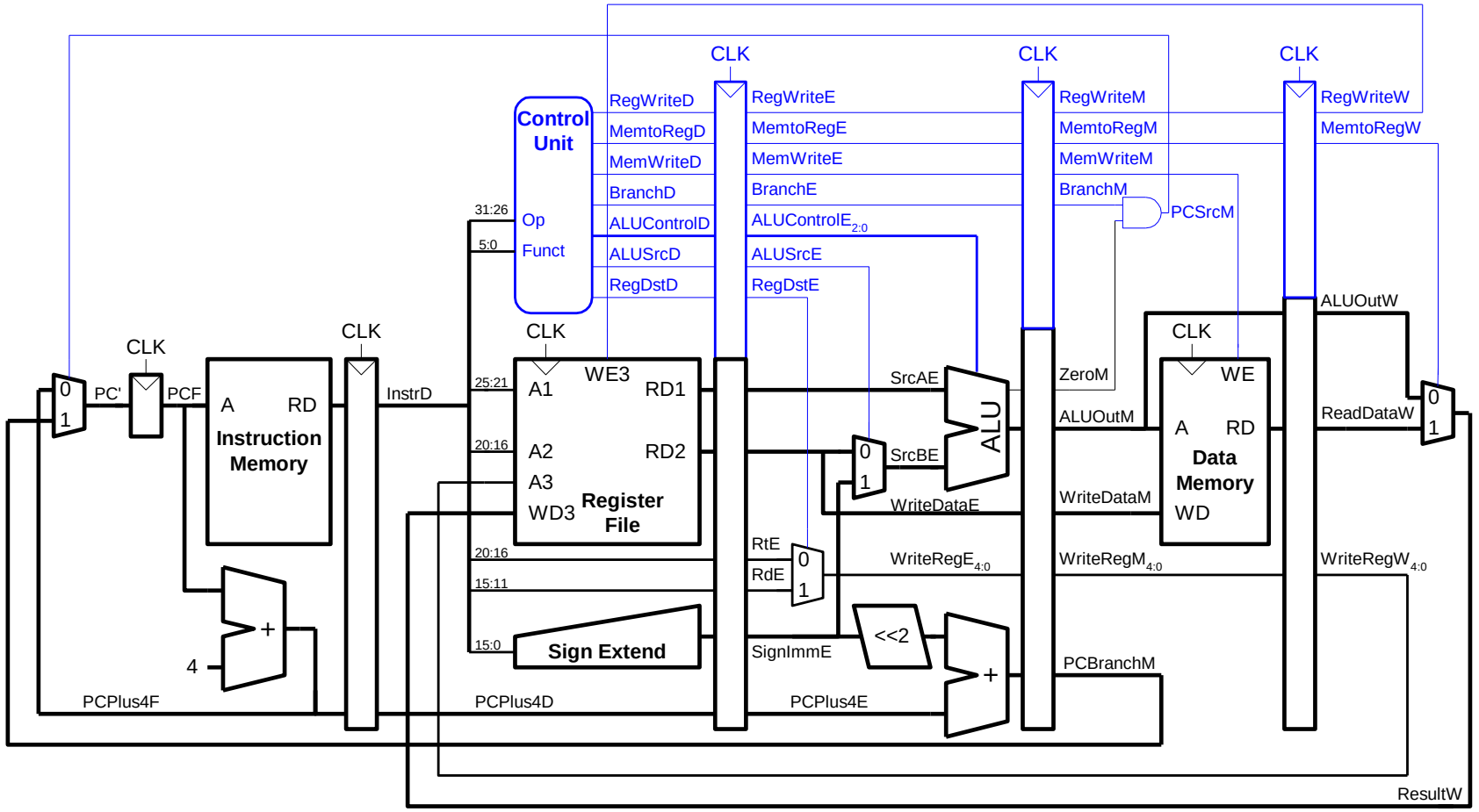
CHW 362 : Computer Architecture & Organization

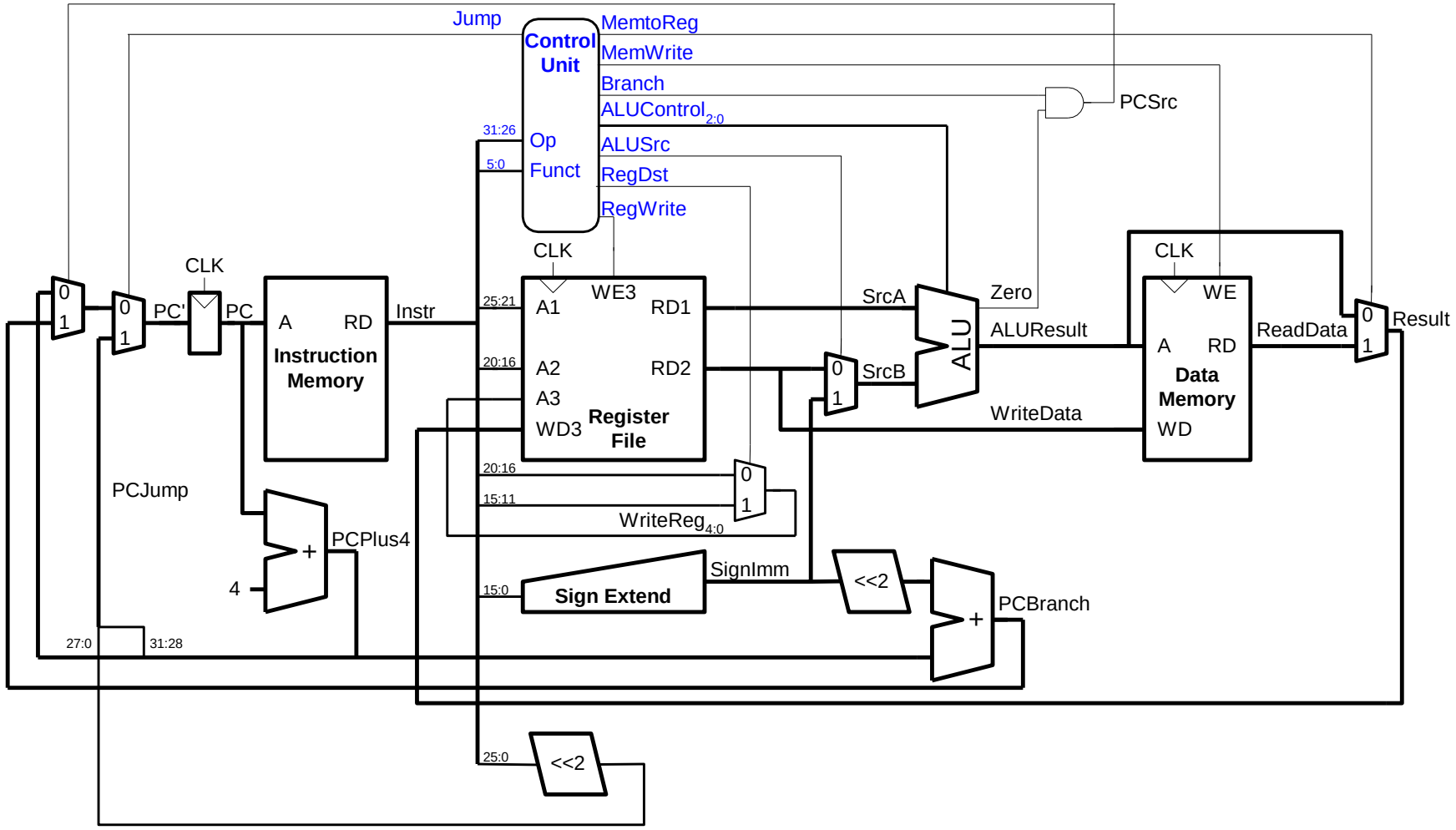
Instructors:
Dr Ahmed Shalaby http://bu.edu.eg/staff/ahmedshalaby14#

Pipelined Processor Control

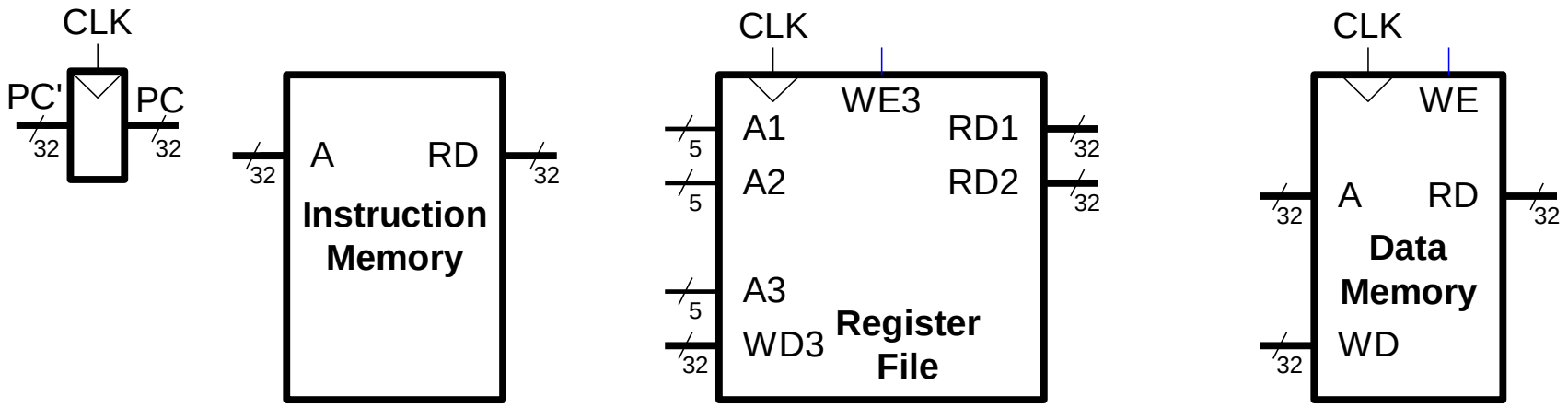


- **Same control unit as single-cycle processor**
- **Control delayed to proper pipeline stage**

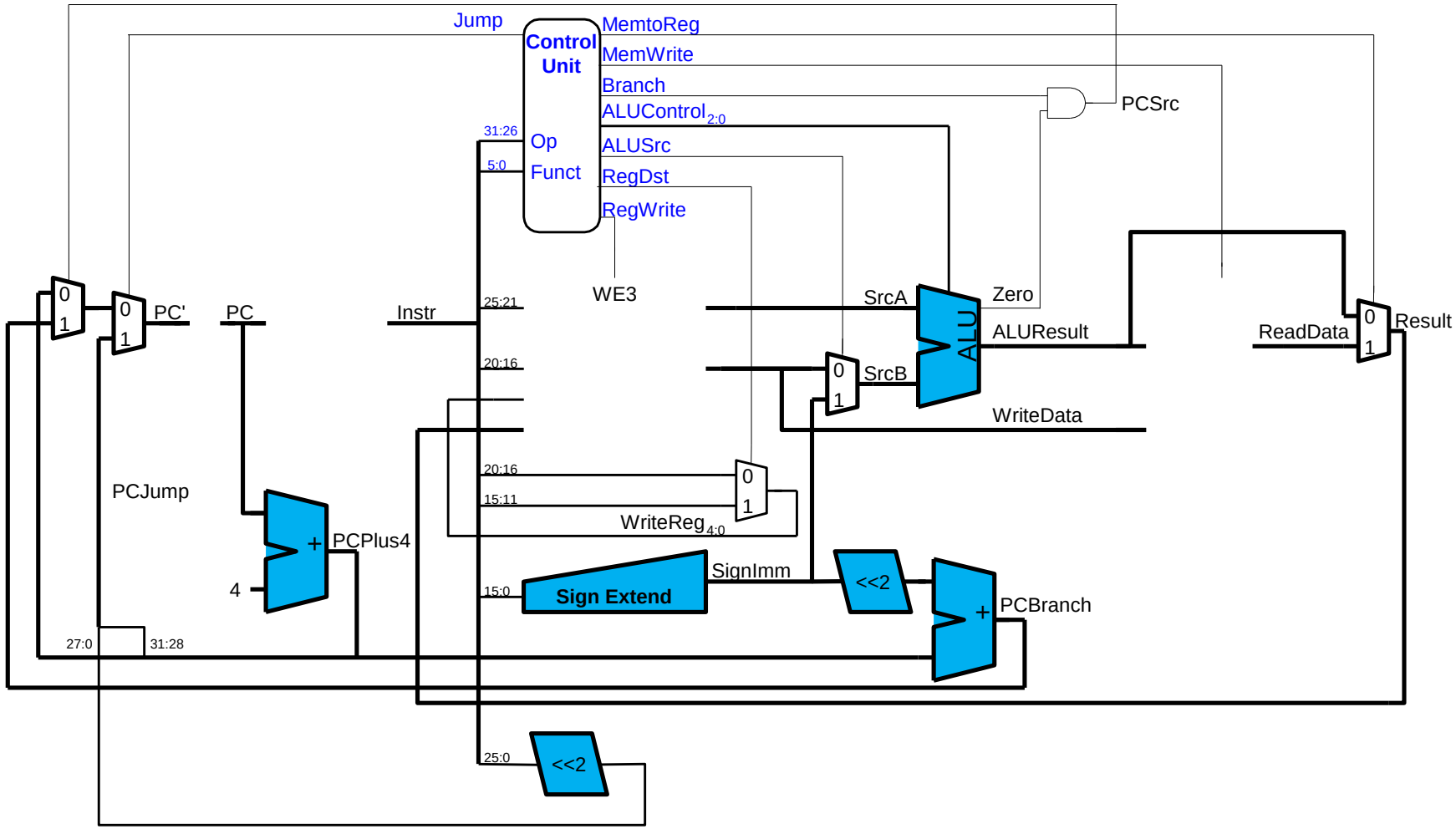
Single-Cycle Processor



Single-Cycle Processor: State Elements



Single-Cycle Processor : Control/ Combinational Elements



Processor

- **Architecture:** programmer's view of computer

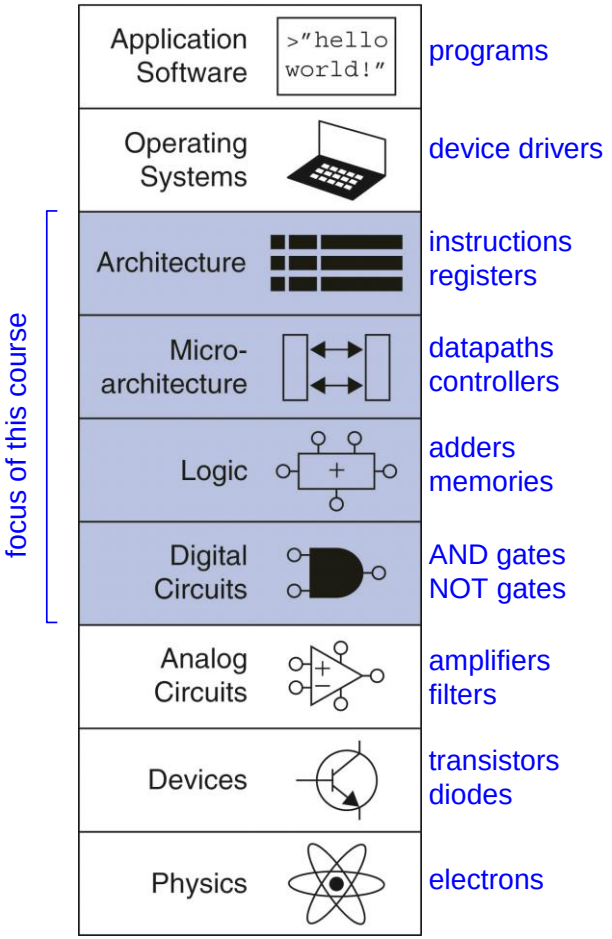
Instructions: commands in a computer's language

