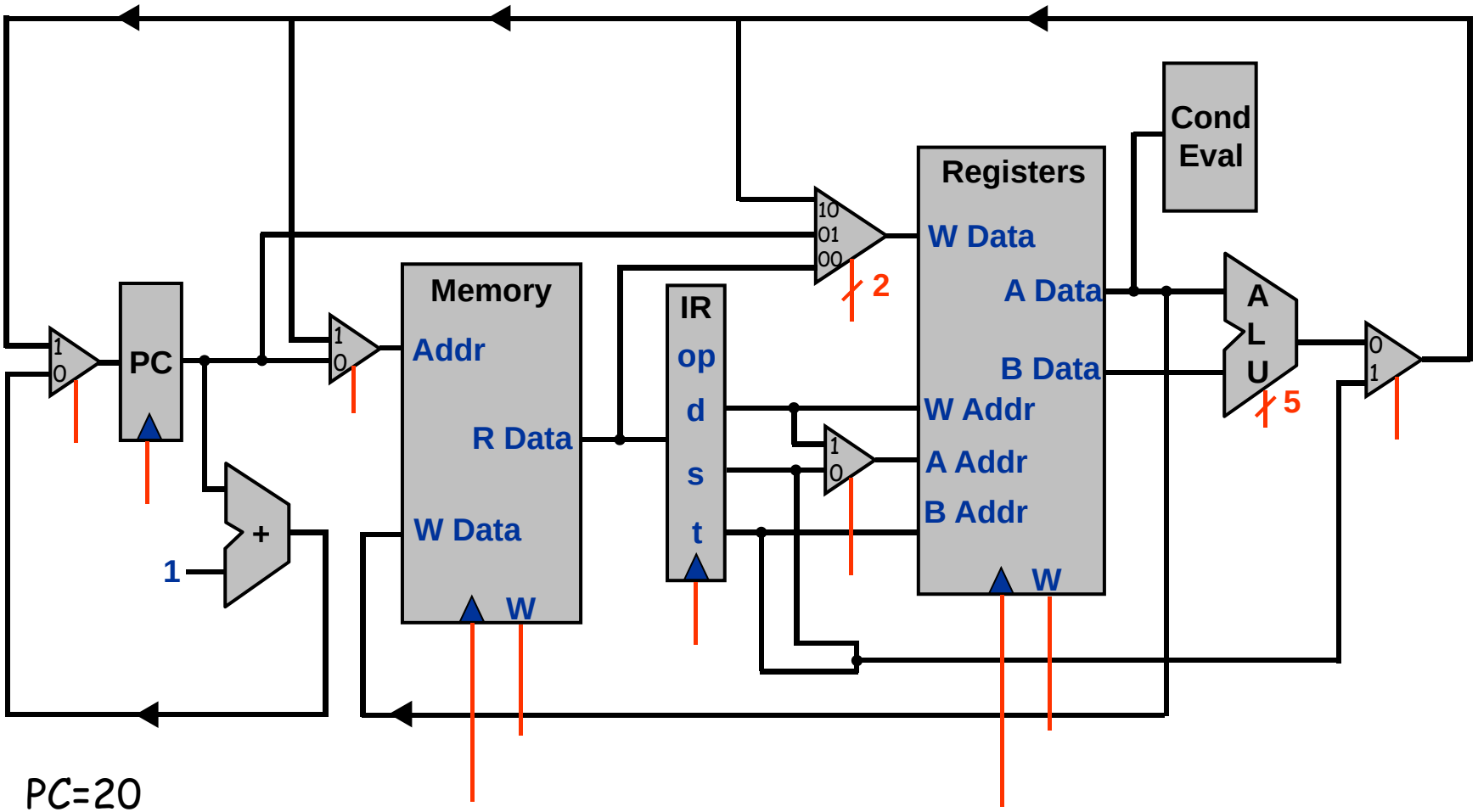
Example: ADD (execute and clock)



Mem[20]=1234
R[3]=0028 R[4]=0064

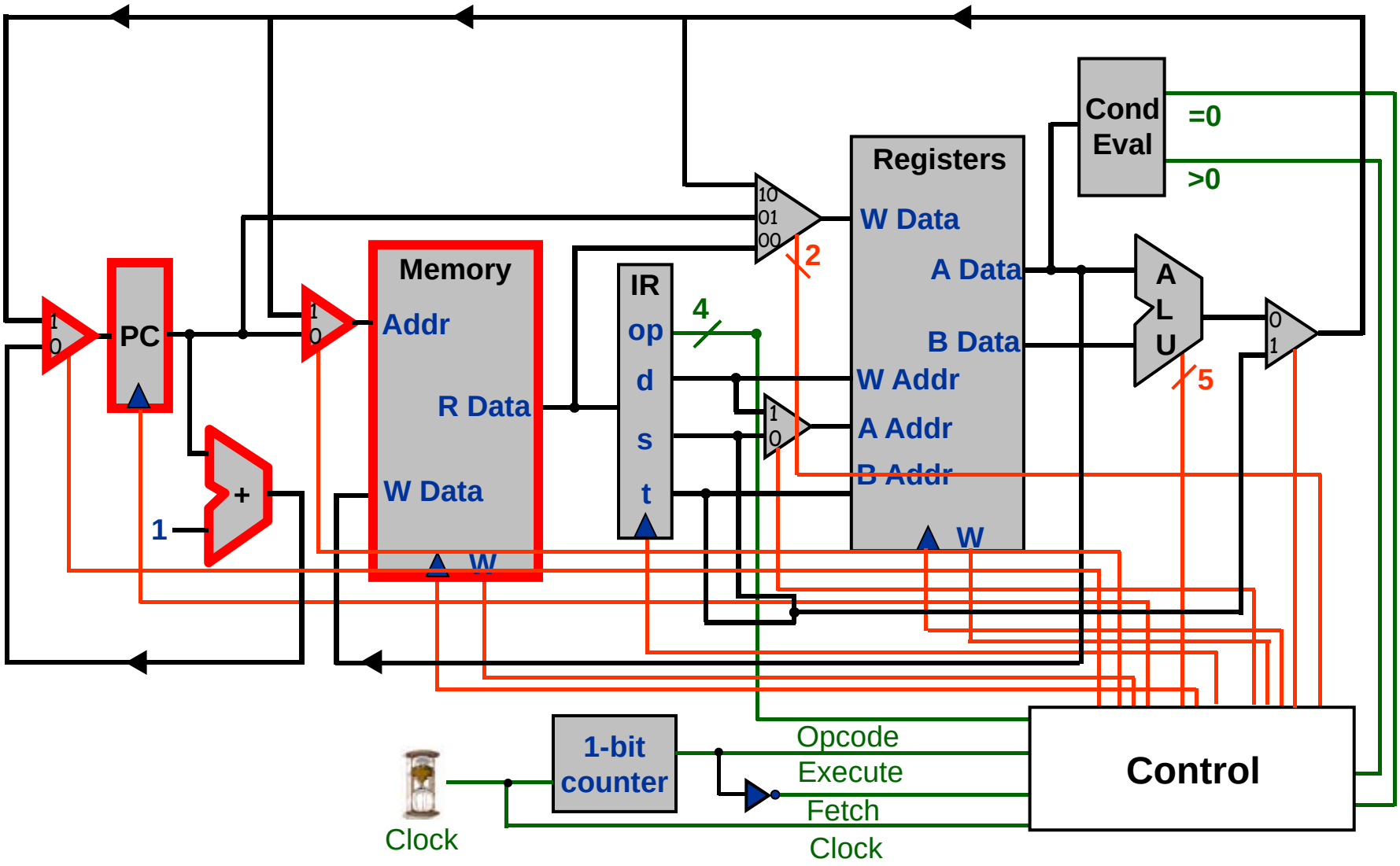
PC=21
IR=1234
R[2]=008C

Example: Jump and link

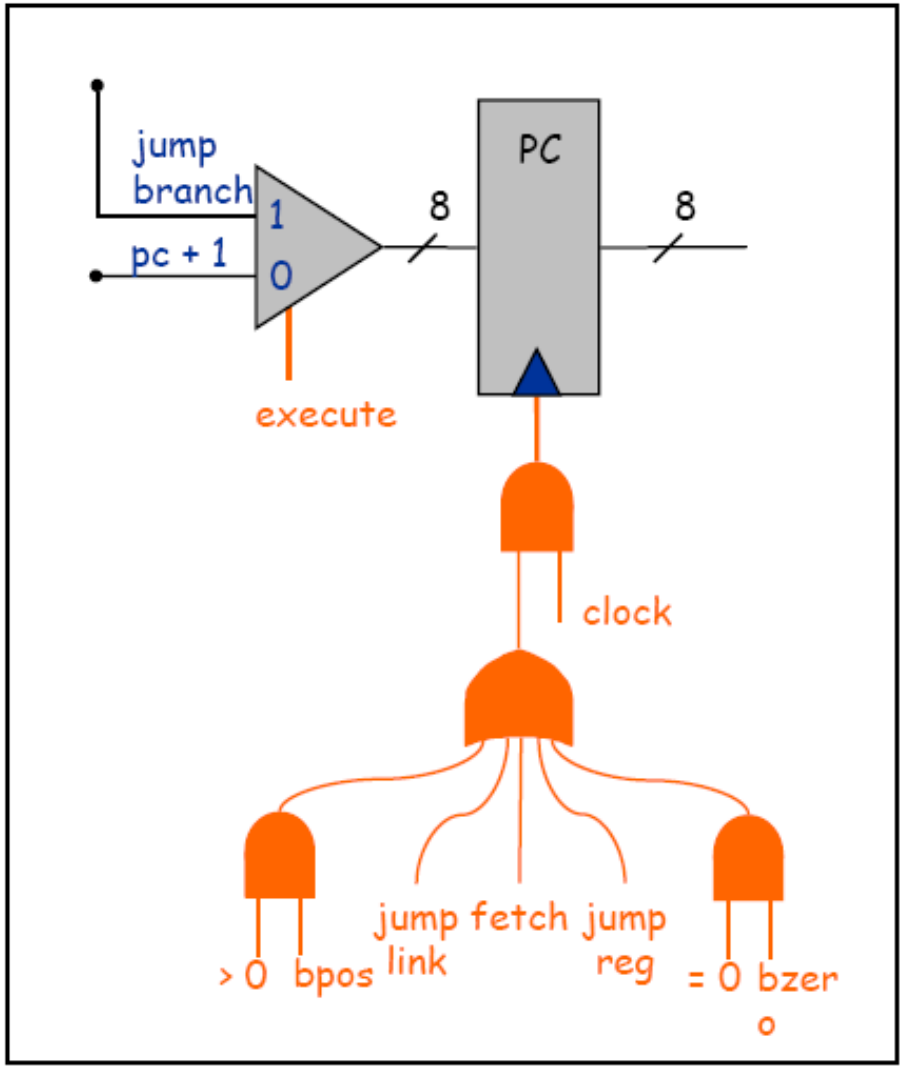


PC=20
Mem[20]=FF30
R[3]=0028 R[4]=0064

Fetch


Cond Eval
=0
>0
Registers
W Data
A Data
B Data
W Addr
A Addr
B Addr
W
10
01
00
2
Memory
Addr
R Data
W Data
W
IR
op
d
s
t
4
A
L
U
5
PC
1
0
+
1
1-bit counter
Clock
Opcode
Execute
Fetch
Clock
Control