  - **Assembly language:** human-readable format of instructions
  - **Machine language:** computer-readable format (1's and 0's)

- **Microarchitecture:** Hardware view how to implement an architecture in hardware

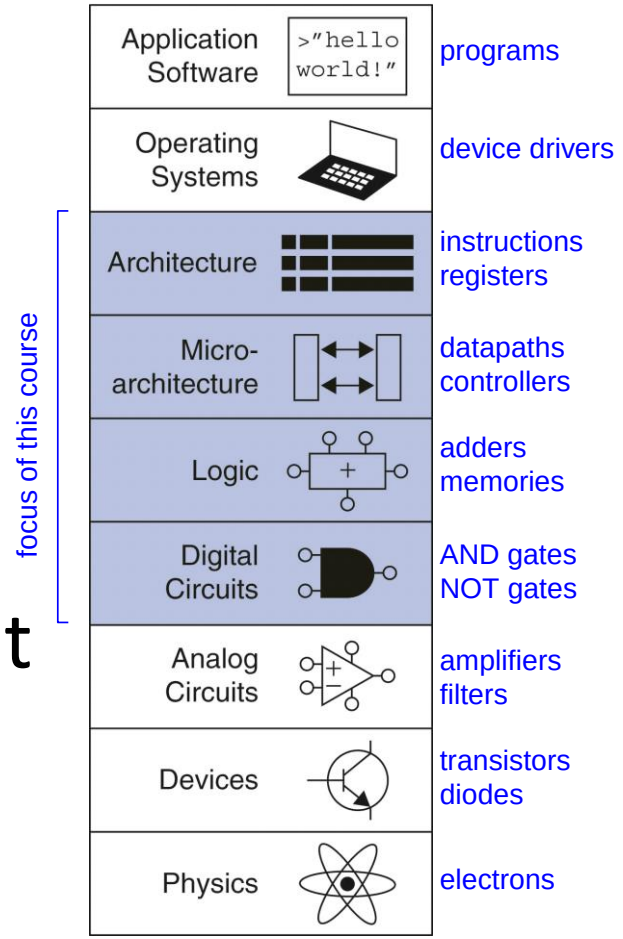
Processor:

  - **Datapath:** **functional blocks**
  - **Control:** **control signals**



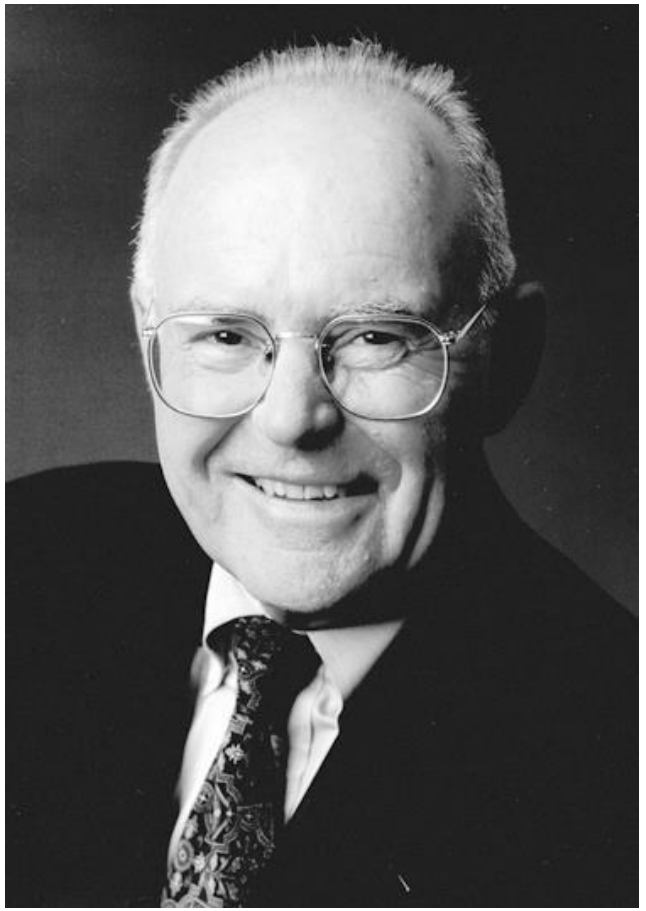
Digital Discipline: Binary Values

- **Two discrete values:**
  - 1's and 0's
  - 1, TRUE, HIGH
  - 0, FALSE, LOW
- **1 and 0:** voltage levels, rotating gears, fluid levels, etc.
- Digital circuits use **voltage** levels to represent 1 and 0
- ***Bit*****:** *B*inary dig*it*

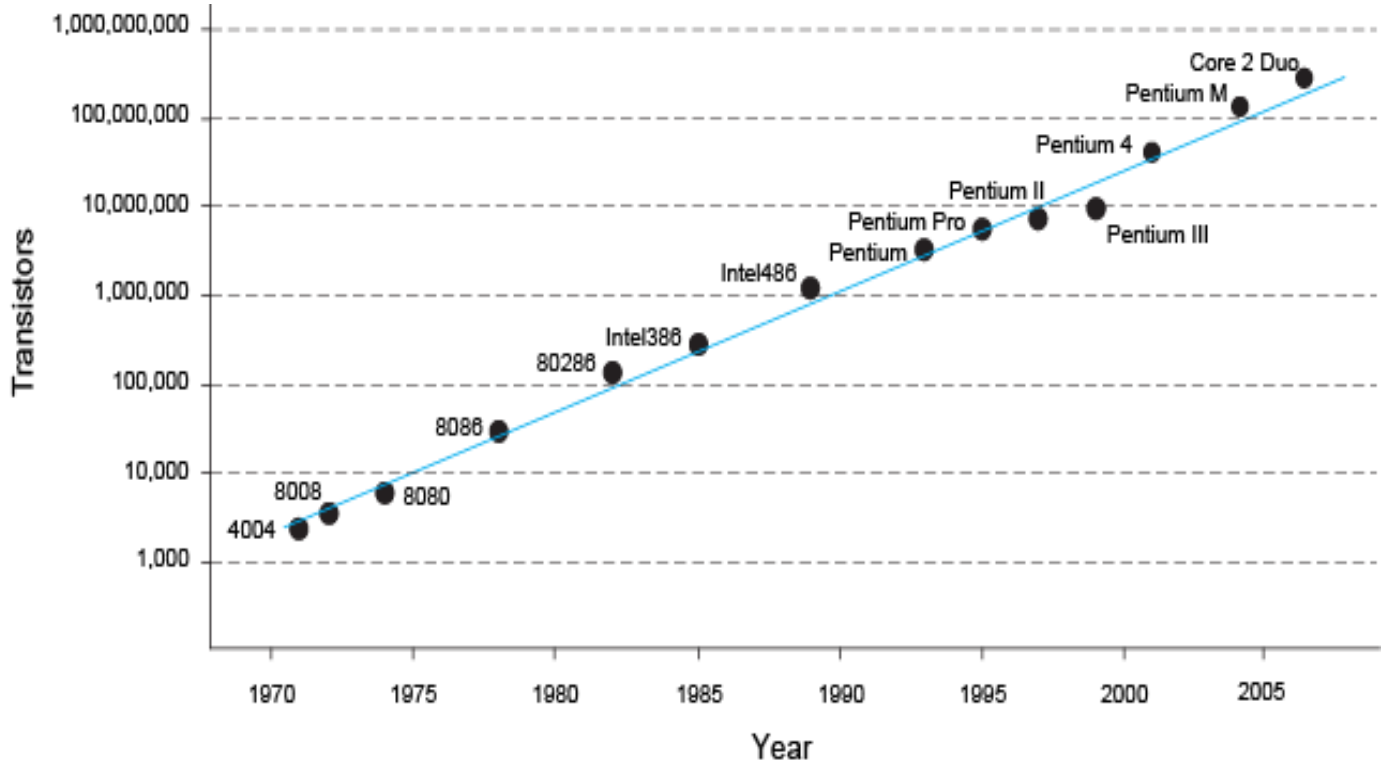


Gordon Moore, 1929-

- Cofounded Intel in 1968 with Robert Noyce.
- **Moore's Law:** number of transistors on a computer chip doubles every year (observed in 1965)
- Since 1975, transistor counts have doubled every two years.



Moore's Law



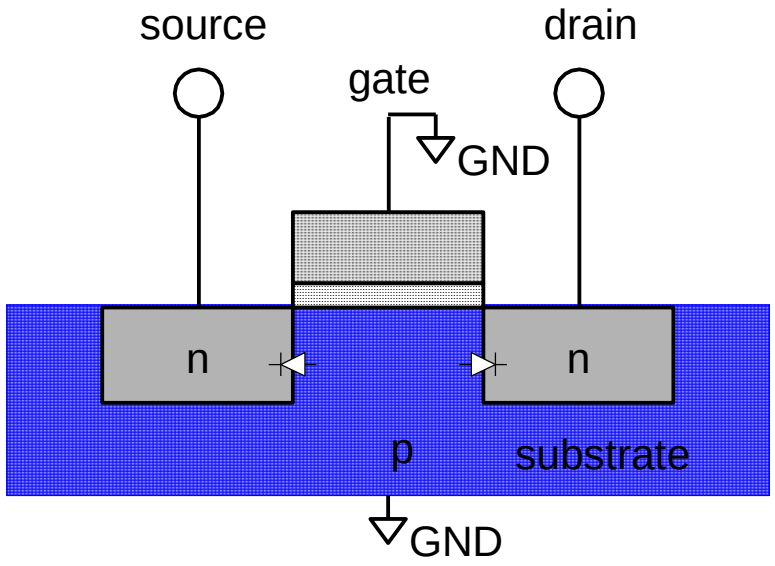
- *"If the automobile had followed the same development cycle as the computer, a Rolls-Royce would today cost $100, get one million miles to the gallon, and explode once a year . . ."*

  – Robert Cringley

Transistors: nMOS

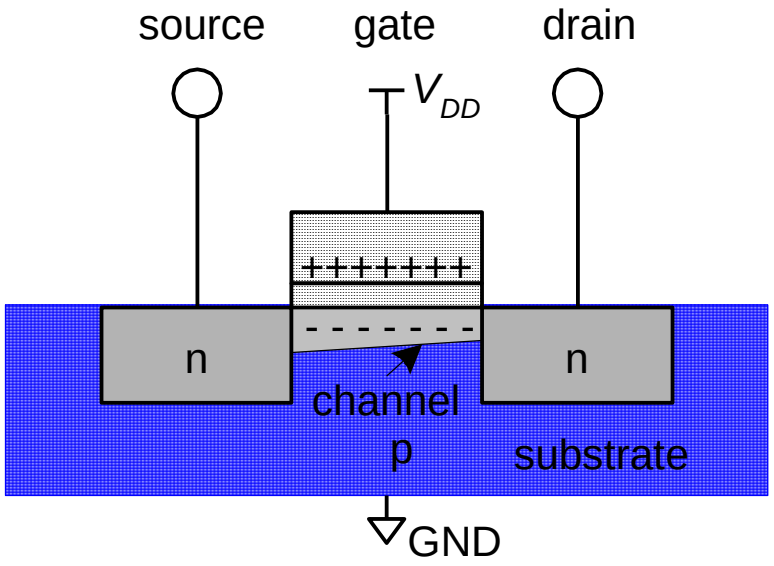
Gate = 0

OFF (no connection between source and drain)

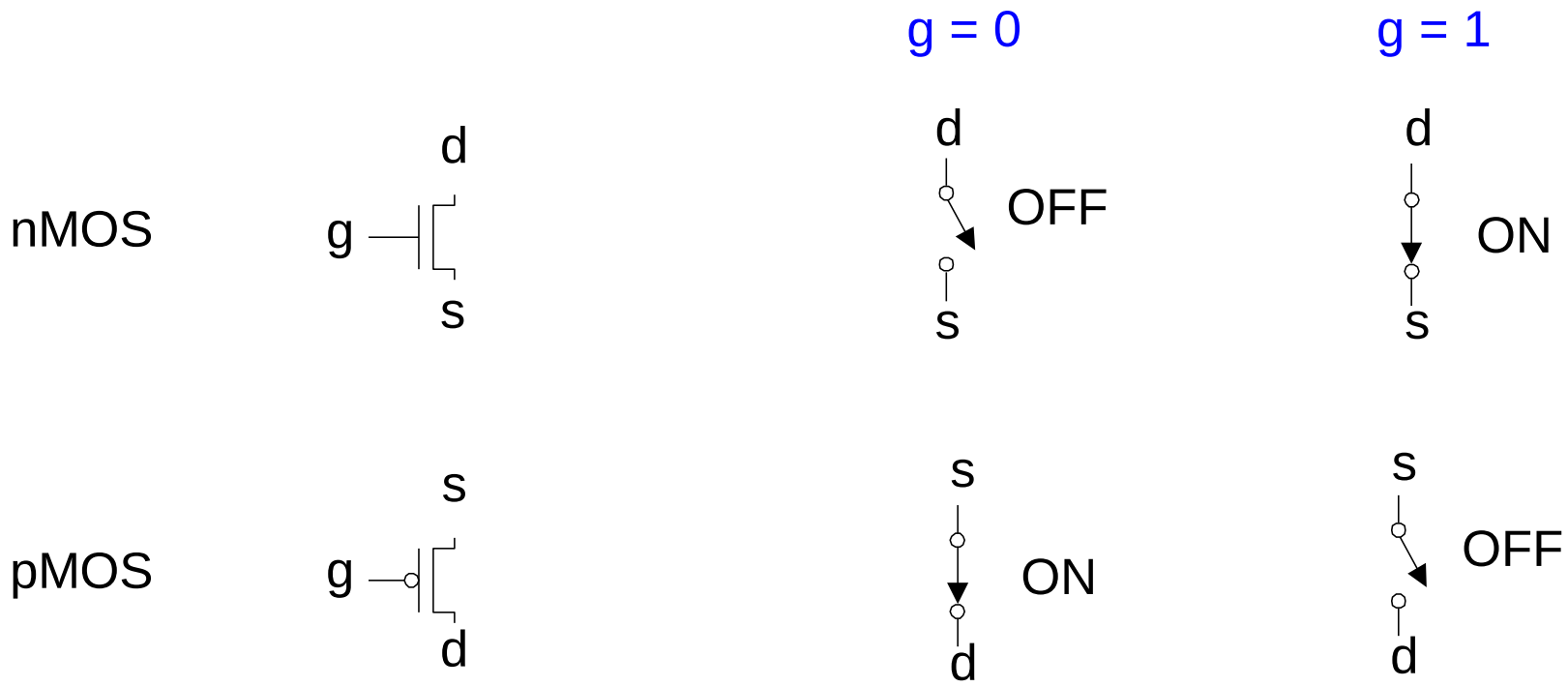


Gate = 1

ON (channel between source and drain)