Program counter



Program Counter

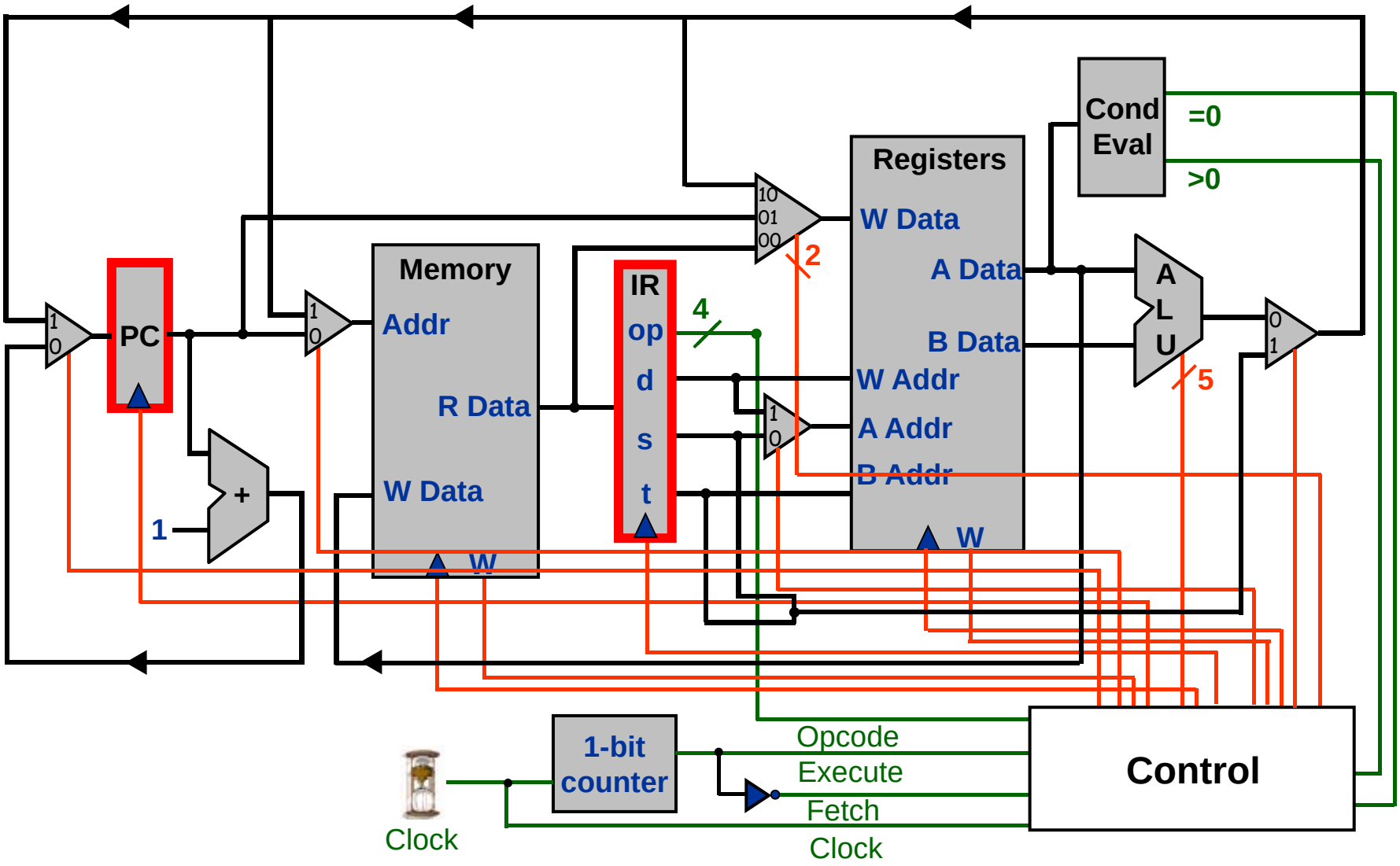
Read program counter when

- Fetch
- Execute for jal

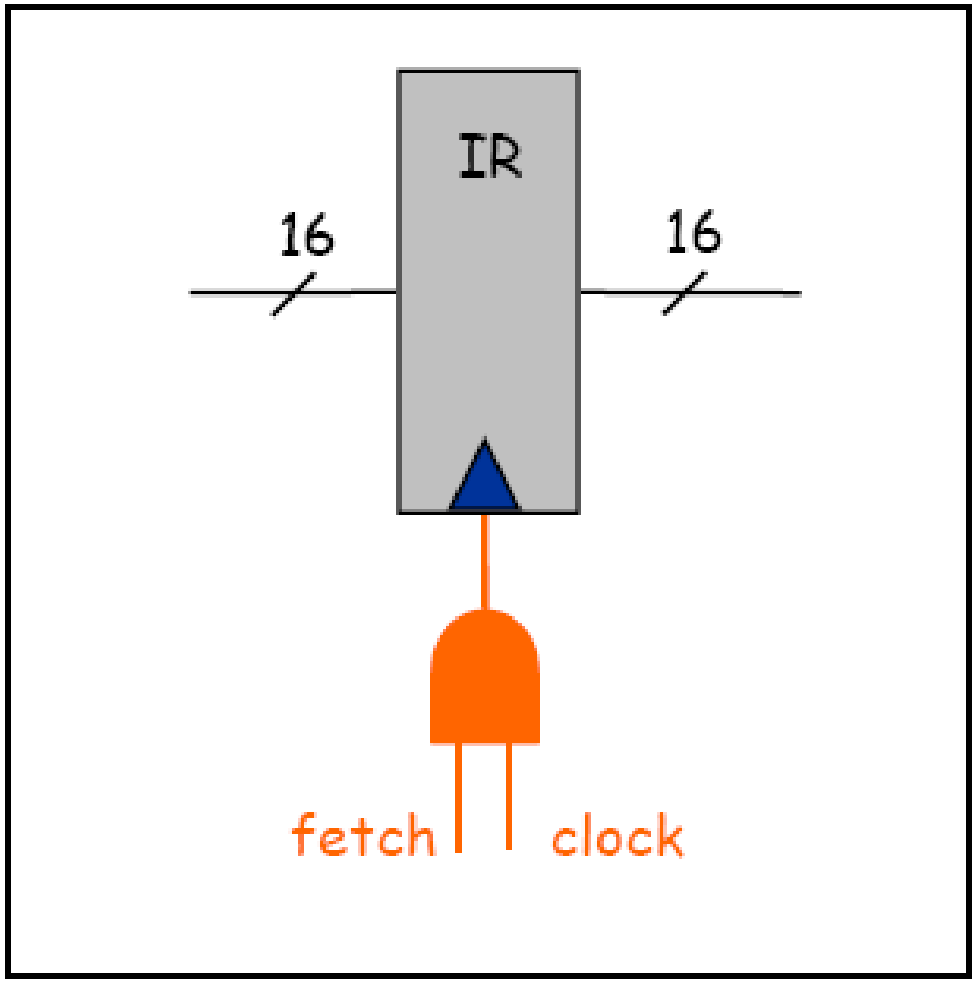
Write program counter when

- Fetch and clock
- Execute and clock depending on conditions

Fetch and clock

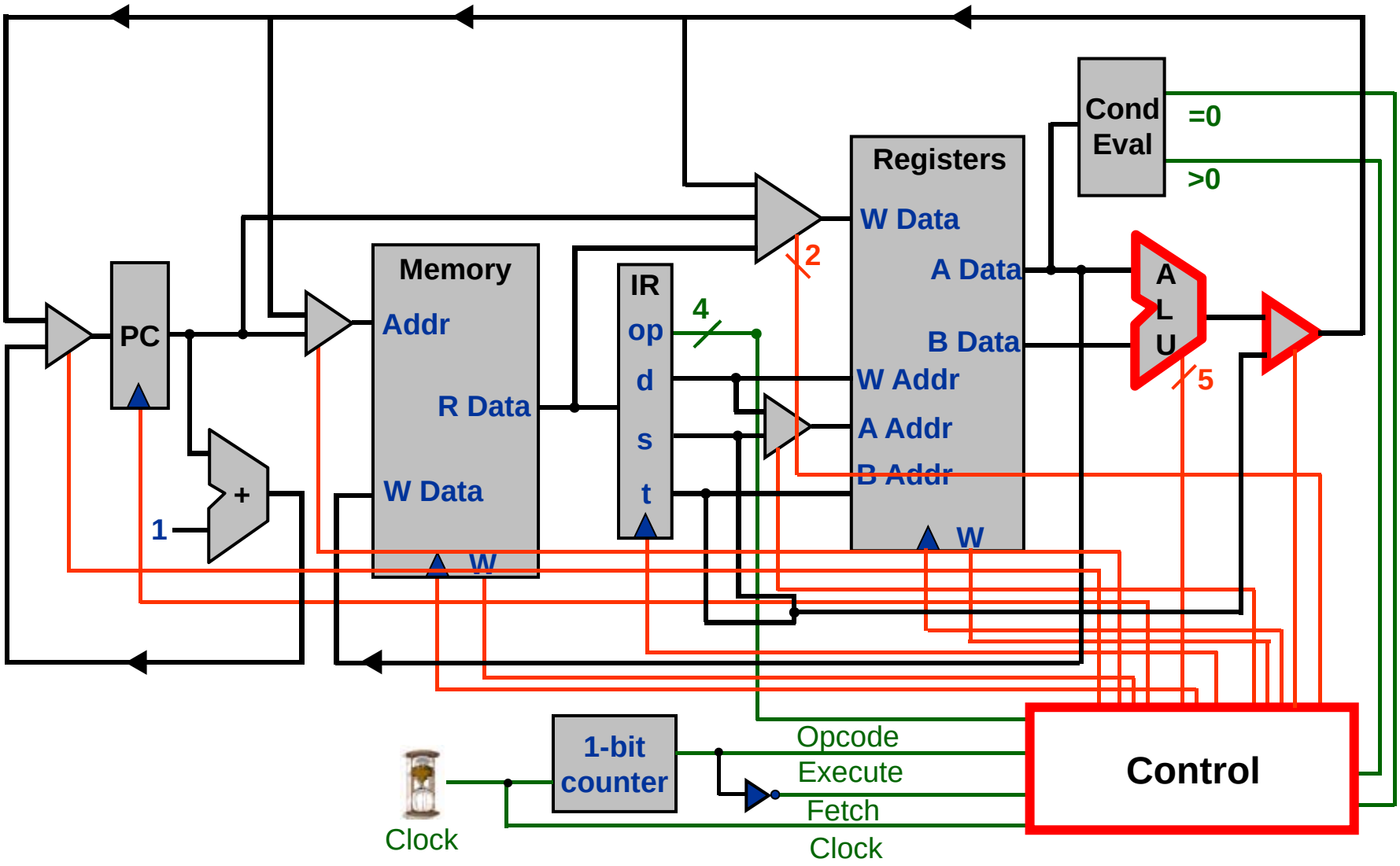


Instruction register



Instruction Register

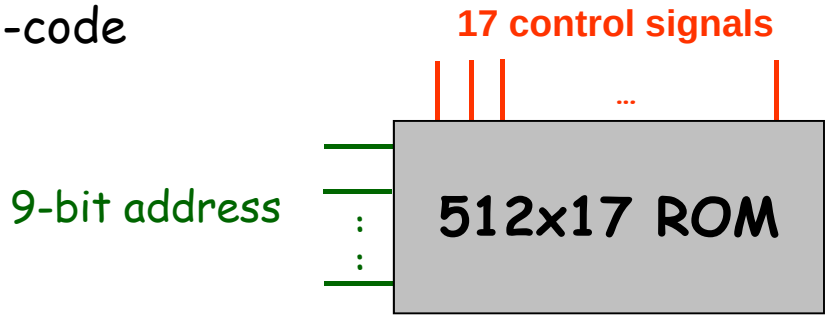
Execute

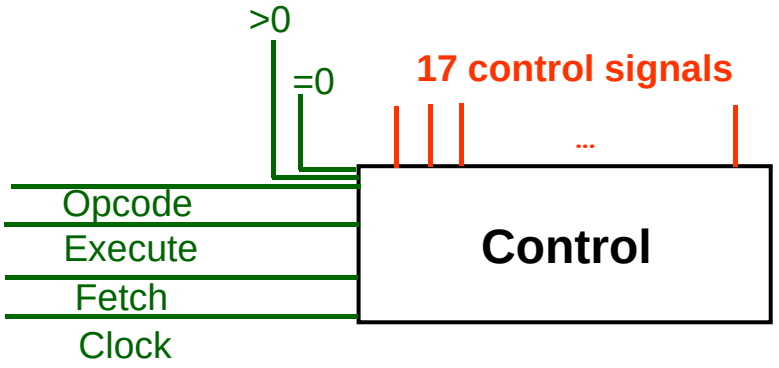