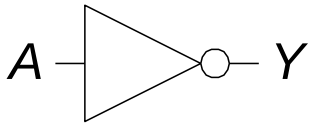


Transistor Function



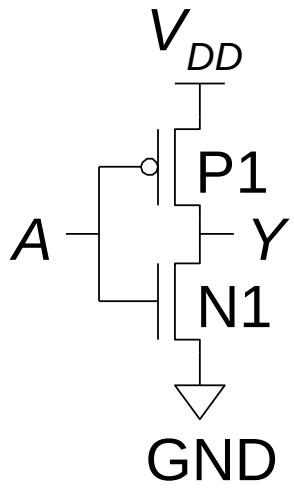
CMOS Gates: NOT Gate

NOT



$Y = \overline{A}$

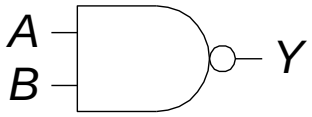
A	Y
0	1
1	0



A	P1	N1	Y
0	ON	OFF	1
1	OFF	ON	0

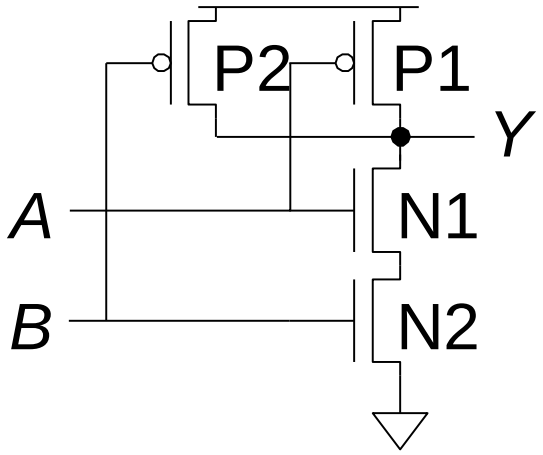
CMOS Gates: NAND Gate

NAND



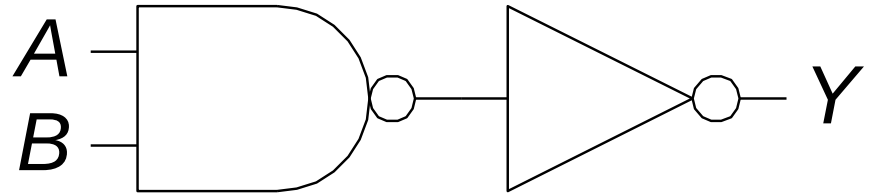
$Y = \overline{AB}$

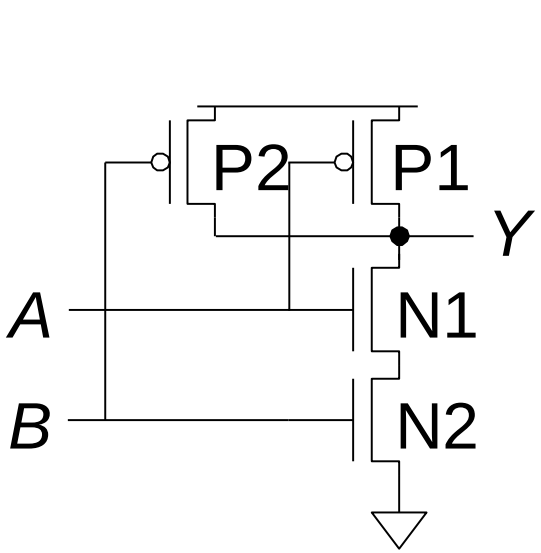
A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

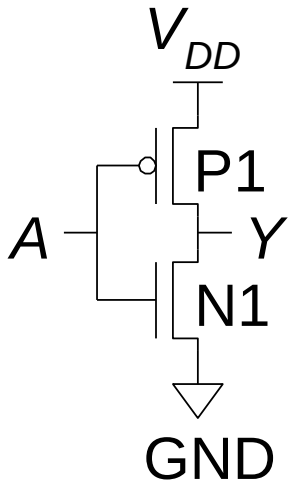


A	B	P1	P2	N1	N2	Y
0	0	ON	ON	OFF	OFF	1
0	1	ON	OFF	OFF	ON	1
1	0	OFF	ON	ON	OFF	1
1	1	OFF	OFF	ON	ON	0

AND Gate

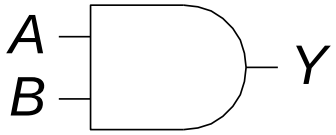






Two-Input Logic Gates

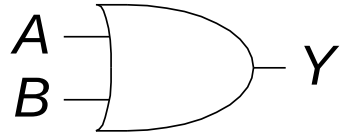
AND



$Y = AB$

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

OR

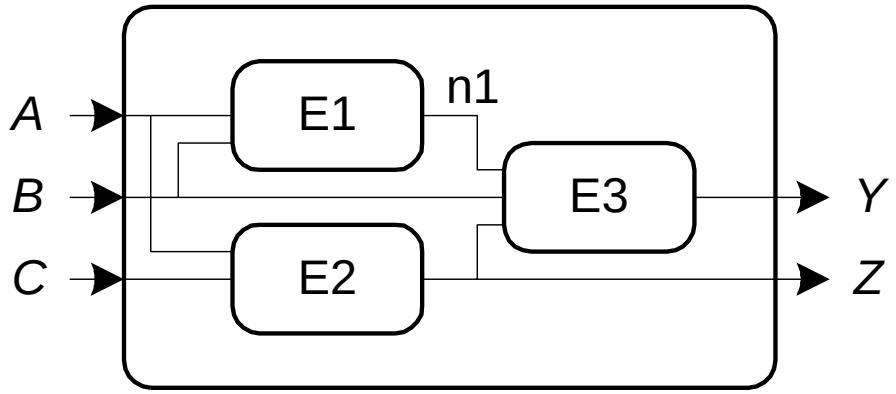


$Y = A + B$

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

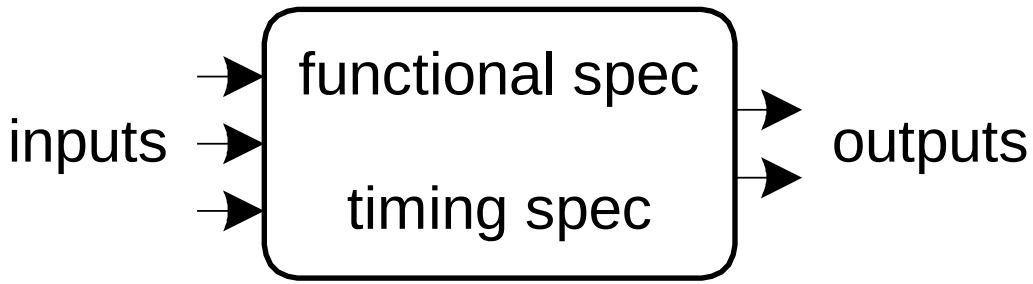
Circuits

- Nodes
  - Inputs: *A*, *B*, *C*
  - Outputs: *Y*, *Z*
  - Internal: n1
- Circuit elements
  - E1, E2, E3
  - Each a circuit



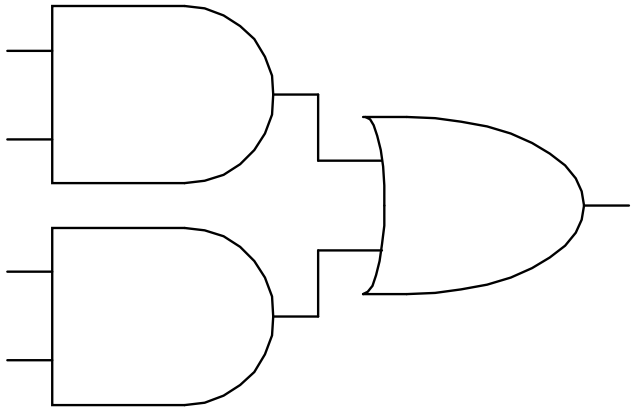
Types of Logic Circuits

- **Combinational Logic**
  - Memoryless
  - Outputs determined by current values of inputs
- **Sequential Logic**
  - Has memory
  - Outputs determined by previous and current values of inputs

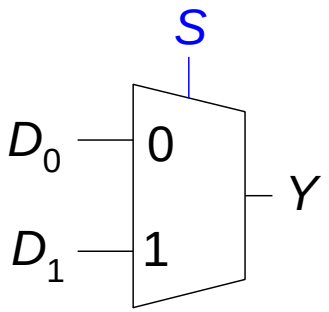


Combinational Composition

- Every element is combinational
- Every node is either an input or connects to *exactly one* output
- The circuit contains no cyclic paths
- **Example:**

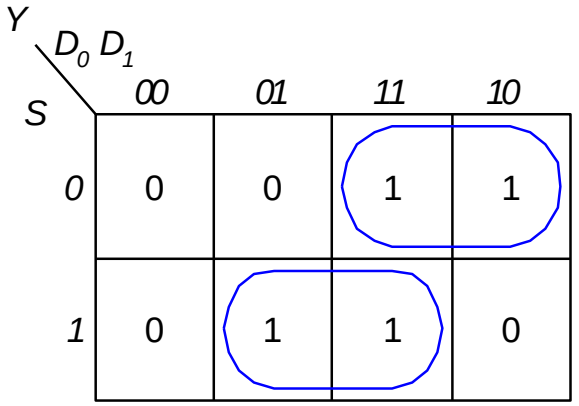


Multiplexer -> Control Unit

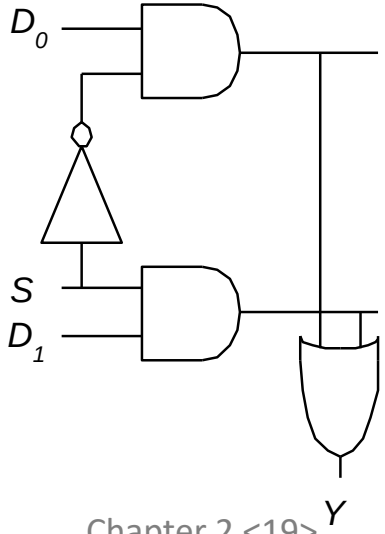


S	D_1	D_0	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

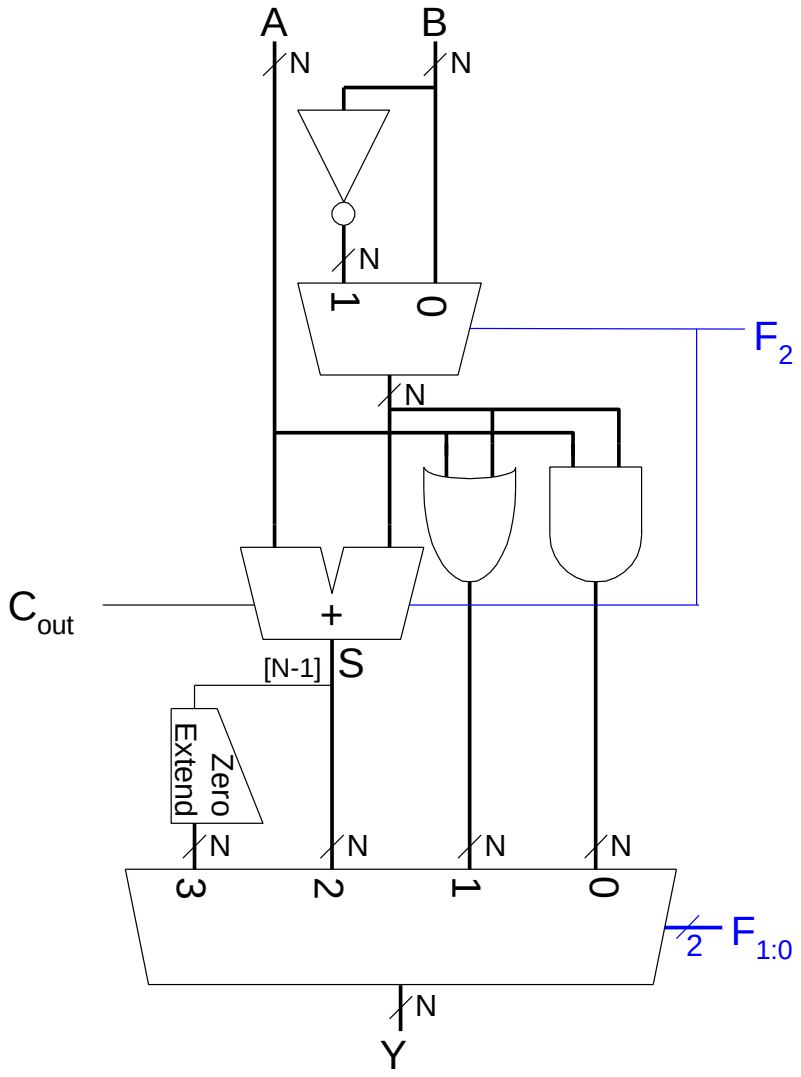
S	Y
0	D_0
1	D_1



$$Y = D_0\overline{S} + D_1S$$



ALU Design

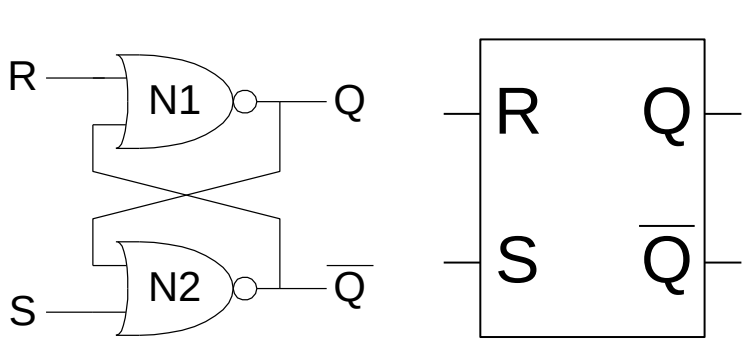


$F_{2:0}$	Function
000	A & B
001	A | B
010	A + B
011	not used
100	A & ~B
101	A | ~B
110	A - B
111	SLT

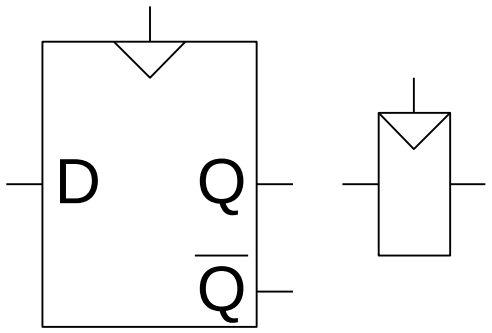
Sequential Circuits

- Give sequence to events
- Have memory (short-term)
- Use feedback from output to input to store information

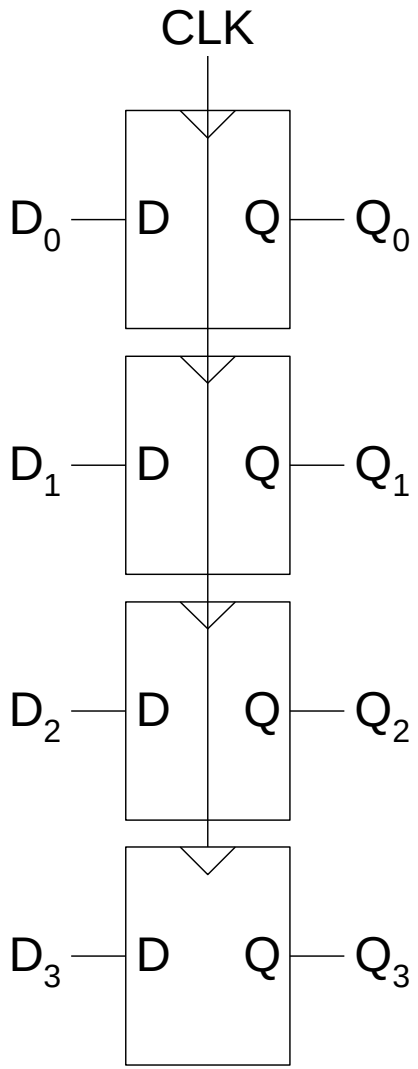
SR Latch Symbol

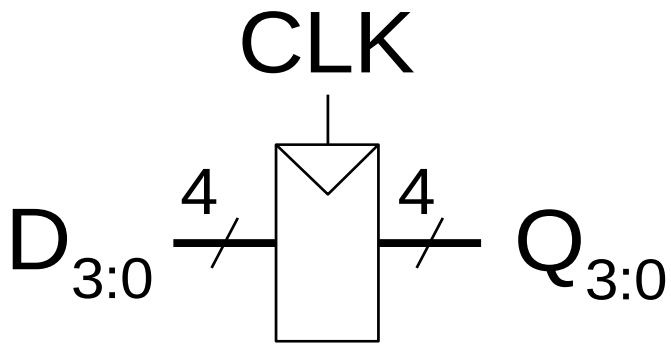


D Flip-Flop Symbols



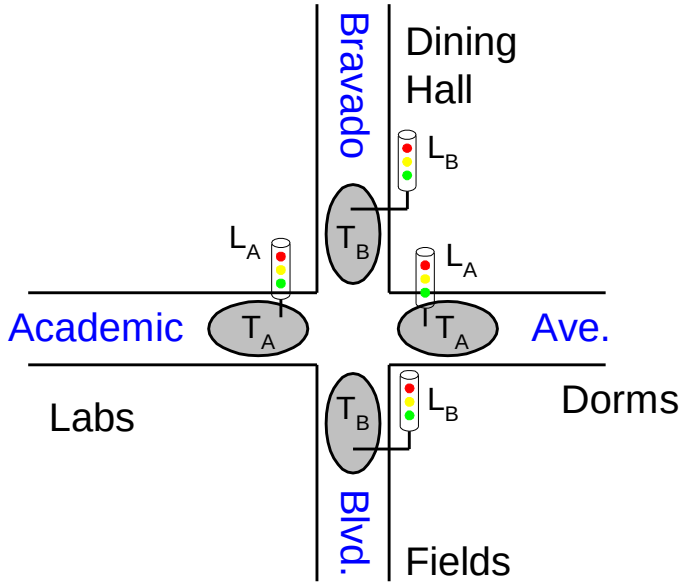
Registers -> Register File

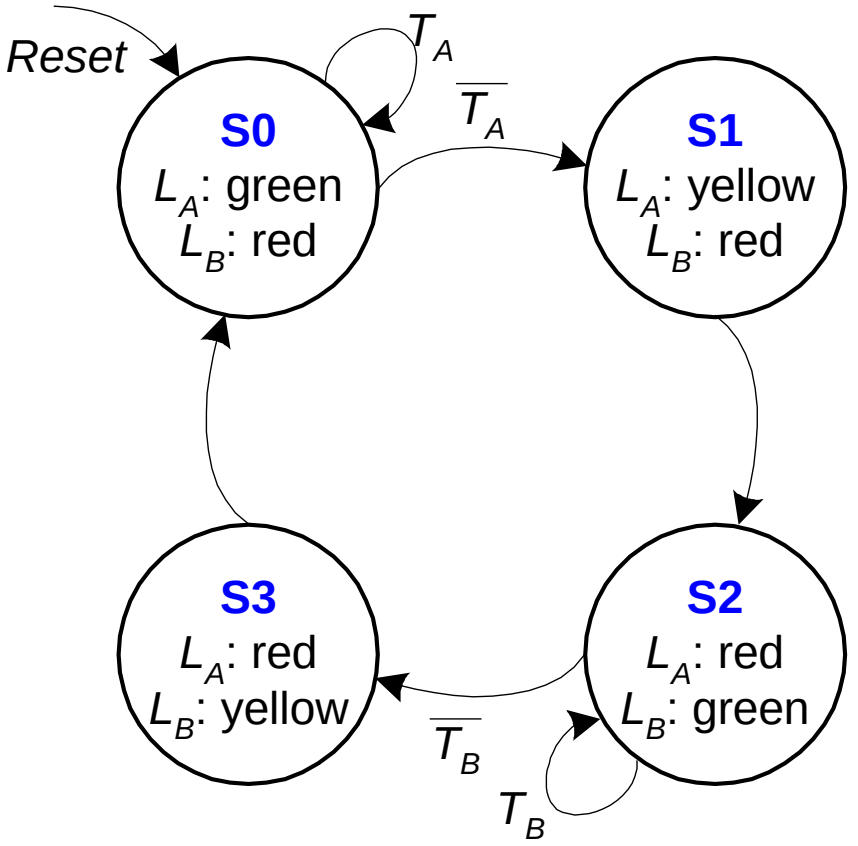




FSM State Transition -> Pipeline Processor

- **Moore FSM:** outputs labeled in each state
- **States:** Circles
- **Transitions:** Arcs

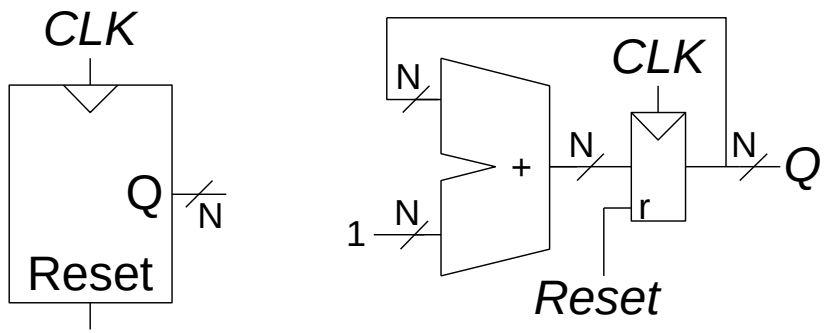




Counters -> PC Counter

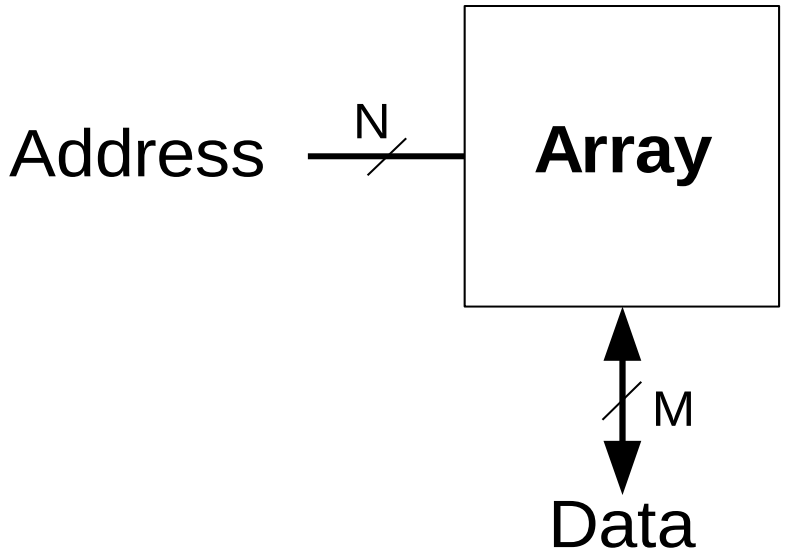
- Increments on each clock edge
- Used to cycle through numbers. For example,
  - 000, 001, 010, 011, 100, 101, 110, 111, 000, 001…
- Example uses:
  - Digital clock displays
  - Program counter: keeps track of current instruction executing

Symbol **Implementation**



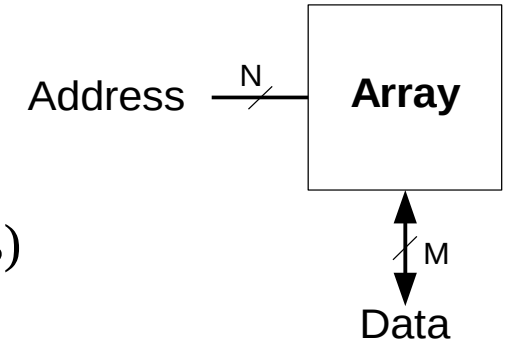
Memory Arrays -> Instruction / Data Memory

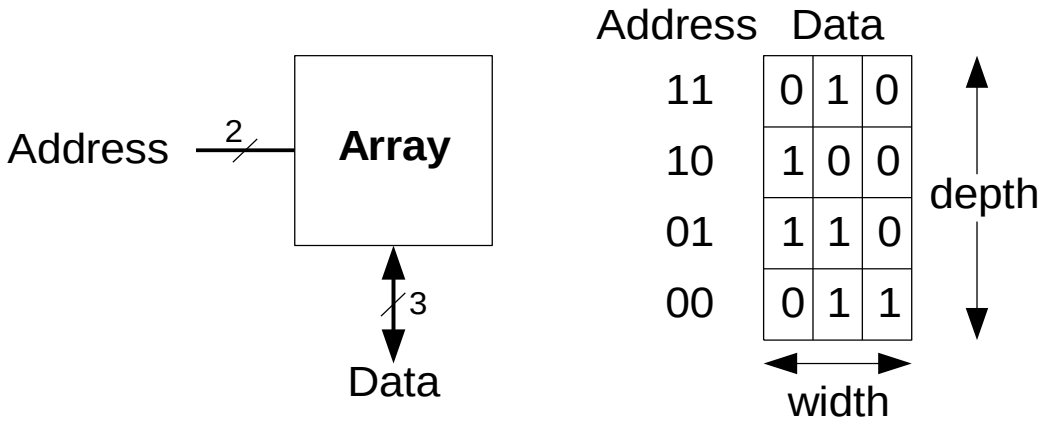
- Efficiently store large amounts of data
- 3 common types:
  - Dynamic random access memory (DRAM)
  - Static random access memory (SRAM)
  - Read only memory (ROM)
- *M*-bit data value read/ written at each unique *N*-bit address



Memory Arrays

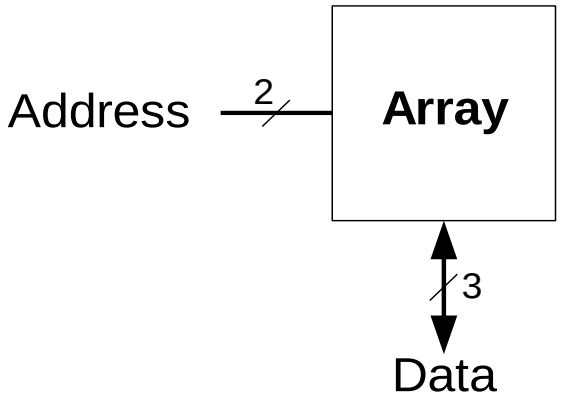
- 2-dimensional array of bit cells
- Each bit cell stores one bit
- N address bits and M data bits:
  - 2^N rows and M columns
  - **Depth:** number of rows (number of words)
  - **Width:** number of columns (size of word)
  - **Array size:** depth × width = $2^N \times M$

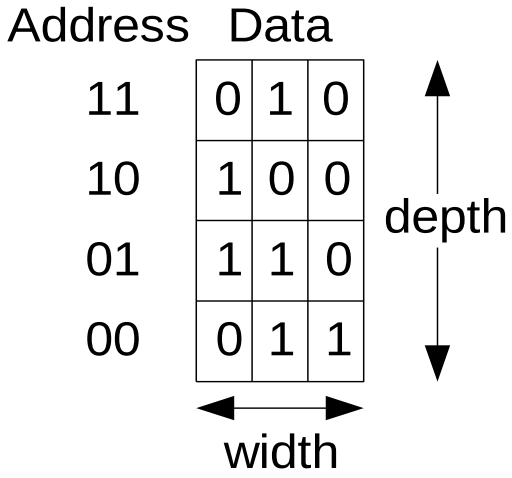


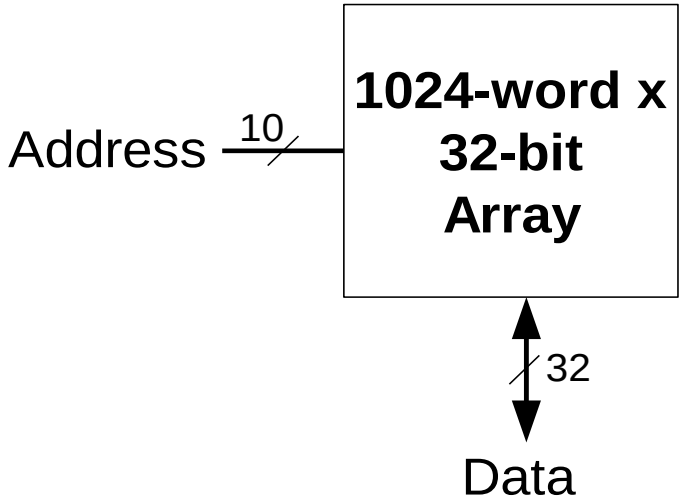


Memory Array Example

- $2^2 \times 3$-bit array
- Number of words: 4
- Word size: 3-bits
- For example, the 3-bit word stored at address 10 is 100