Cond Eval
=0
>0
Registers
W Data
A Data
B Data
W Addr
A Addr
B Addr
W
2
ALU
5
Memory
Addr
R Data
W Data
W
IR
op
d
s
t
4
PC
+
1
1-bit counter
Clock
Opcode
Execute
Fetch
Clock
Control

Control

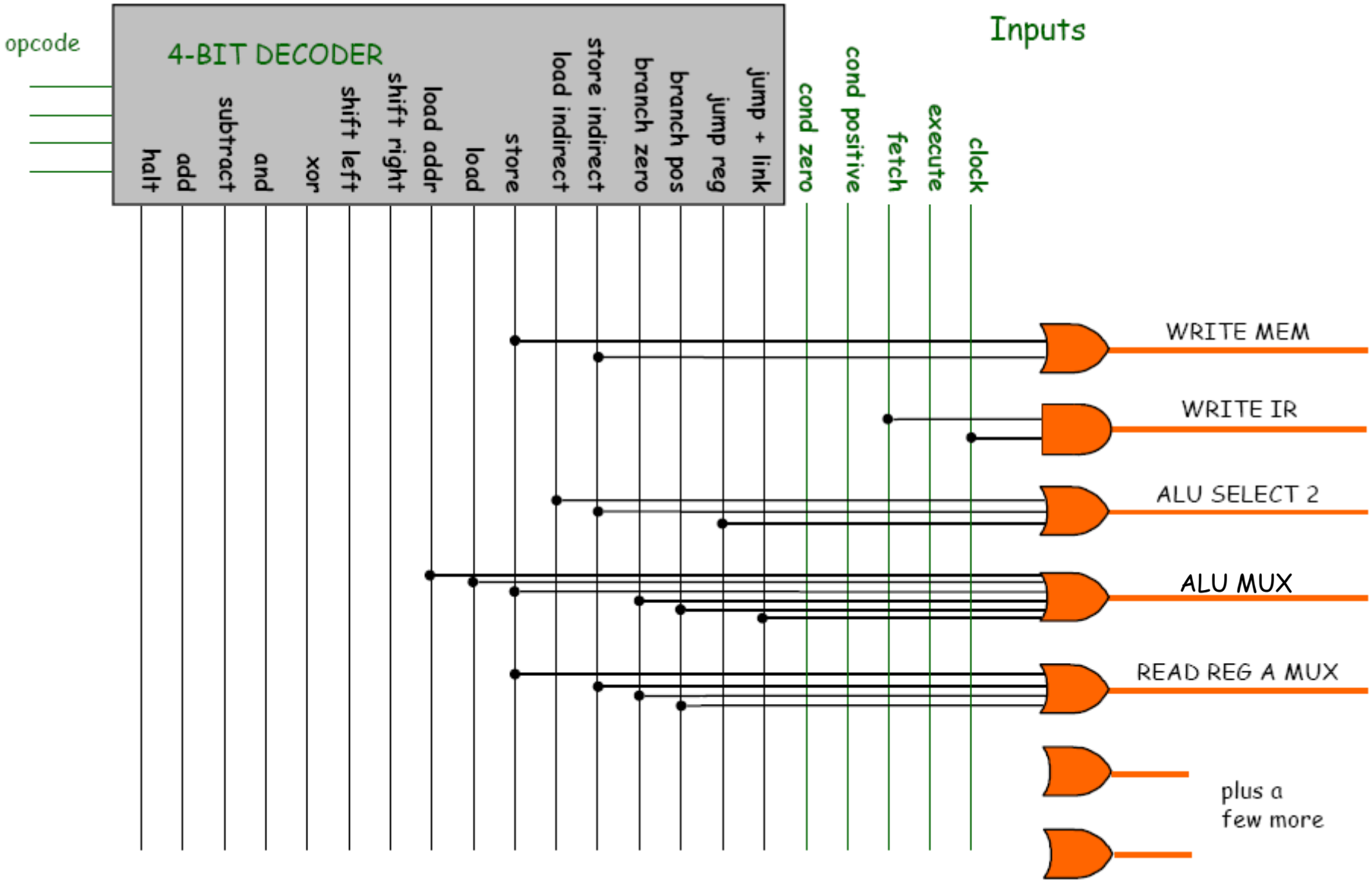
Two approaches to implement control

- Micro-programming
  - Use a memory (ROM) for micro-code
  - More flexible
  - Easier to program
- Hard-wired
  - Use logic gates and wire
  - More efficient