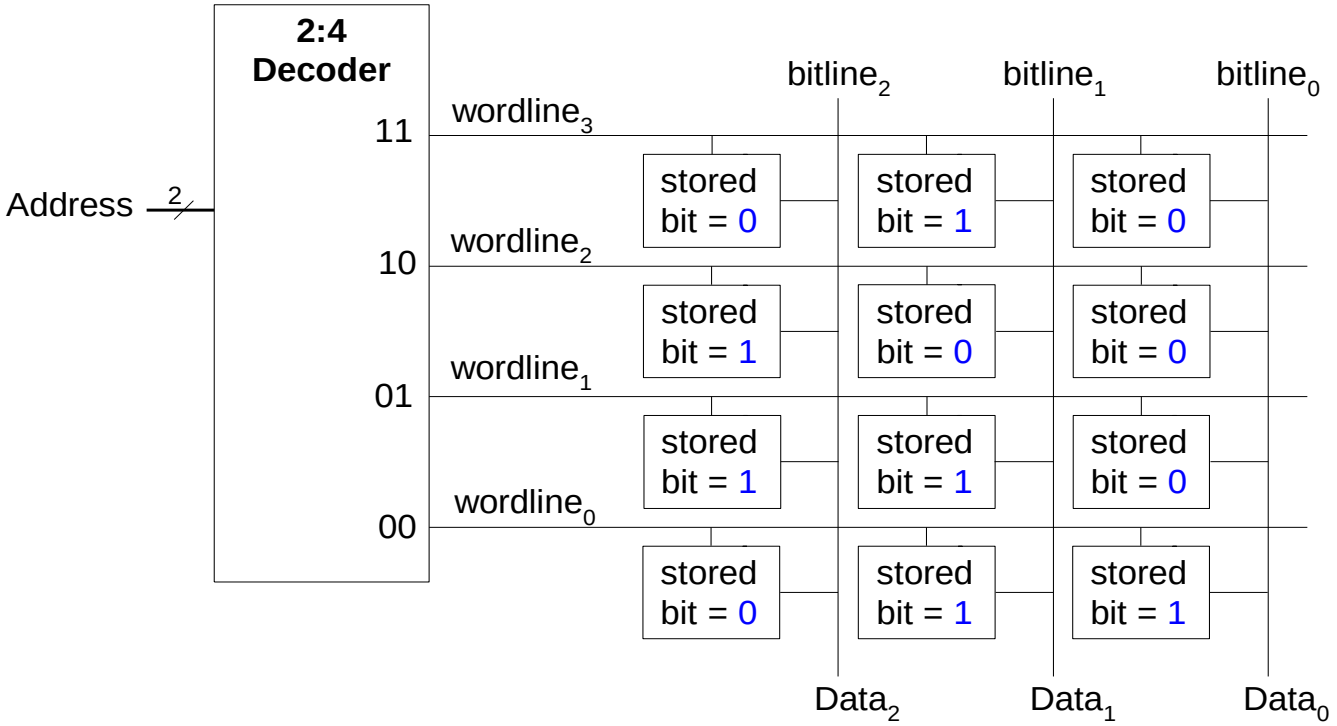




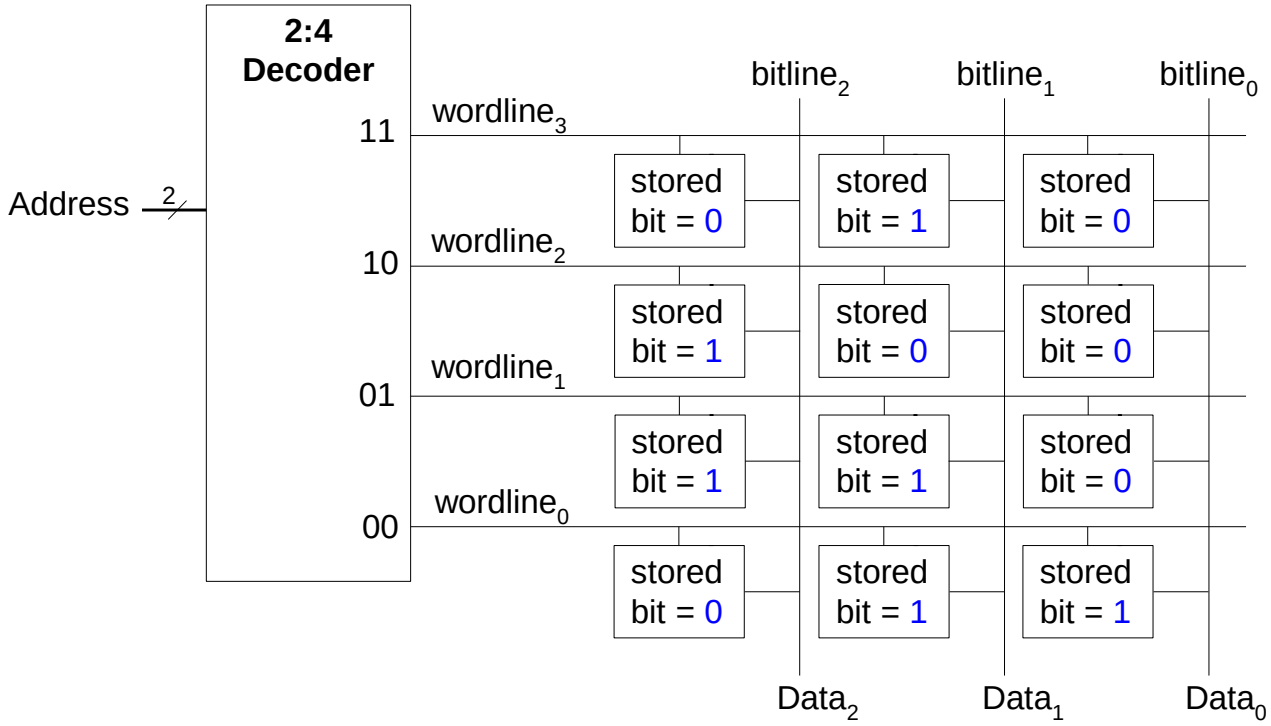


Memory Array

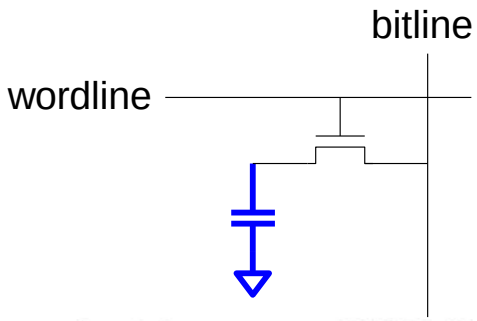
- **Wordline:**
  - like an enable
  - single row in memory array read/written
  - corresponds to unique address
  - only one wordline HIGH at once



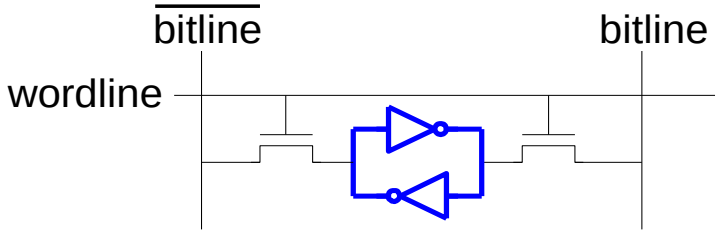
Memory Arrays Types



DRAM bit cell:



SRAM bit cell:



Processor

- **Architecture: programmer's view** of computer

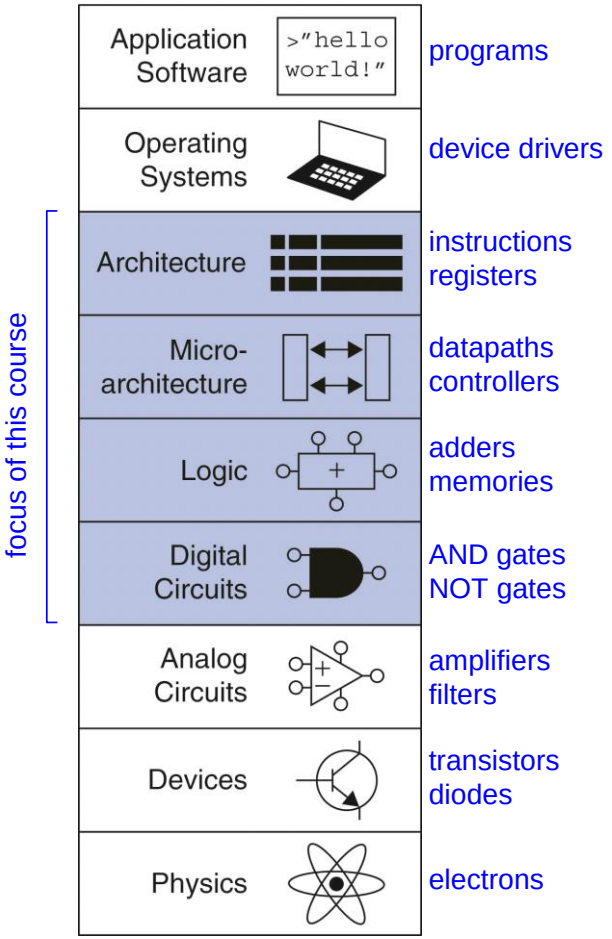
Instructions: commands in a computer's language

  - **Assembly language:** human-readable format of instructions
  - **Machine language:** computer-readable format (1's and 0's)

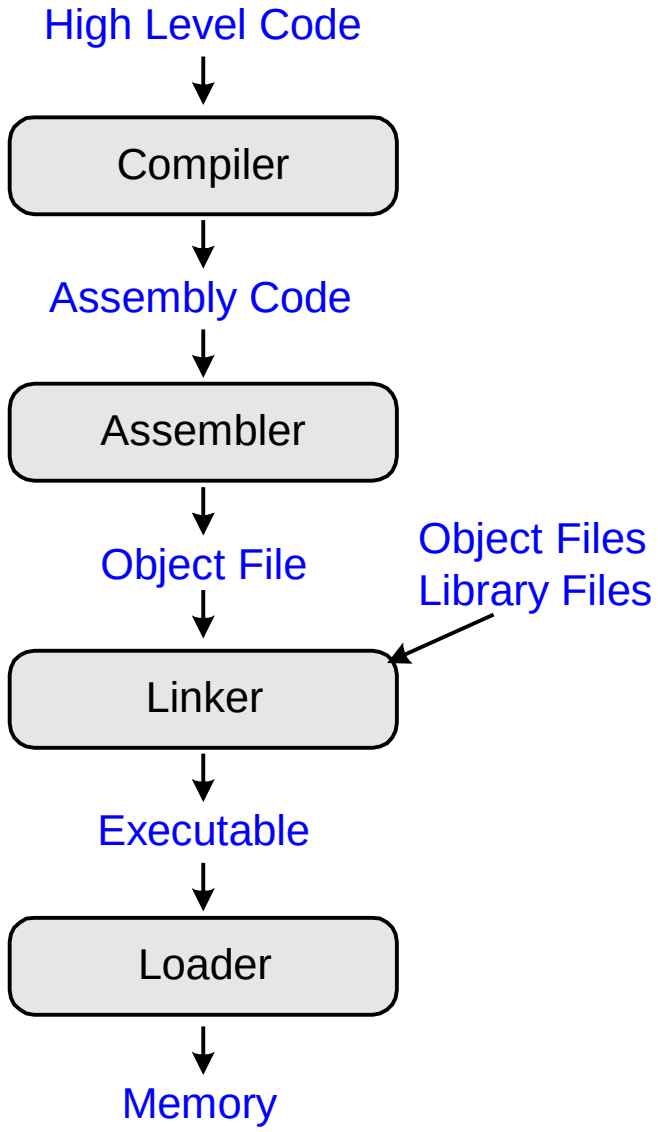
- **Microarchitecture: Hardware view** how to implement an architecture in hardware

Processor:

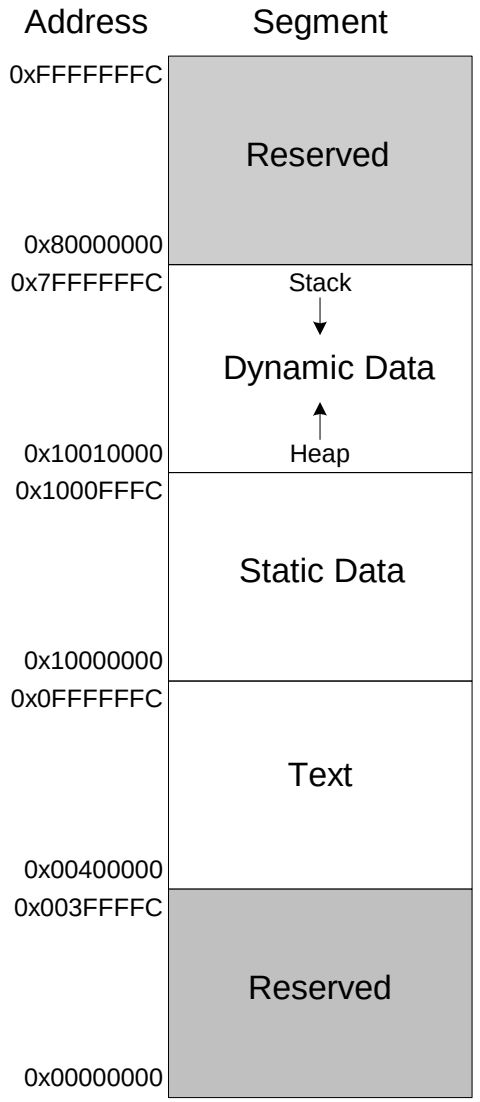
  - **HDL Language: Verilog.**
  - **Datapath: functional blocks**
  - **Control: control signals**



How to Compile & Run a Program



programmer's view : MIPS Memory Map



Machine Language

- Binary representation of instructions
- Computers only understand 1's and 0's
- 32-bit instructions
  - Simplicity favors regularity: 32-bit data & instructions
- 3 instruction formats:
  - **R-Type:** register operands
  - **I-Type:** immediate operand
  - **J-Type:** for jumping (discuss later)

MIPS Register Set

Name	Register Number	Usage
`$0`	0	the constant value 0
`$at`	1	assembler temporary
`$v0-$v1`	2-3	Function return values
`$a0-$a3`	4-7	Function arguments
`$t0-$t7`	8-15	temporaries
`$s0-$s7`	16-23	saved variables
`$t8-$t9`	24-25	more temporaries
`$k0-$k1`	26-27	OS temporaries
`$gp`	28	global pointer
`$sp`	29	stack pointer
`$fp`	30	frame pointer
`$ra`	31	Function return address

Machine Language: R-Type

- *Register-type*
- 3 register operands:
  - `rs`, `rt`: source registers
  - `rd`: destination register
- Other fields:
  - `op`: the *operation code* or *opcode* (0 for R-type instructions)
  - `funct`: the *function* with opcode, tells computer what operation to perform
  - `shamt`: the *shift amount* for shift instructions, otherwise it's 0

R-Type

R-Type Examples

Assembly Code

```
add $s0, $s1, $s2
sub $t0, $t3, $t5
```

Field Values

op	rs	rt	rd	shamt	funct
0	17	18	16	0	32
0	11	13	8	0	34
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Machine Code

op	rs	rt	rd	shamt	funct	
000000	10001	10010	10000	00000	100000	(0x02328020)
000000	01011	01101	01000	00000	100010	(0x016D4022)
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

Note the order of registers in the assembly code:

```
add rd, rs, rt
```

Machine Language: I-Type

- *Immediate-type*
- 3 operands:
  - `rs`, `rt`: register operands
  - `imm`: 16-bit two's complement immediate
- Other fields:
  - `op`: the opcode
  - Simplicity favors regularity: all instructions have opcode
  - Operation is completely determined by opcode

I-Type

I-Type Examples

Assembly Code	op	rs	rt	imm
`addi $s0, $s1, 5`	8	17	16	5
`addi $t0, $s3, -12`	8	19	8	-12
`lw $t2, 32($0)`	35	0	10	32
`sw $s1, 4($t1)`	43	9	17	4
	6 bits	5 bits	5 bits	16 bits

Note the differing order of registers in assembly and machine codes:

```
addi rt, rs, imm
lw   rt, imm(rs)
sw   rt, imm(rs)
```