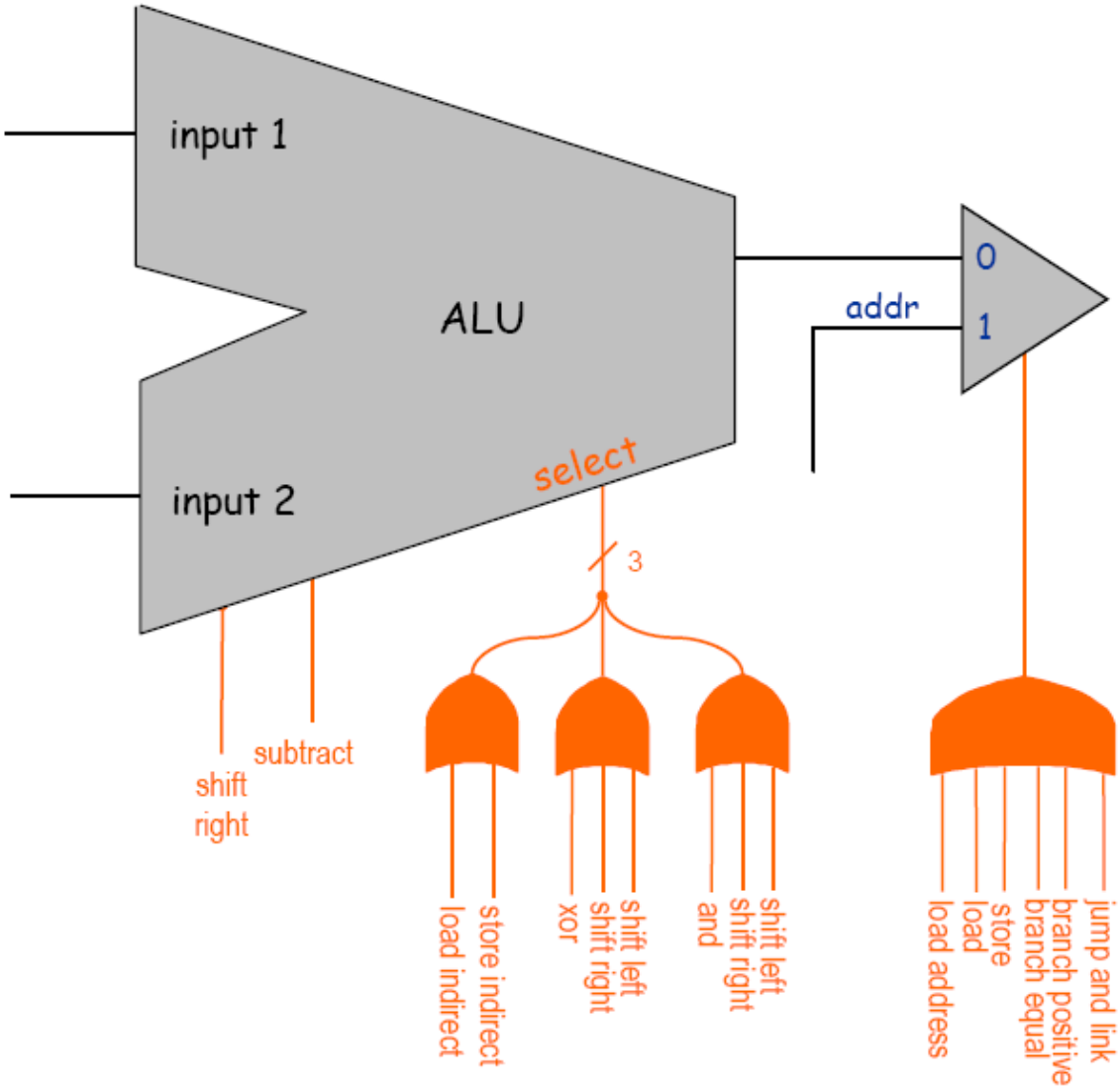




Control

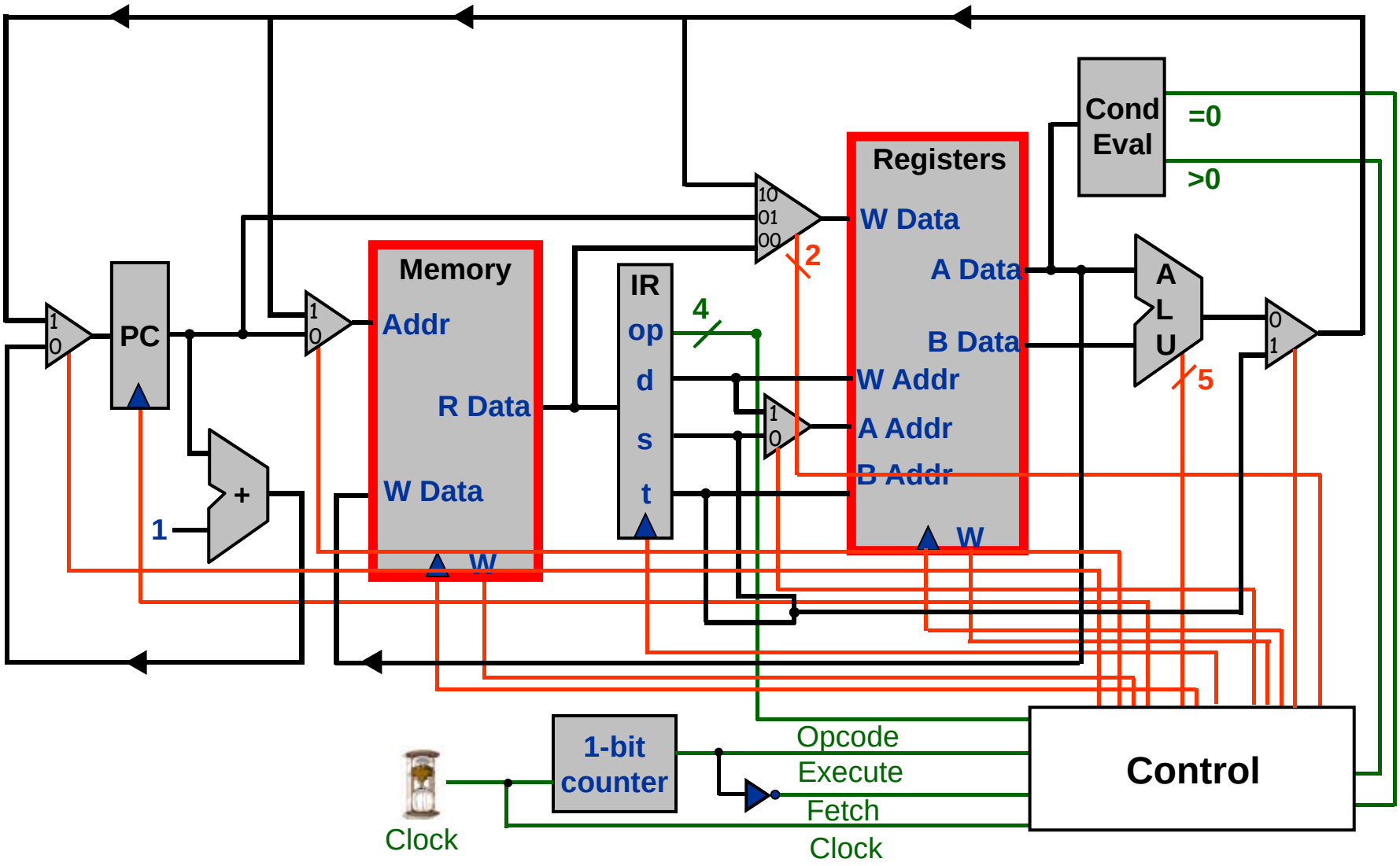

opcode
4-BIT DECODER
halt
add
subtract
and
xor
shift left
shift right
load addr
load
store
store indirect
load indirect
branch zero
branch pos
jump reg
jump + link
Inputs
cond zero
cond positive
fetch
execute
clock
WRITE MEM
WRITE IR
ALU SELECT 2
ALU MUX
READ REG A MUX
plus a
few more

ALU control

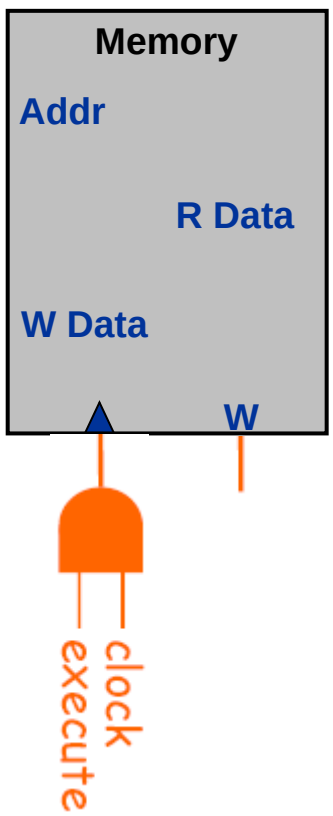


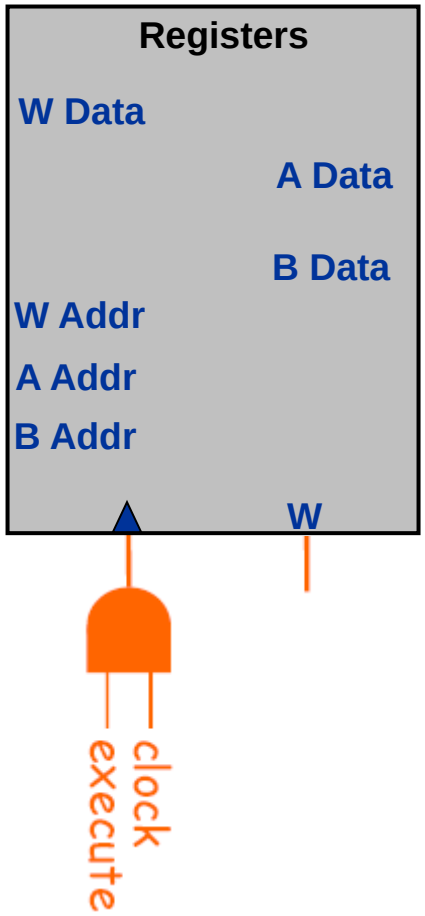
op	2	1	0
+, -	0	0	0
&	0	0	1
^	0	1	0
<<, >>	0	1	1
input 2	1	0	0

Execute and clock (write-back)

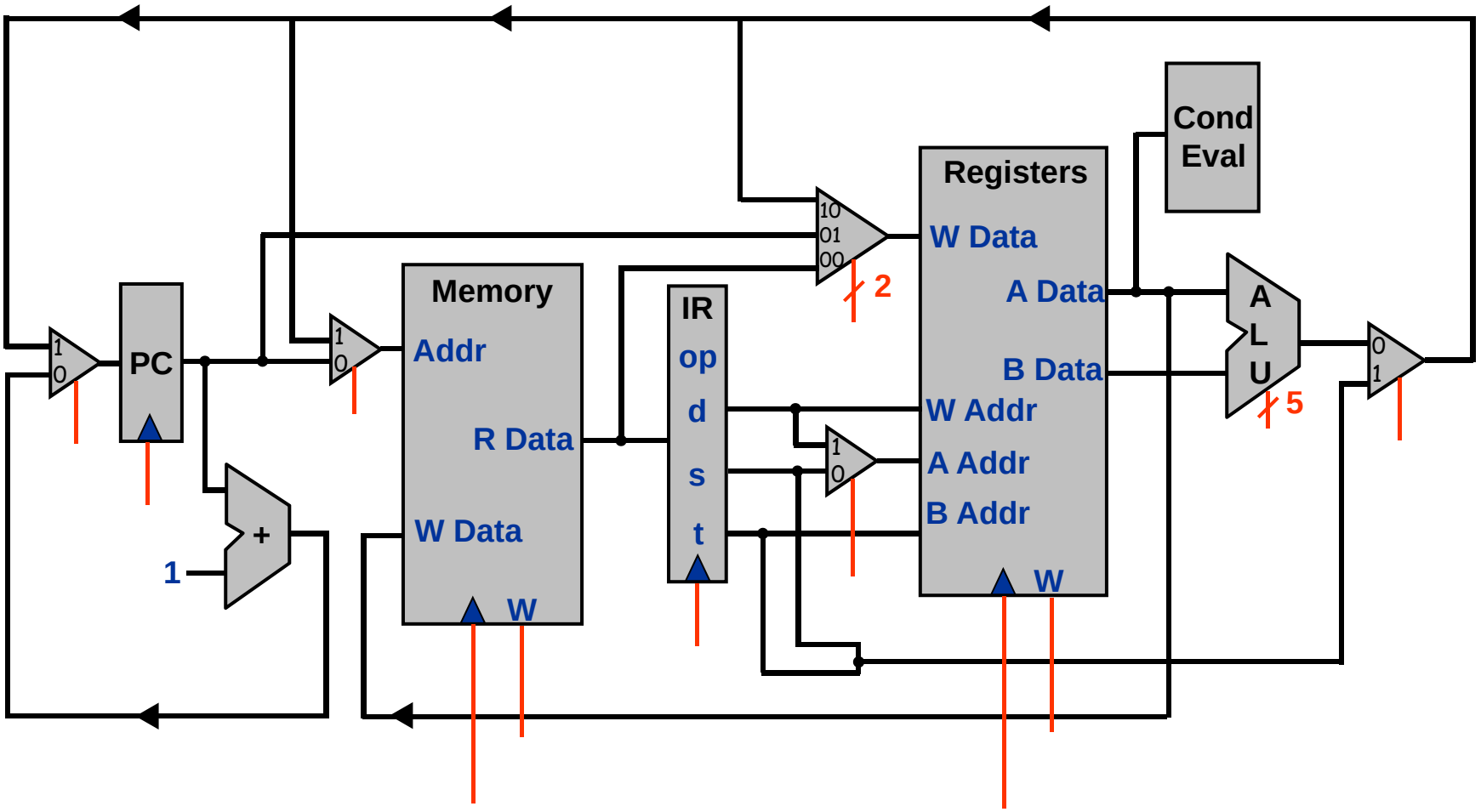


Writing registers and memory

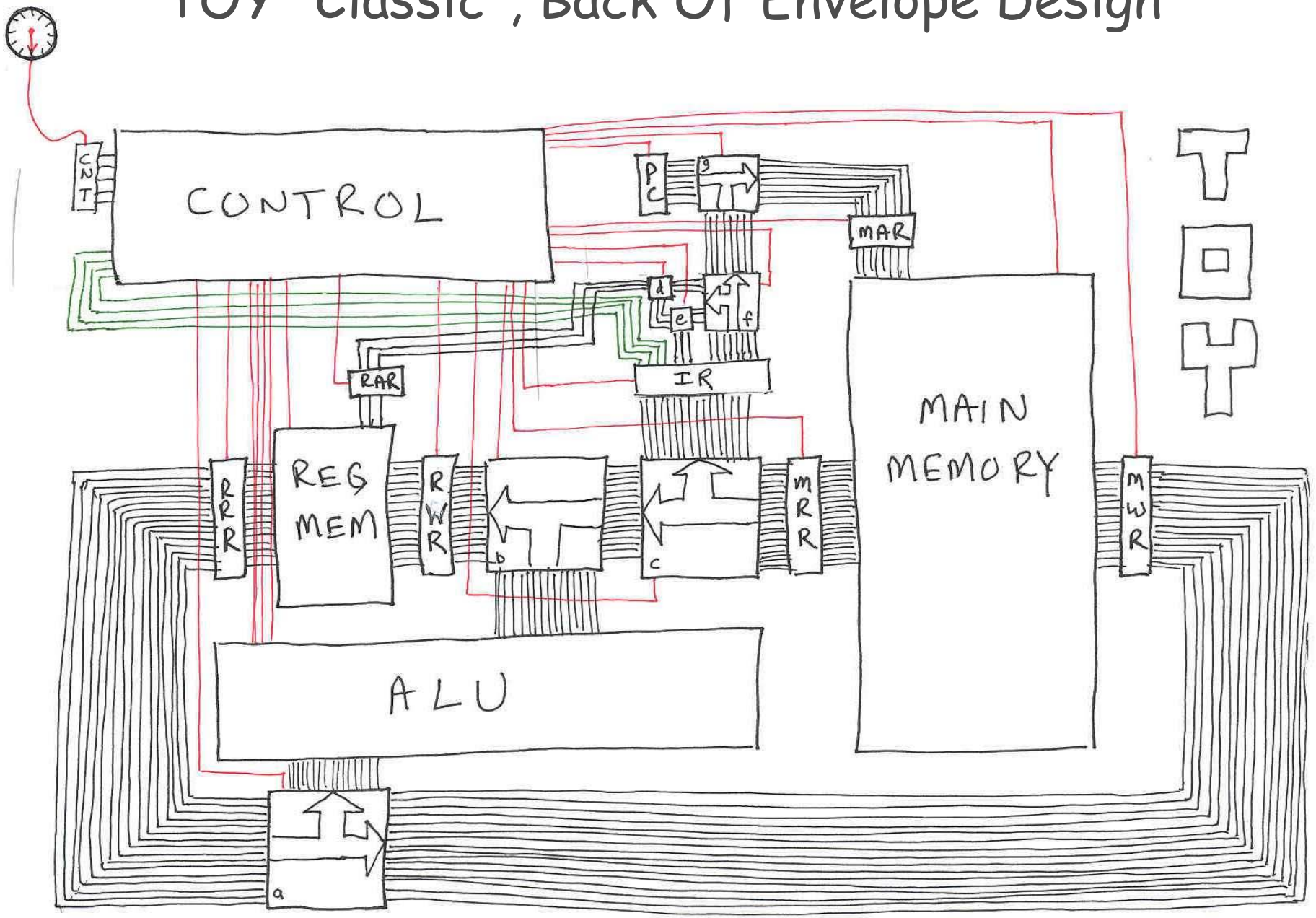




More examples

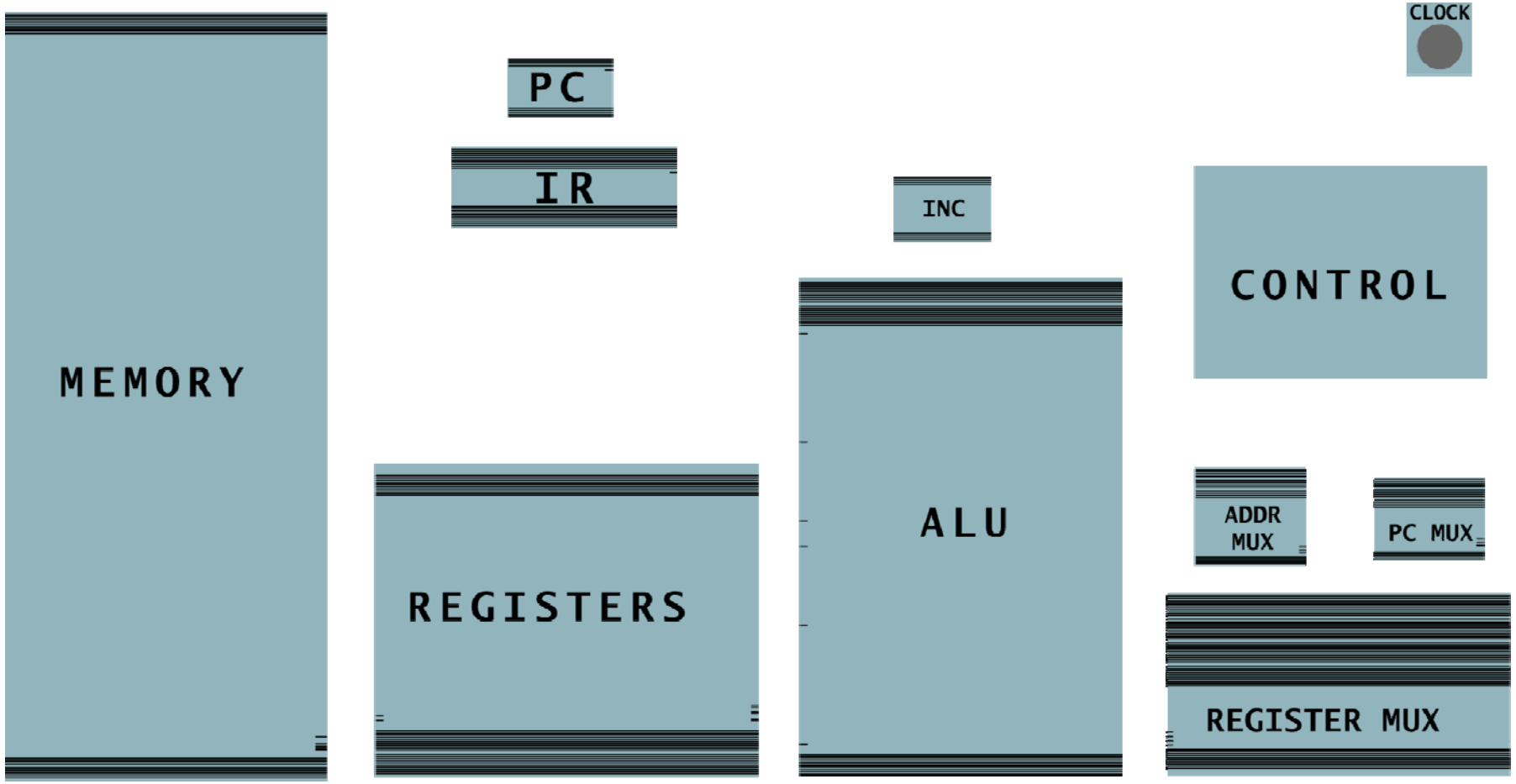


TOY "Classic", Back Of Envelope Design