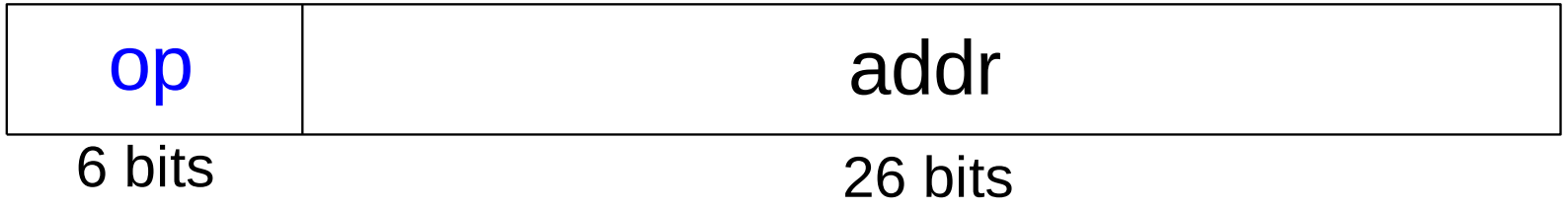
Machine Code

op	rs	rt	imm	
001000	10001	10000	0000 0000 0000 0101	(0x22300005)
001000	10011	01000	1111 1111 1111 0100	(0x2268FFF4)
100011	00000	01010	0000 0000 0010 0000	(0x8C0A0020)
101011	01001	10001	0000 0000 0000 0100	(0xAD310004)
6 bits	5 bits	5 bits	16 bits	

Machine Language: J-Type

- *Jump-type*
- 26-bit address operand (`addr`)
- Used for jump instructions (`j`)

J-Type



J-Type Examples

Pseudo-direct Addressing

```
0x0040005C          jal   sum
...
0x004000A0  sum:    add   $v0, $a0, $a1
```

JTA 0000 0000 0100 0000 0000 0000 1010 0000 (0x004000A0)

26-bit addr 0000 0000 0100 0000 0000 0000 1010 0000 (0x0100028)

0 1 0 0 0 2 8

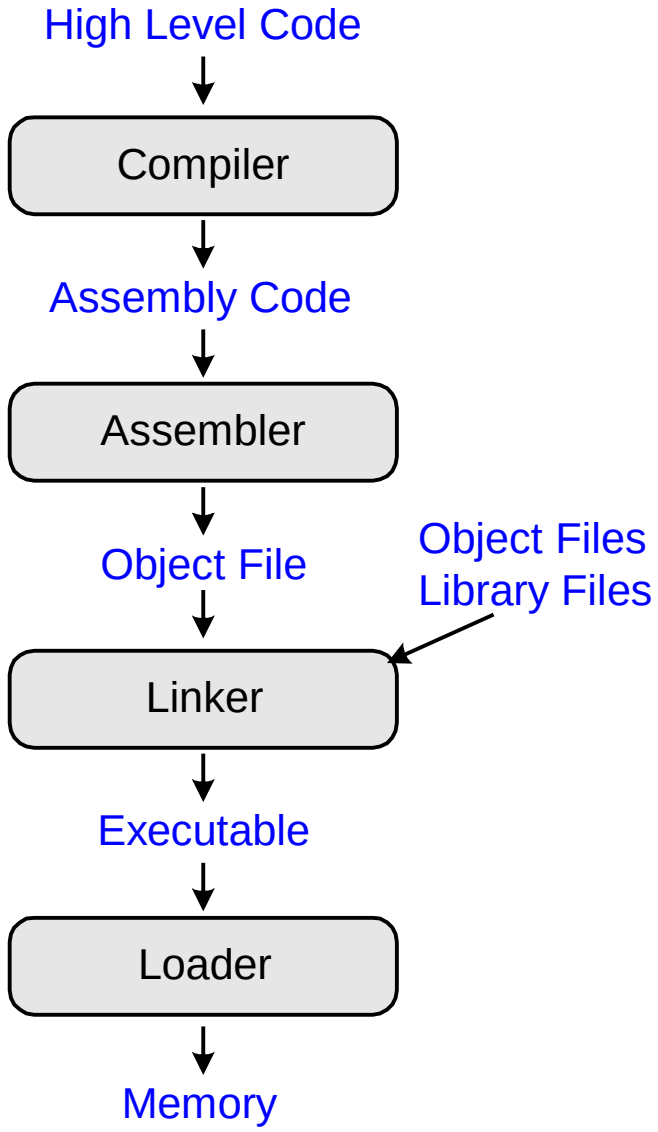
Field Values

op	imm
3	0x0100028
6 bits	26 bits

Machine Code

op	addr	
000011	00 0001 0000 0000 0000 0010 1000	(0x0C100028)
6 bits	26 bits	

How to Compile & Run a Program



Example Program: MIPS Assembly

```
int f, g, y;  // global

int main(void)
{

  f = 2;
  g = 3;

  y = sum(f, g);
  return y;
}

int sum(int a, int b) {
  return (a + b);
}
```

```
.data
f:
g:
y:
.text
main:
  addi $sp, $sp, -4    # stack frame
  sw   $ra, 0($sp)     # store $ra
  addi $a0, $0, 2      # $a0 = 2
  sw   $a0, f          # f = 2
  addi $a1, $0, 3      # $a1 = 3
  sw   $a1, g          # g = 3
  jal  sum             # call sum
  sw   $v0, y          # y = sum()
  lw   $ra, 0($sp)     # restore $ra
  addi $sp, $sp, 4     # restore $sp
  jr   $ra             # return to OS
sum:
  add  $v0, $a0, $a1   # $v0 = a + b
  jr   $ra             # return
```

Example Program: Executable

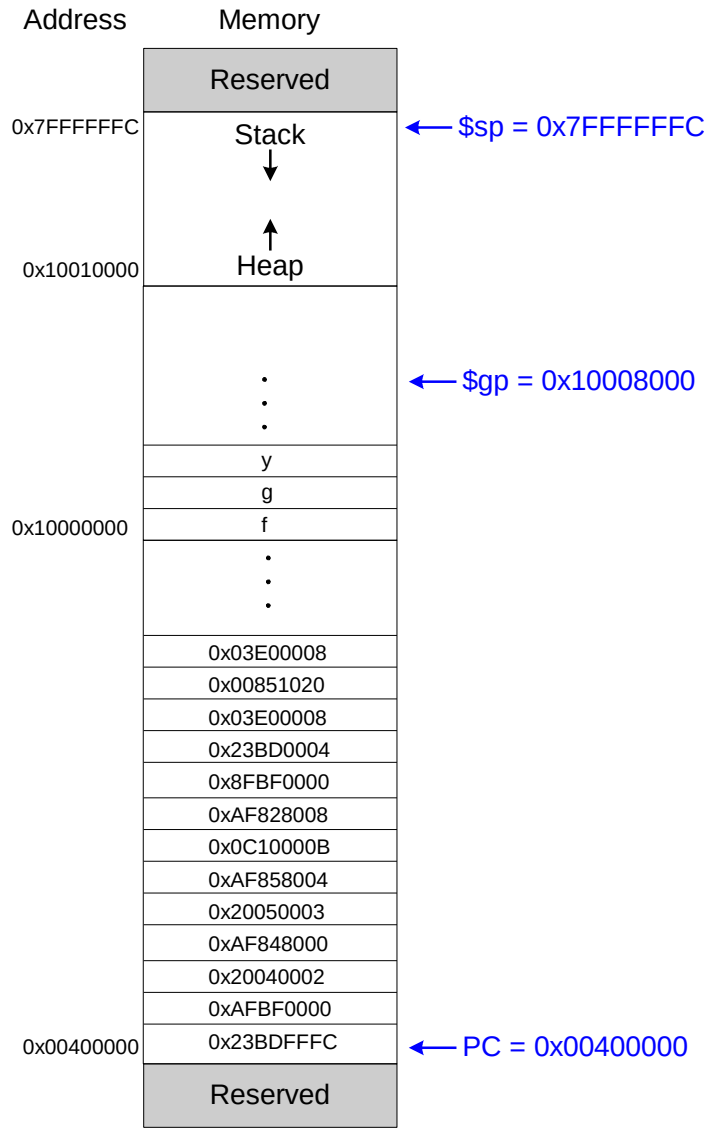
Executable file header	Text Size	Data Size
	0x34 (52 bytes)	0xC (12 bytes)

Text segment	Address	Instruction
	0x00400000	0x23BDFFFC
	0x00400004	0xAFBF0000
	0x00400008	0x20040002
	0x0040000C	0xAF848000
	0x00400010	0x20050003
	0x00400014	0xAF858004
	0x00400018	0x0C10000B
	0x0040001C	0xAF828008
	0x00400020	0x8FBF0000
	0x00400024	0x23BD0004
	0x00400028	0x03E00008
	0x0040002C	0x00851020
	0x00400030	0x03E00008

Data segment	Address	Data
	0x10000000	f
	0x10000004	g
	0x10000008	y

```
addi $sp, $sp, -4
sw   $ra, 0 ($sp)
addi $a0, $0, 2
sw   $a0, 0x8000 ($gp)
addi $a1, $0, 3
sw   $a1, 0x8004 ($gp)
jal  0x0040002C
sw   $v0, 0x8008 ($gp)
lw   $ra, 0 ($sp)
addi $sp, $sp, -4
jr   $ra
add  $v0, $a0, $a1
jr   $ra
```

Example Program: In Memory



Processor

- **Architecture: programmer's view** of computer

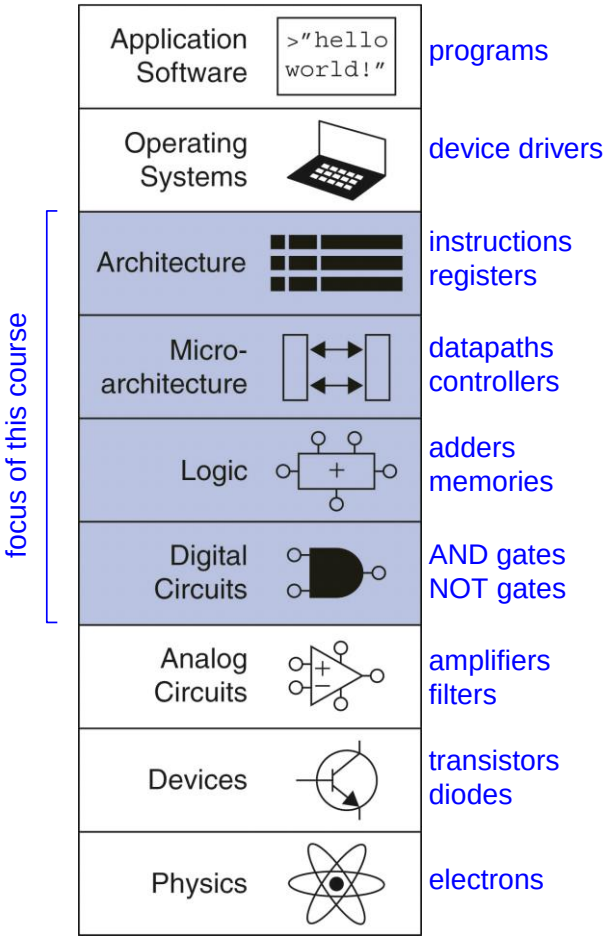
Instructions: commands in a computer's language

  - **Assembly language:** human-readable format of instructions
  - **Machine language:** computer-readable format (1's and 0's)

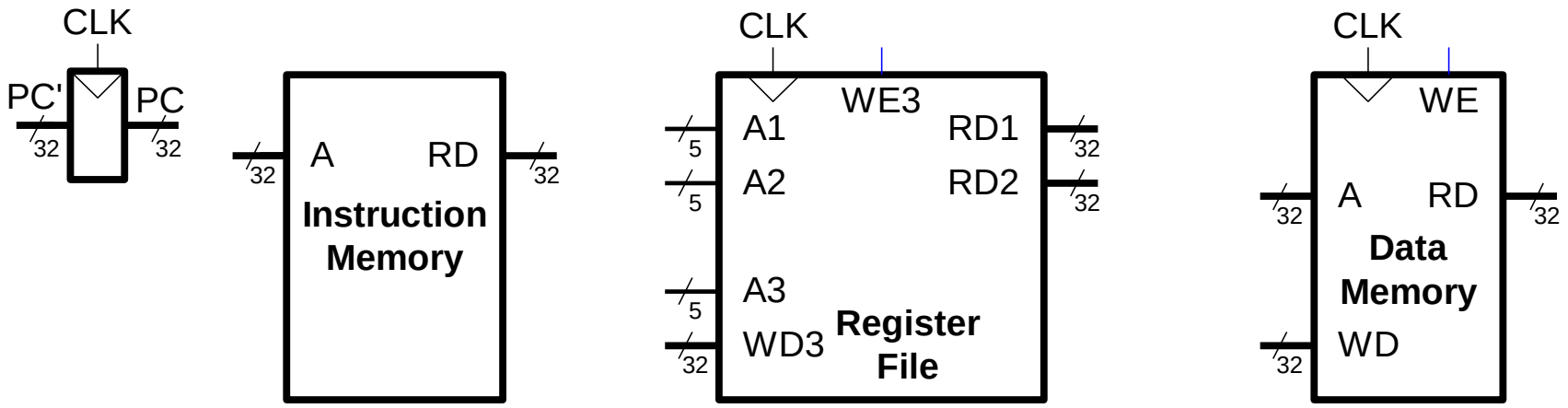
- **Microarchitecture: Hardware view** how to implement an architecture in hardware

Processor:

  - **HDL Language: Verilog.**
  - **Datapath: functional blocks**
  - **Control: control signals**

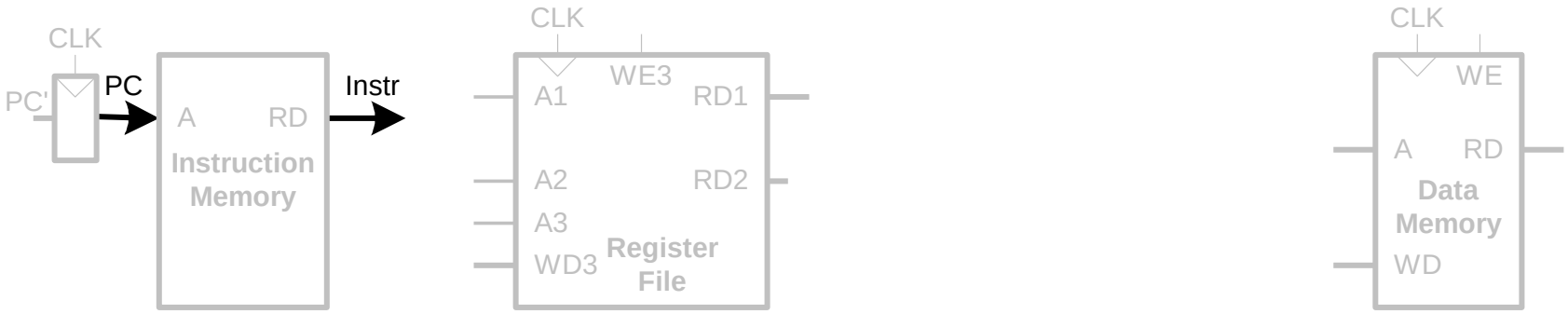


MIPS State Elements



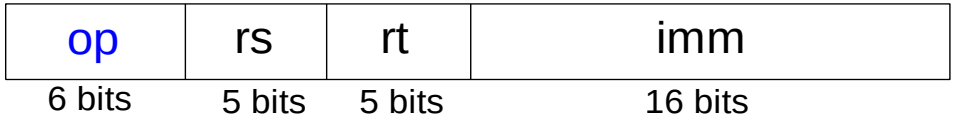
Single-Cycle Datapath: `lw` fetch

STEP 1: Fetch instruction



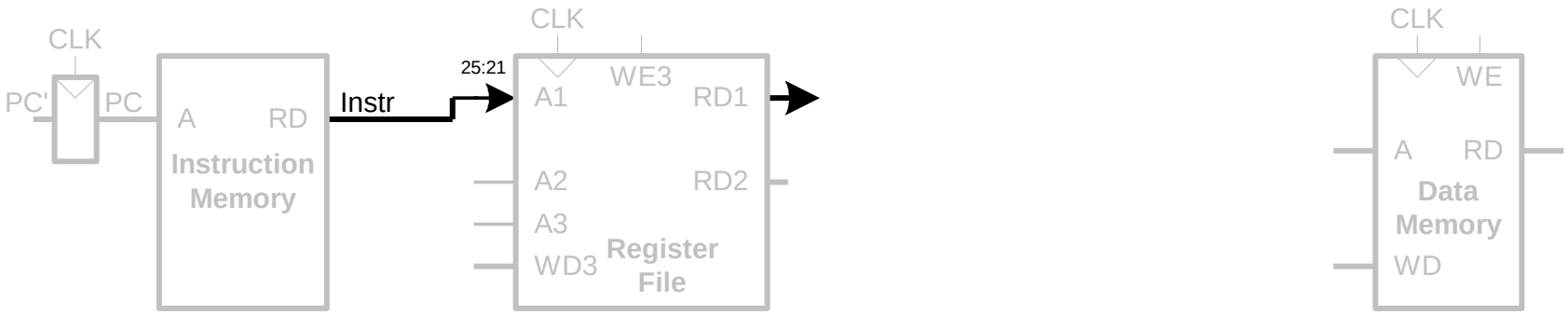
`lw rt, imm(rs)`

I-Type



Single-Cycle Datapath: `lw` Register Read

STEP 2: Read source operands from RF



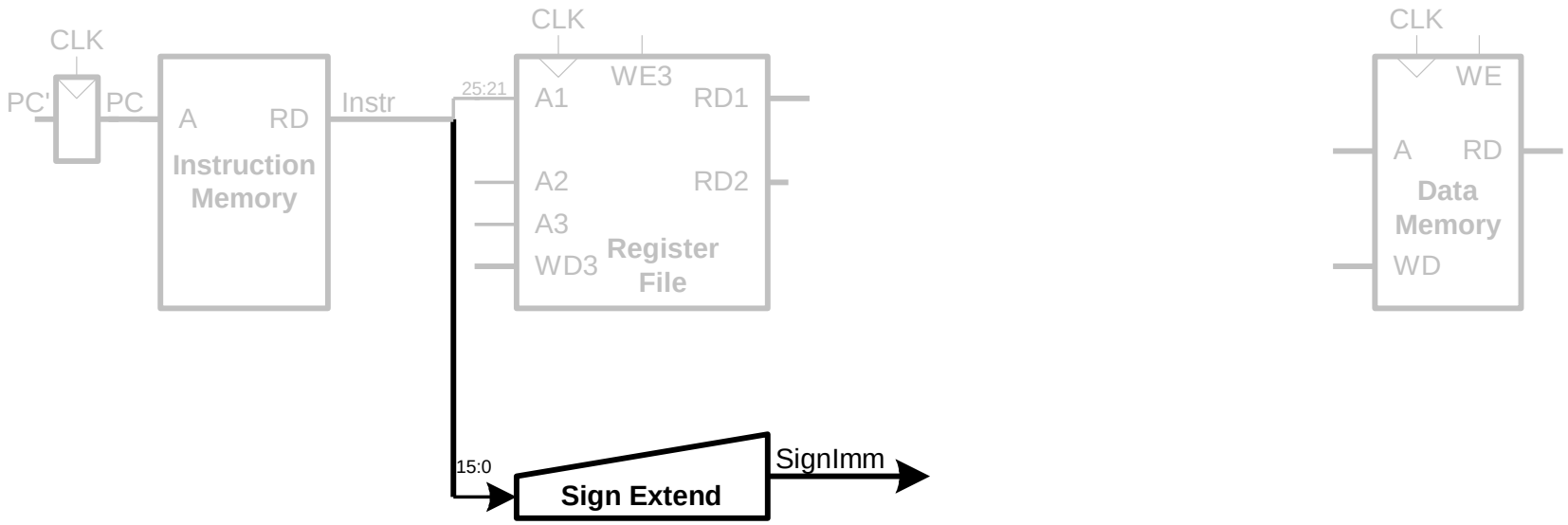
`lw rt, imm(rs)`

I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

Single-Cycle Datapath: `lw` Immediate

STEP 3: Sign-extend the immediate



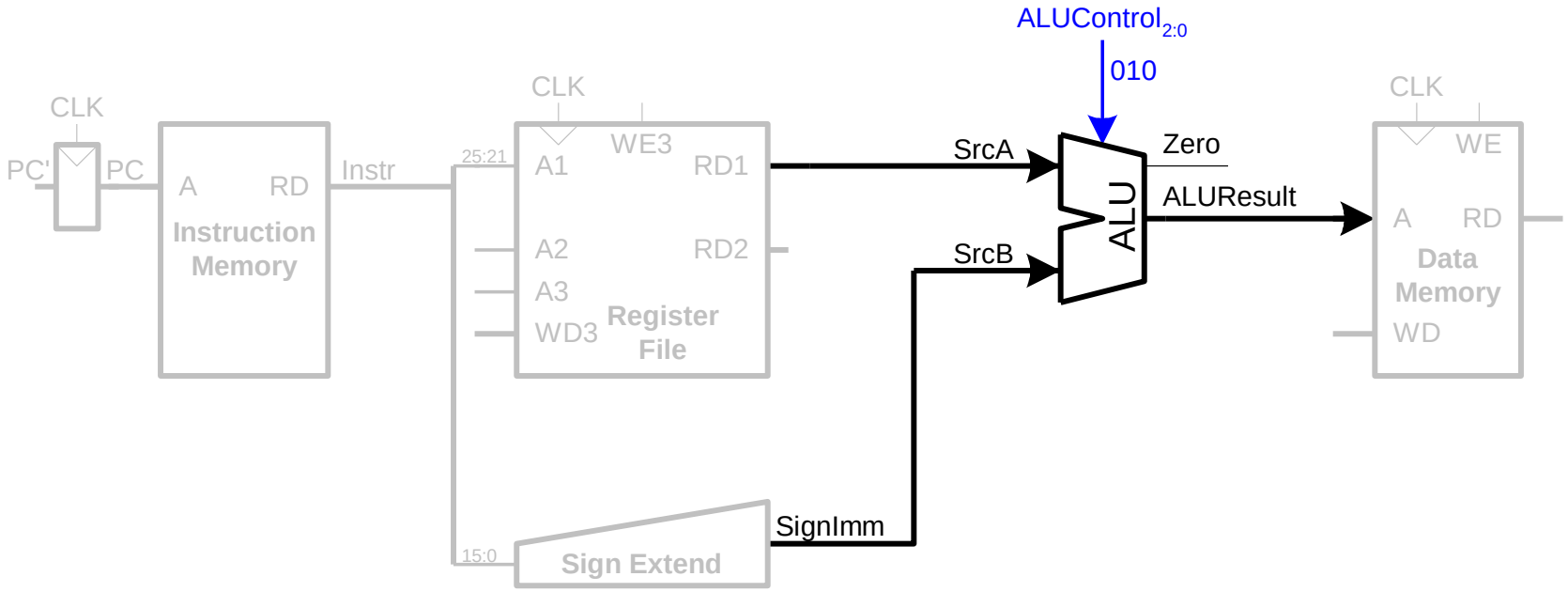
`lw rt, imm(rs)`

I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

Single-Cycle Datapath: `lw` address

STEP 4: Compute the memory address



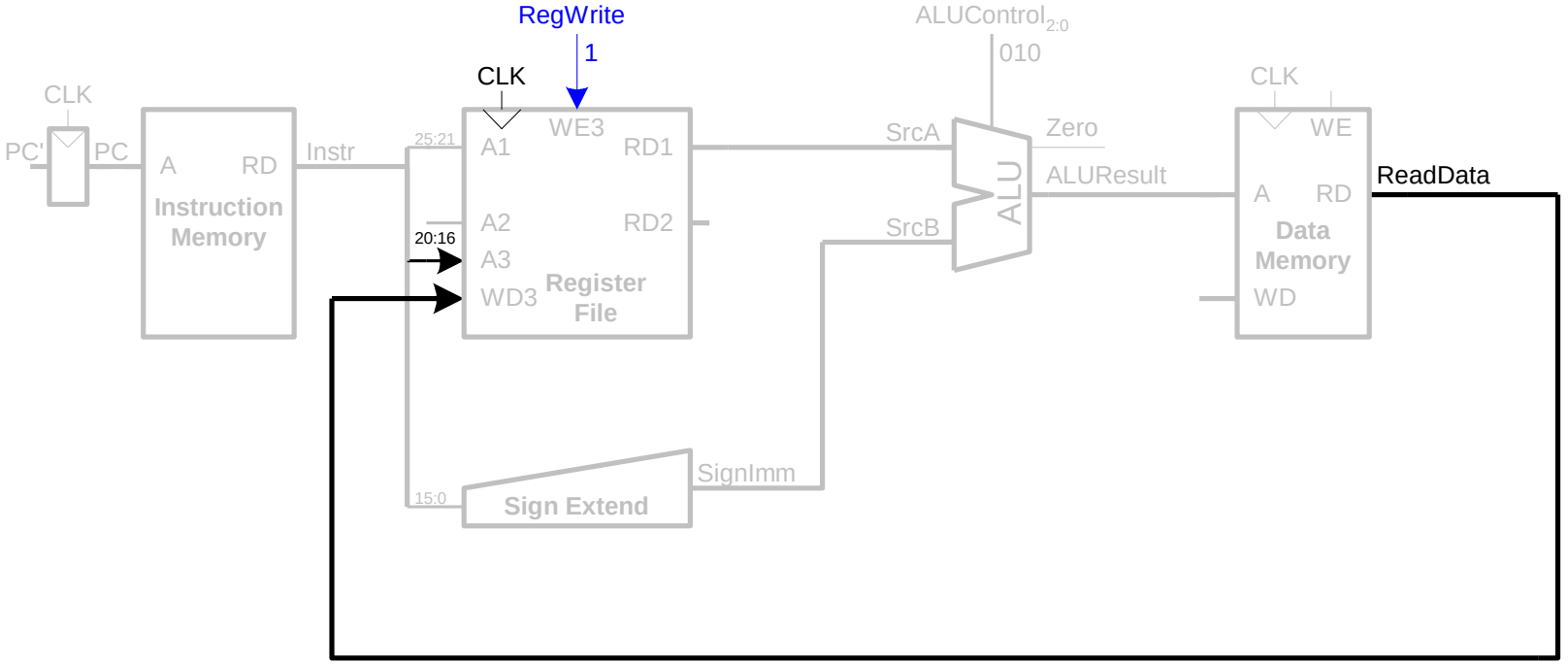
`lw rt, imm(rs)`

I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

Single-Cycle Datapath: `lw` Memory Read

- **STEP 5:** Read data from memory and write it back to register file



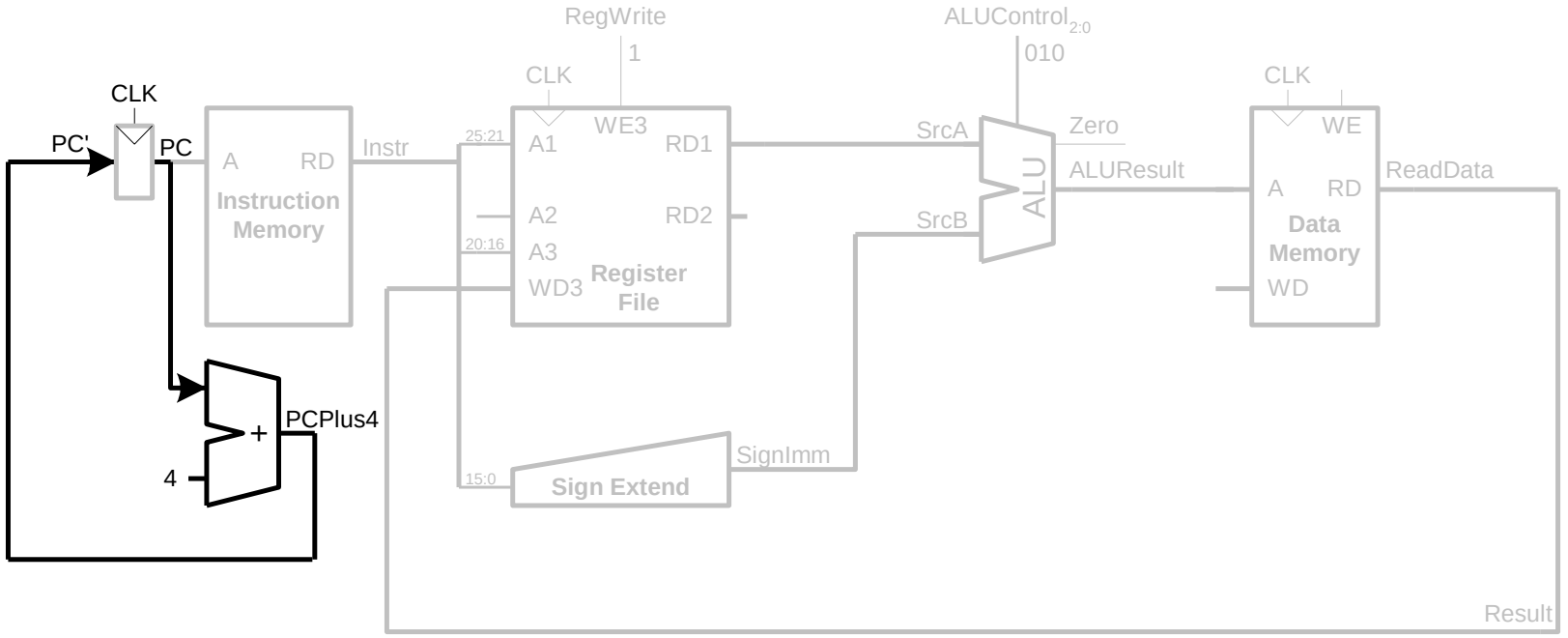
I-Type

`lw rt, imm(rs)`

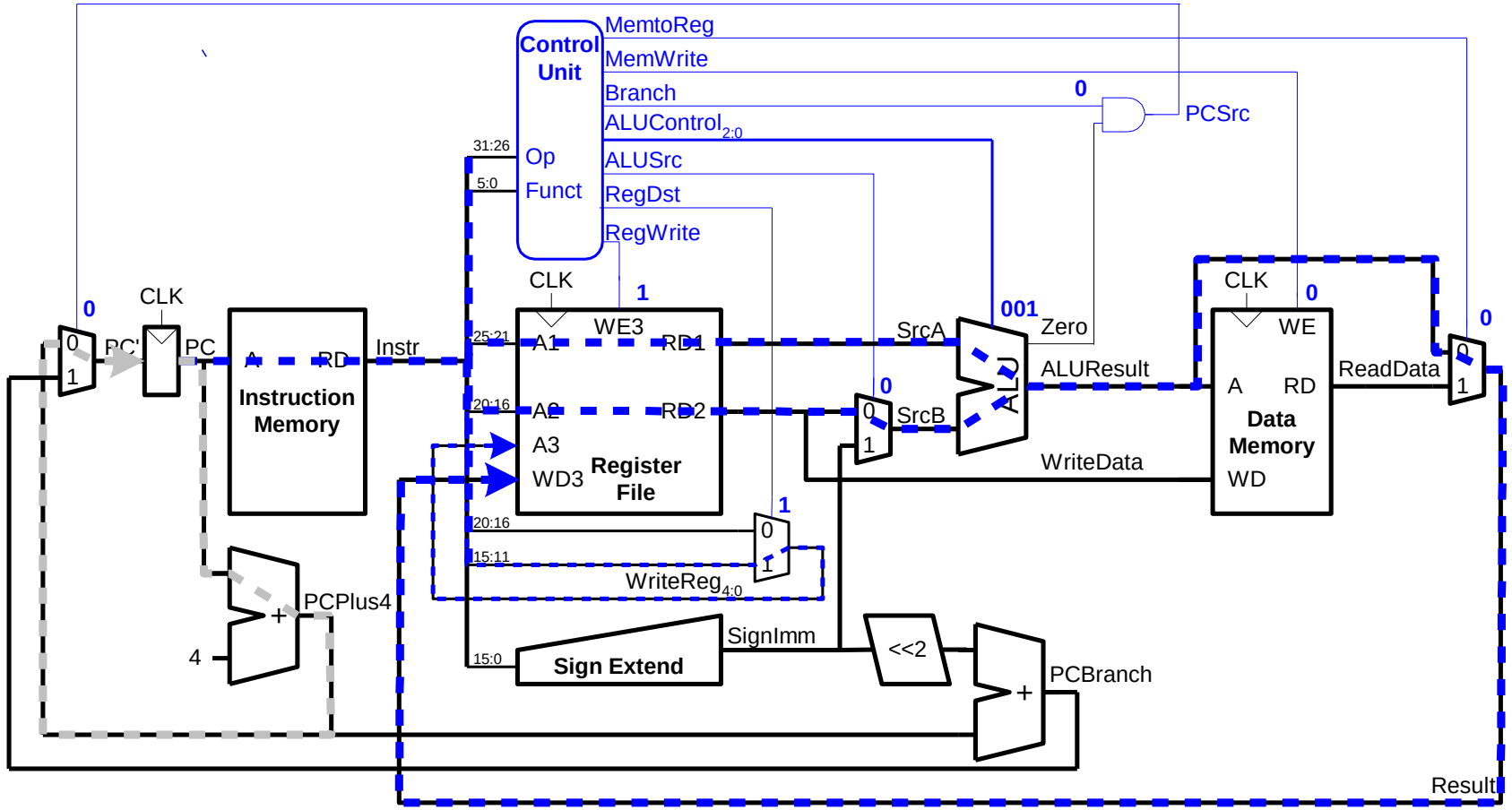