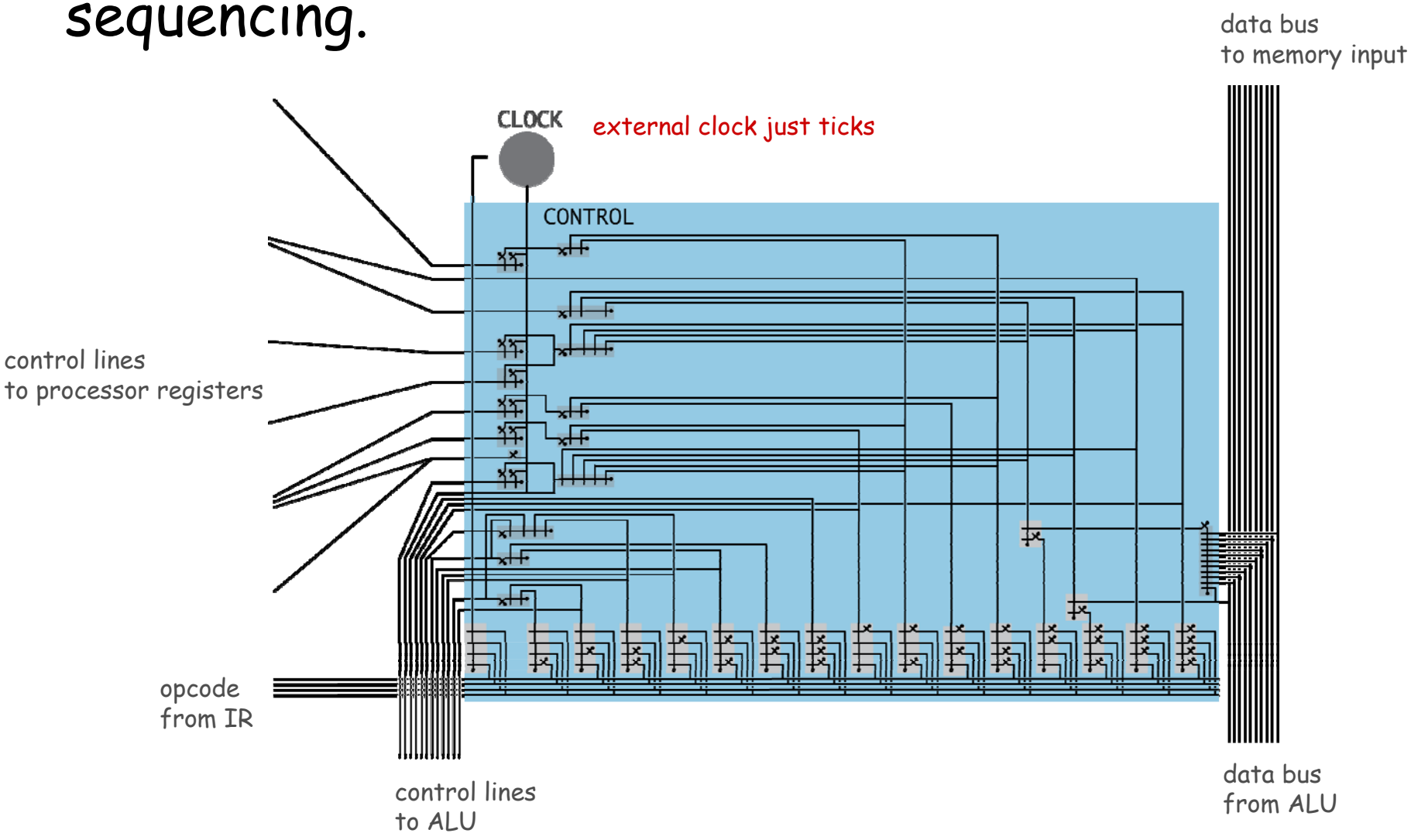
Build a TOY-Lite: Devices

10-bit word,
4-word register
16-word memory

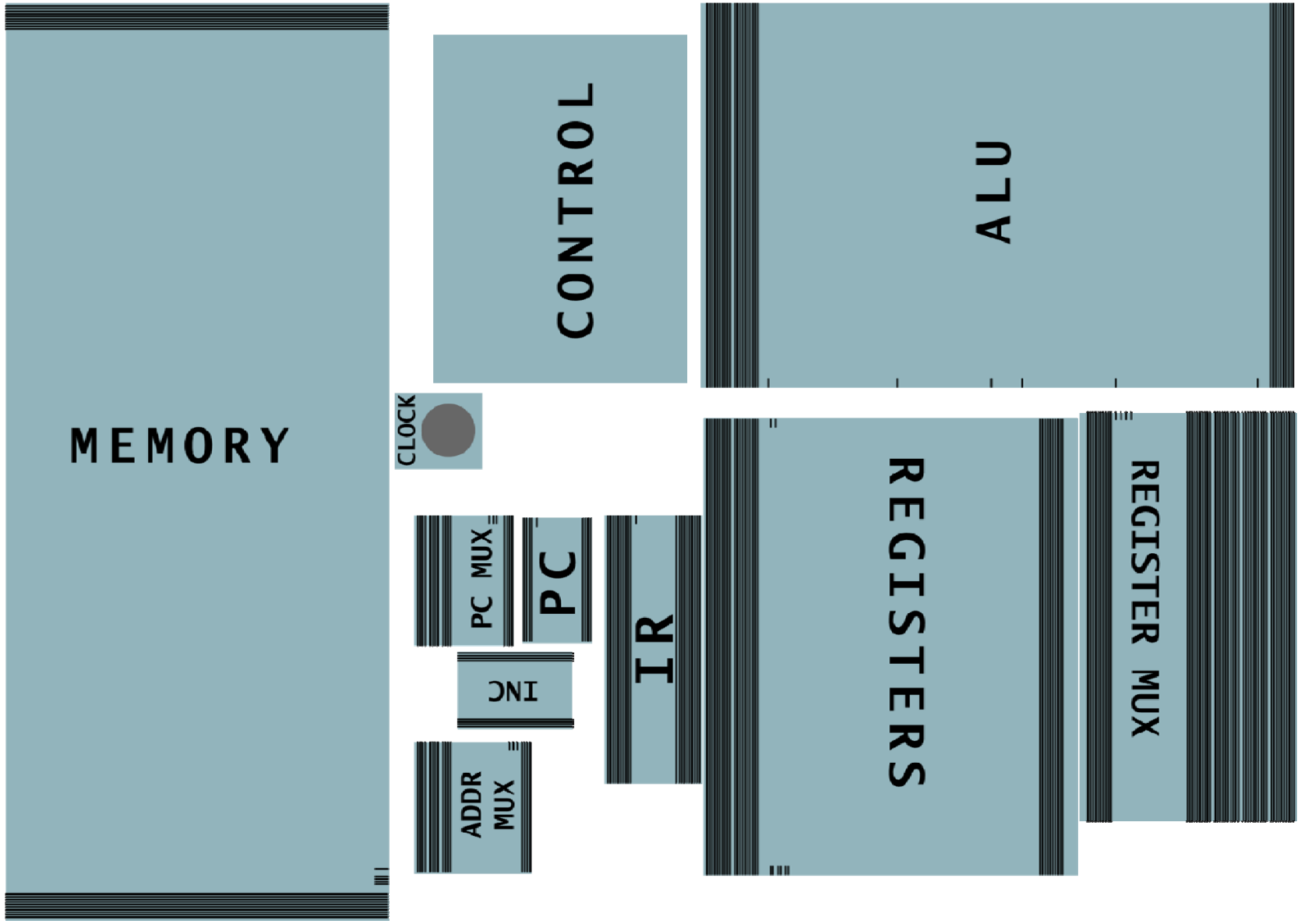


Control

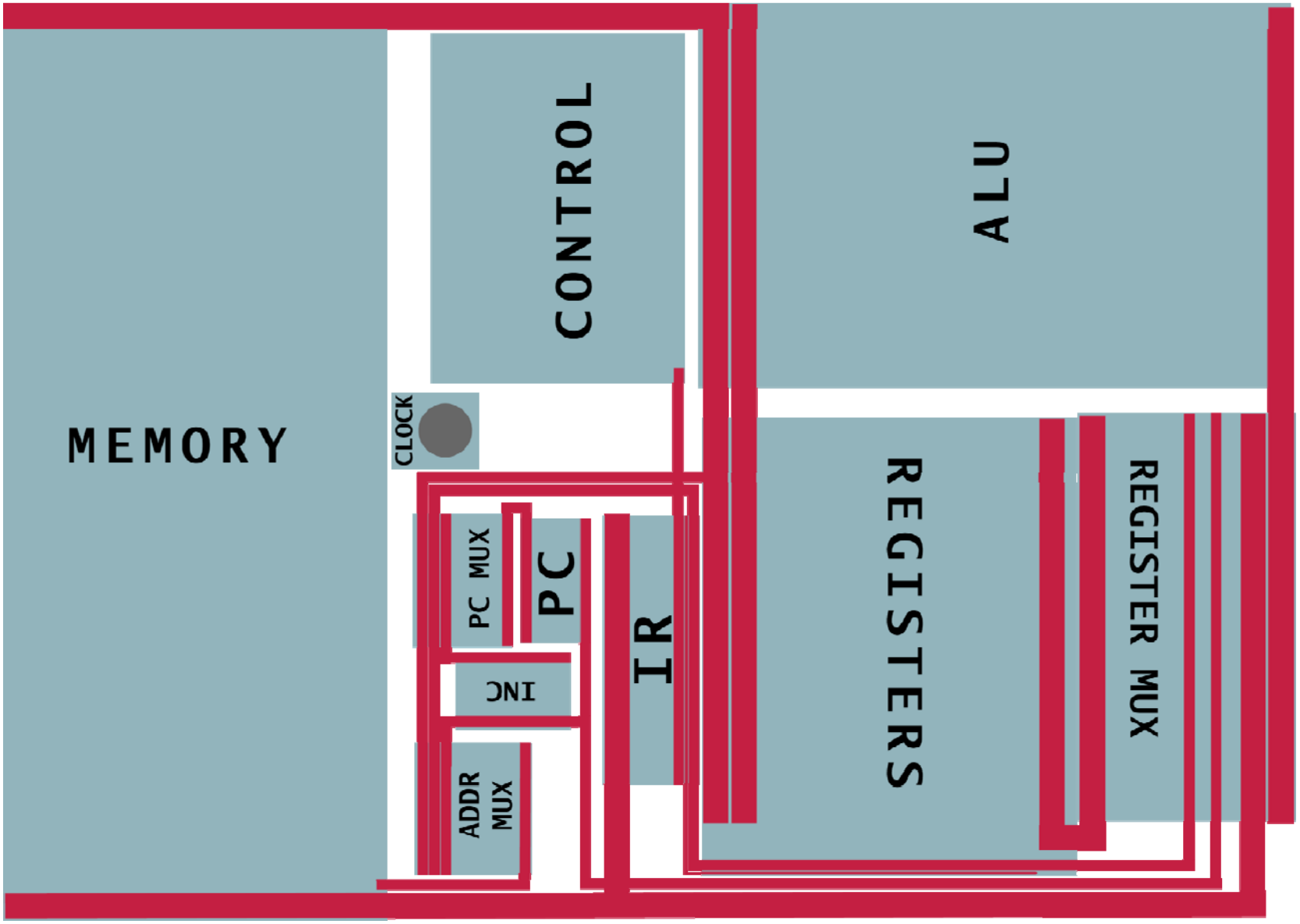
Control. Circuit that determines control line sequencing.



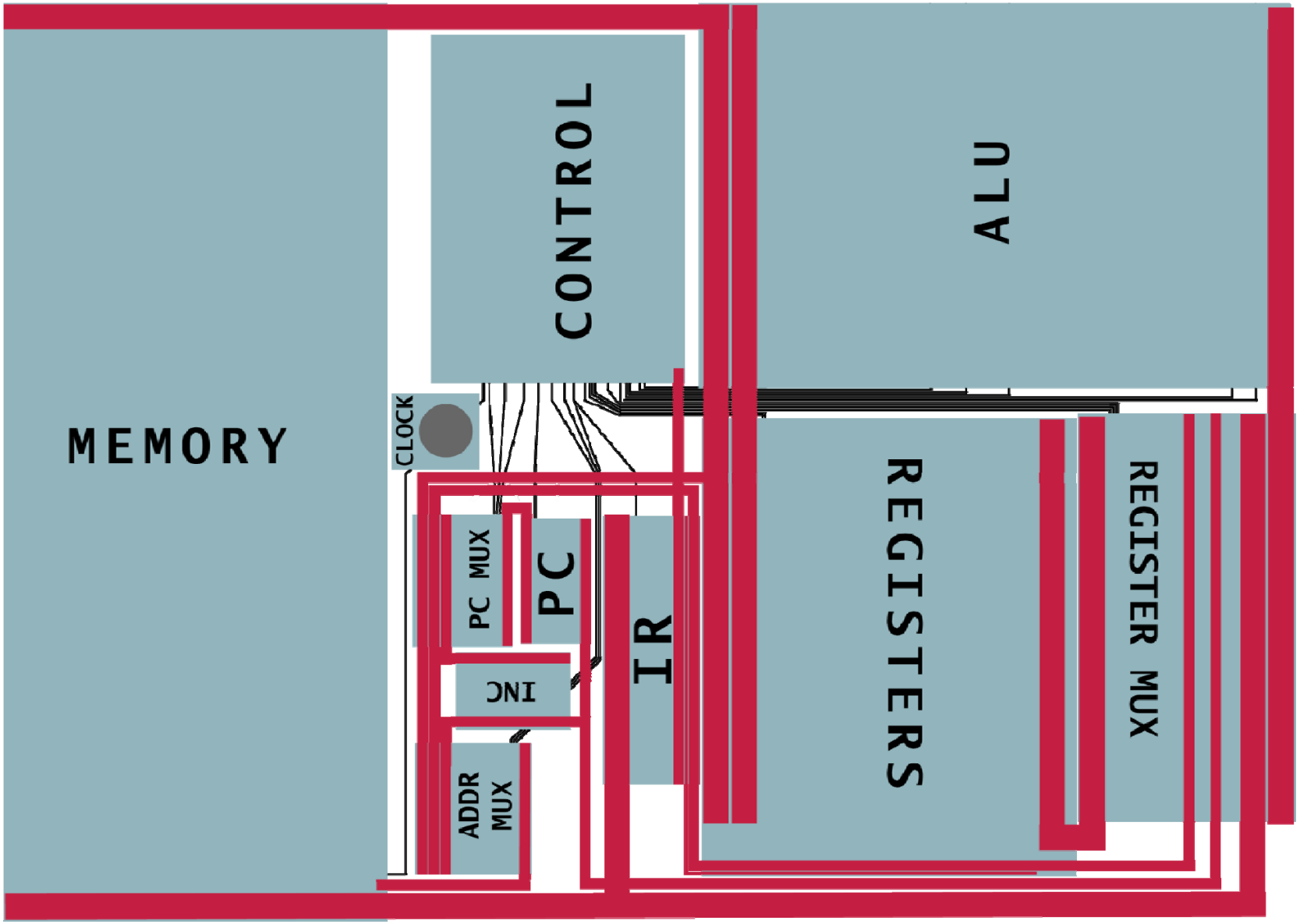
Build a TOY-Lite: Layout



Build a TOY-Lite: Datapath

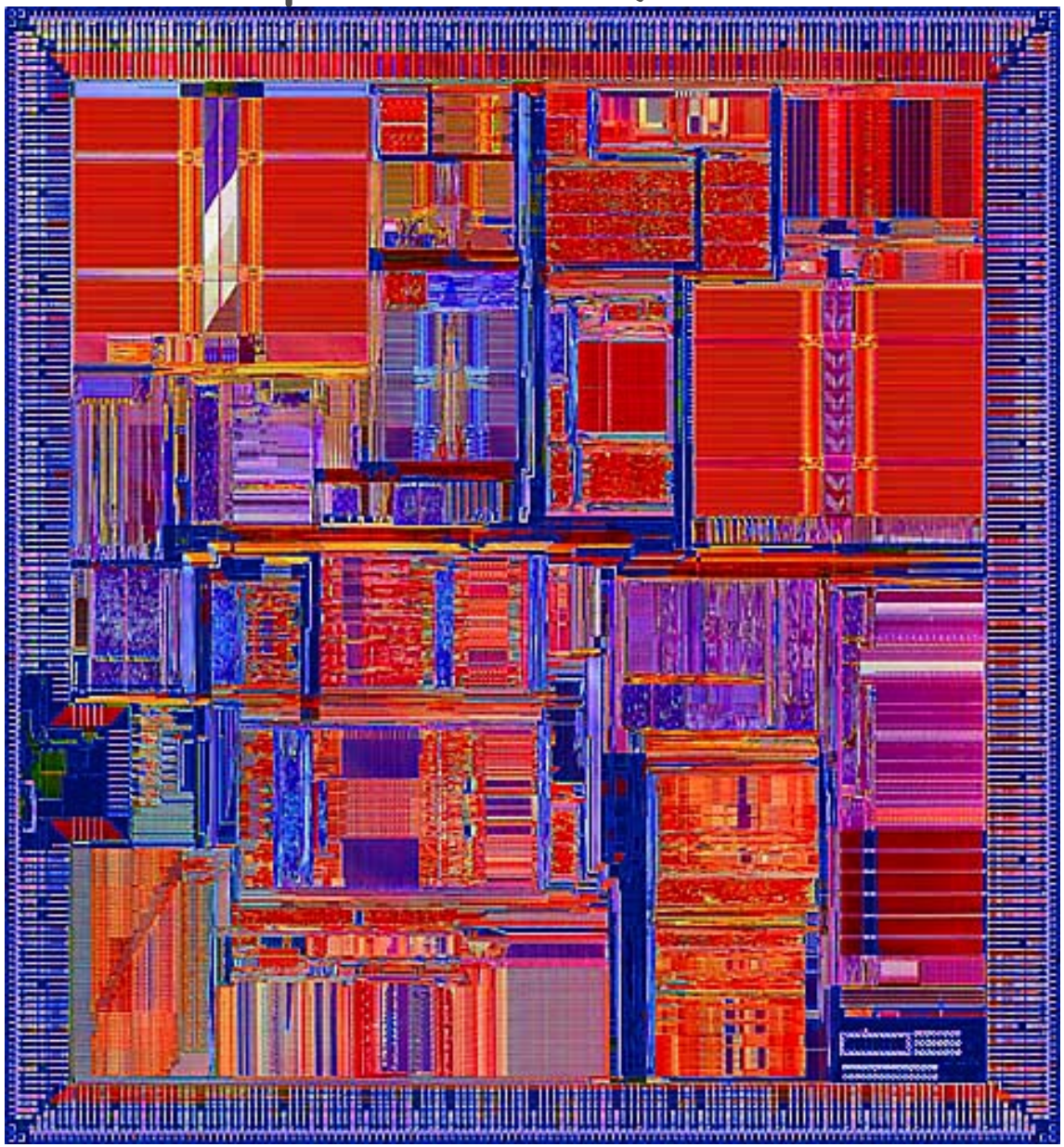


Build a TOY-Lite: Control

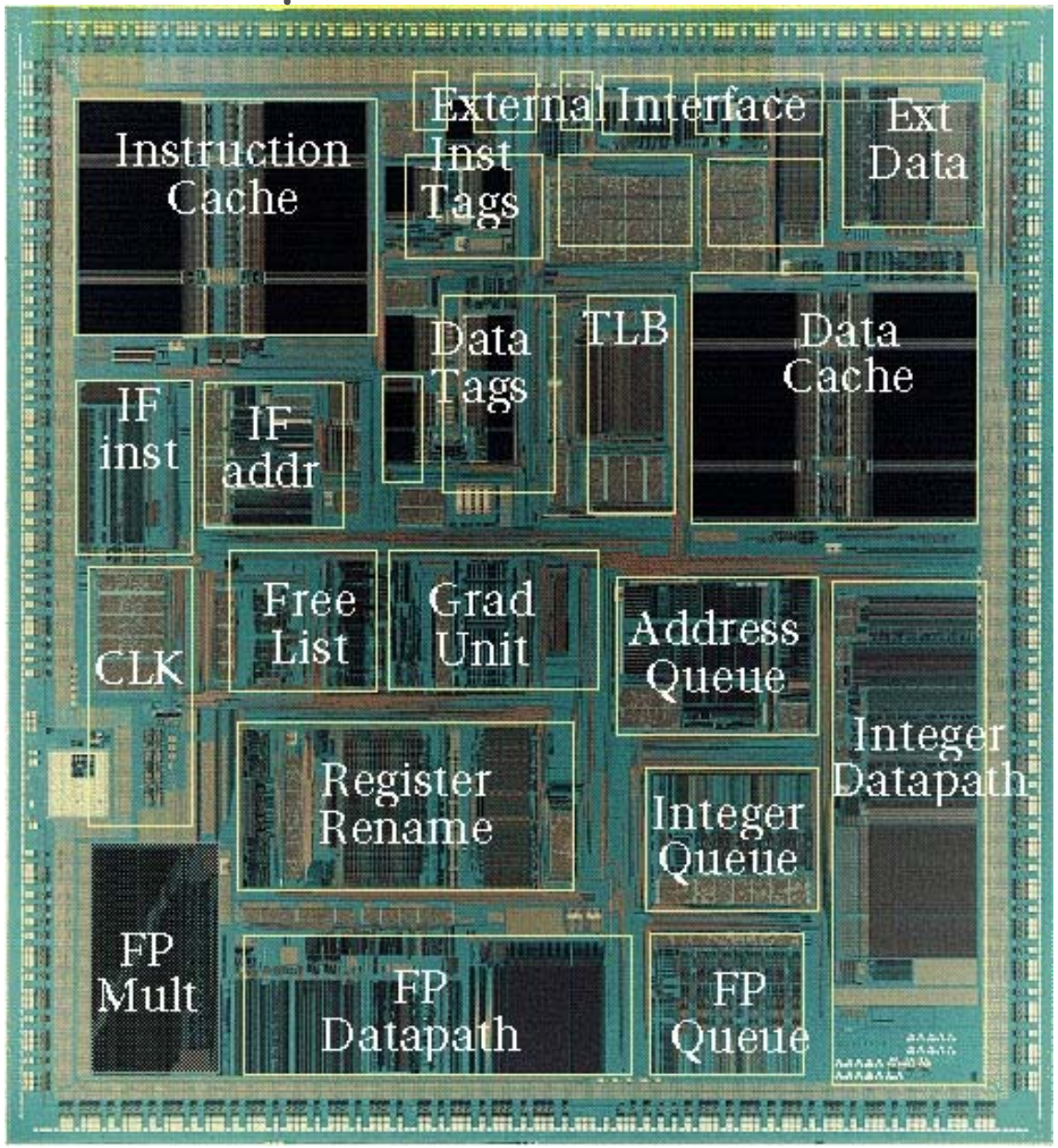


MAIN MEMORY (16 10 BIT WORDS)
CONTROL
CLOCK
ARITHMETIC/LOGICAL UNIT (ALU)
REGISTERS (4 10-BIT WORDS)
REGISTER INPUT MUX
PC INPUT MUX
INSTRUCTION REGISTER (IR)
MEMORY ADDRESS MUX

Real Microprocessor (MIPS R10000)



Real Microprocessor (MIPS R10000)



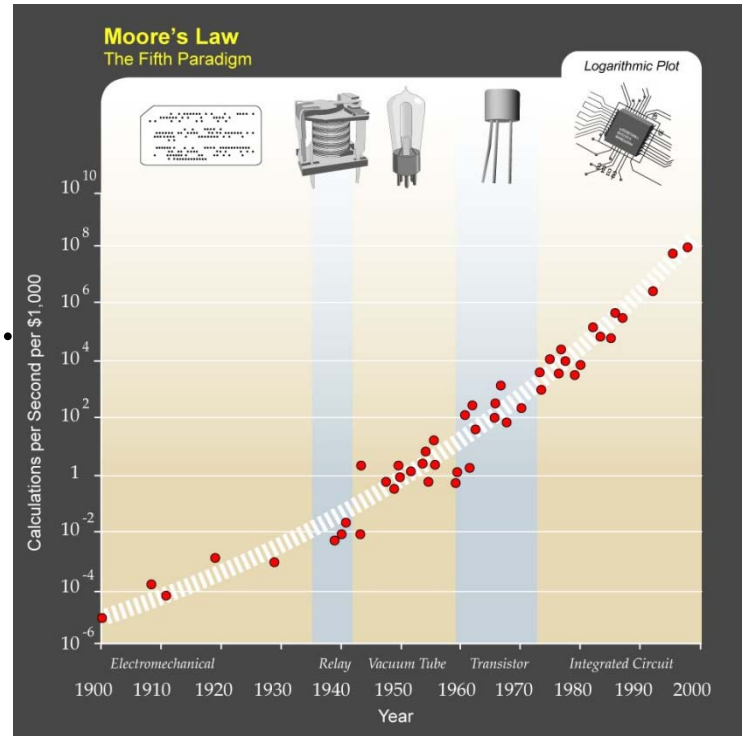
History + Future

Computer constructed by layering abstractions.

- Better implementation at low levels improves everything.
- Ongoing search for better abstract switch!

History.

- 1820s: mechanical switches.
- 1940s: relays, vacuum tubes.
- 1950s: transistor, core memory.
- 1960s: integrated circuit.
- 1970s: microprocessor.
- 1980s: VLSI.
- 1990s: integrated systems.
- 2000s: web computer.
- Future: quantum, optical soliton, ...



Ray Kurzweil (`http://en.wikipedia.org/wiki/Image:PPTMooresLawai.jpg`)