op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

Single-Cycle Datapath: `lw` PC Increment

STEP 6: Determine address of next instruction



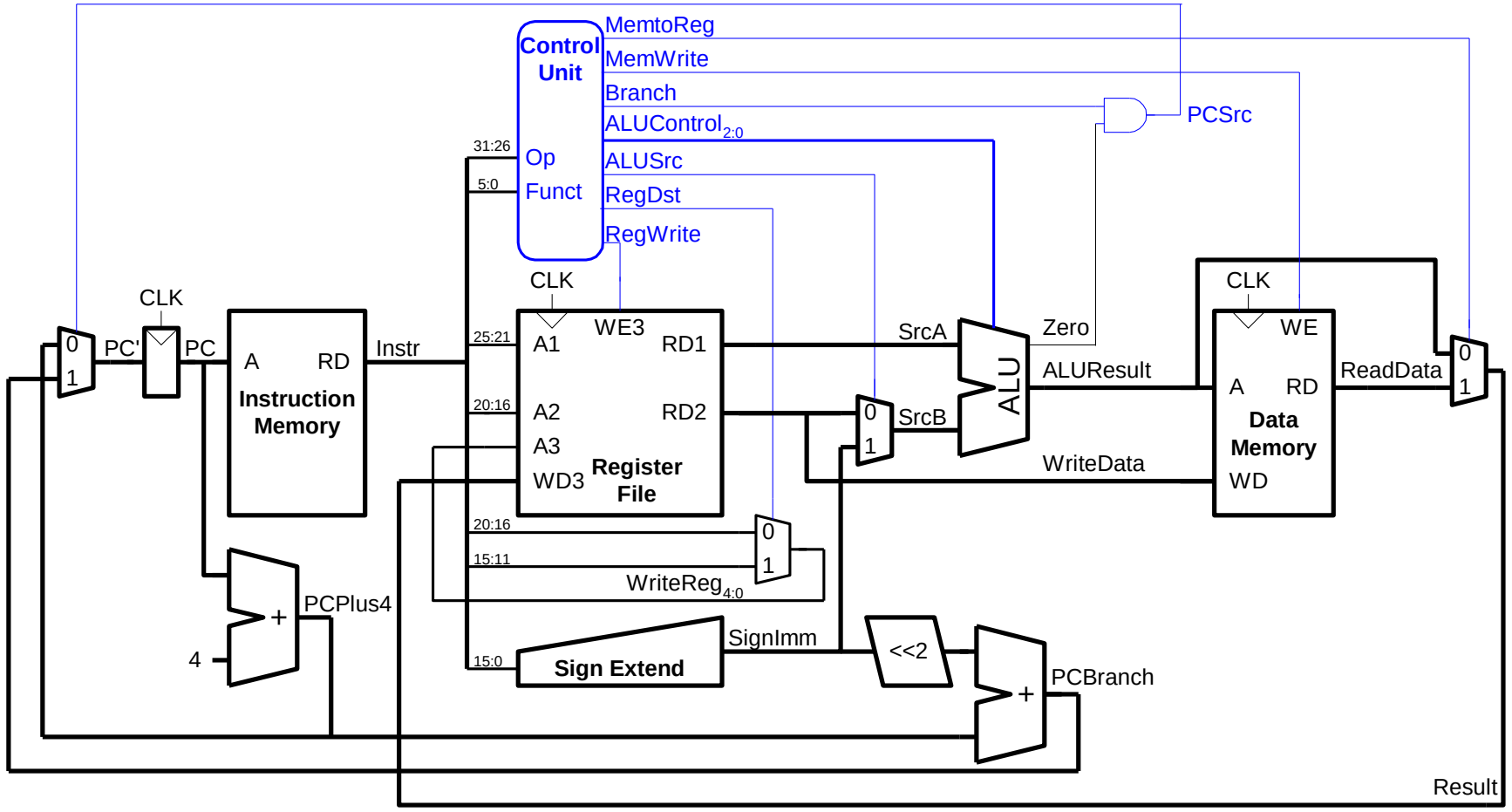
Single-Cycle Datapath: or



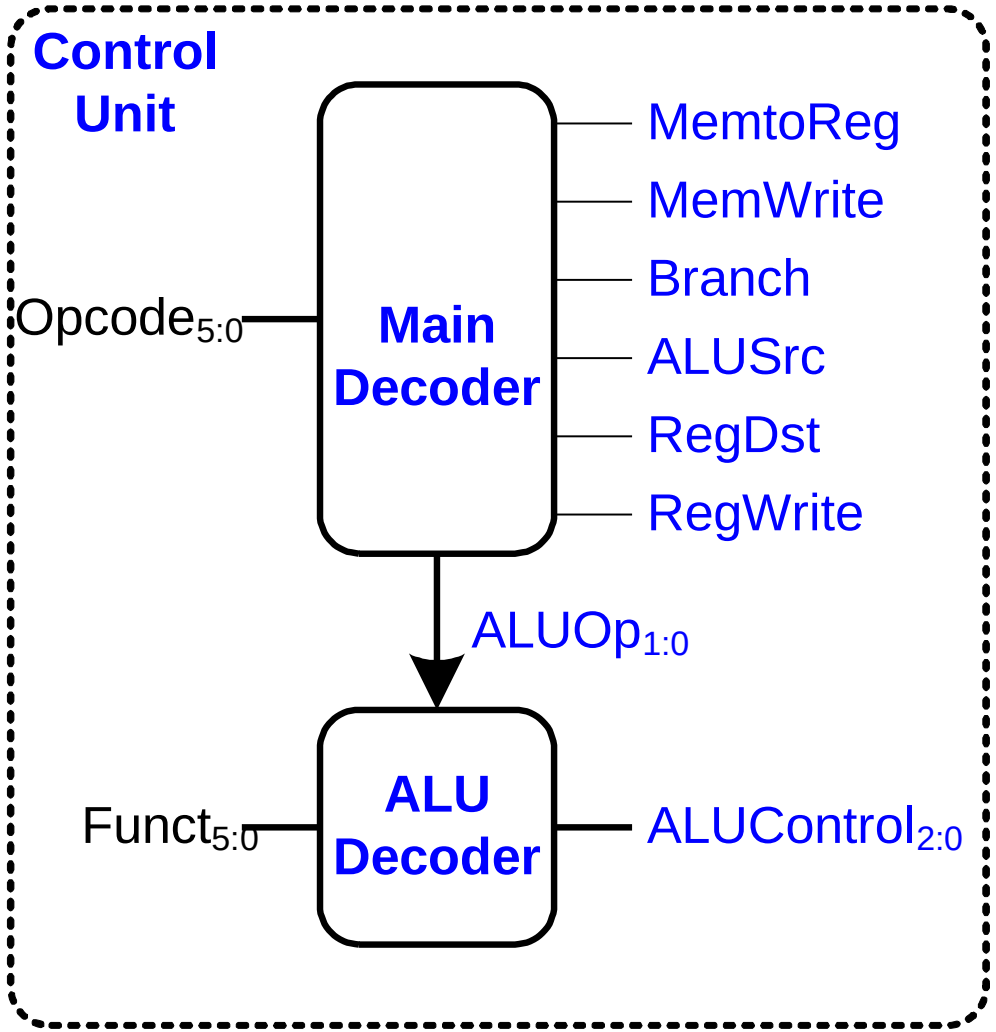
R-Type

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Single-Cycle Processor



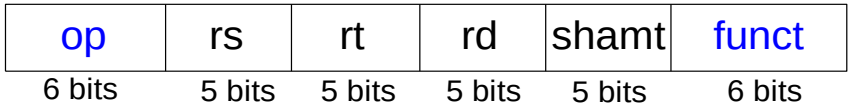
Single-Cycle Control



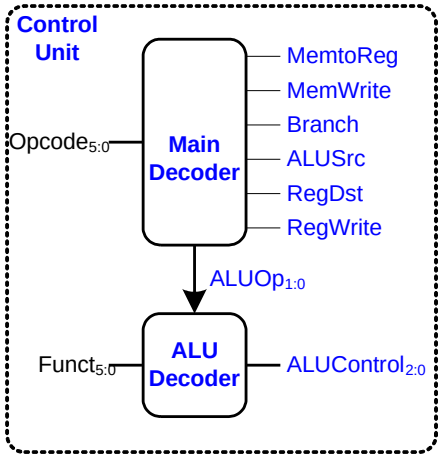
Control Unit: ALU Decoder

$ALUOp_{1:0}$	Meaning
00	Add
01	Subtract
10	Look at Funct
11	Not Used

R-Type

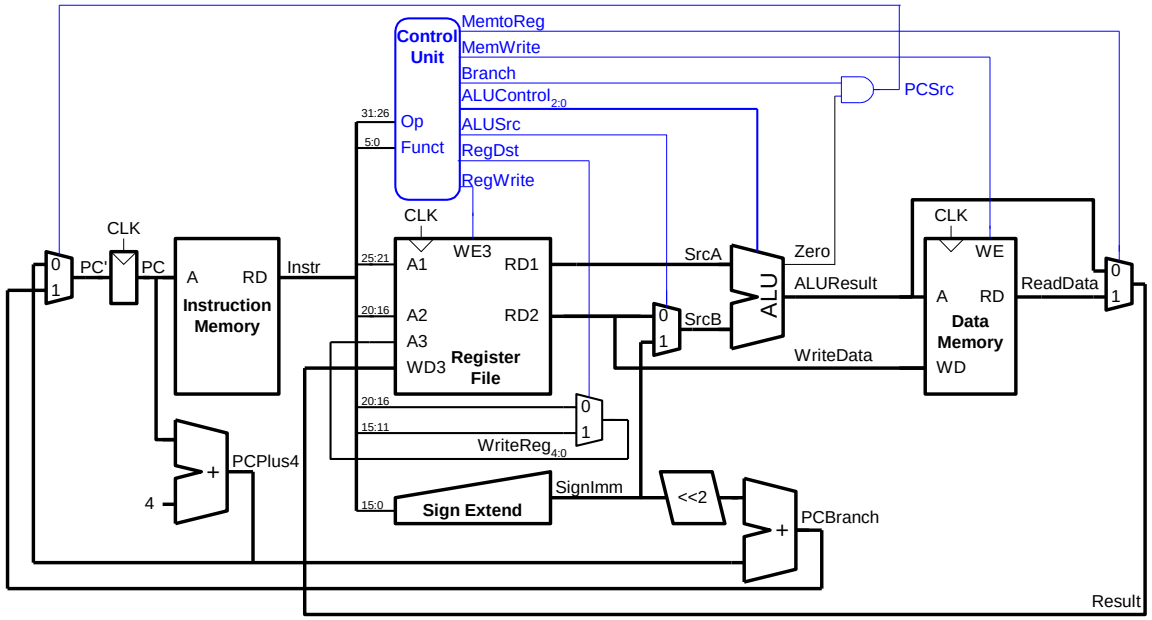


$ALUOp_{1:0}$	Funct	$ALUControl_{2:0}$
00	X	010 (Add)
X1	X	110 (Subtract)
1X	100000 (`add`)	010 (Add)
1X	100010 (`sub`)	110 (Subtract)
1X	100100 (`and`)	000 (And)
1X	100101 (`or`)	001 (Or)
1X	101010 (`slt`)	111 (SLT)



Control Unit Main Decoder

Instruction	$Op_{5:0}$	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	$ALUOp_{1:0}$
R-type	000000							
`lw`	100011							
`sw`	101011							
`beq`	000100							



Control Unit: Main Decoder

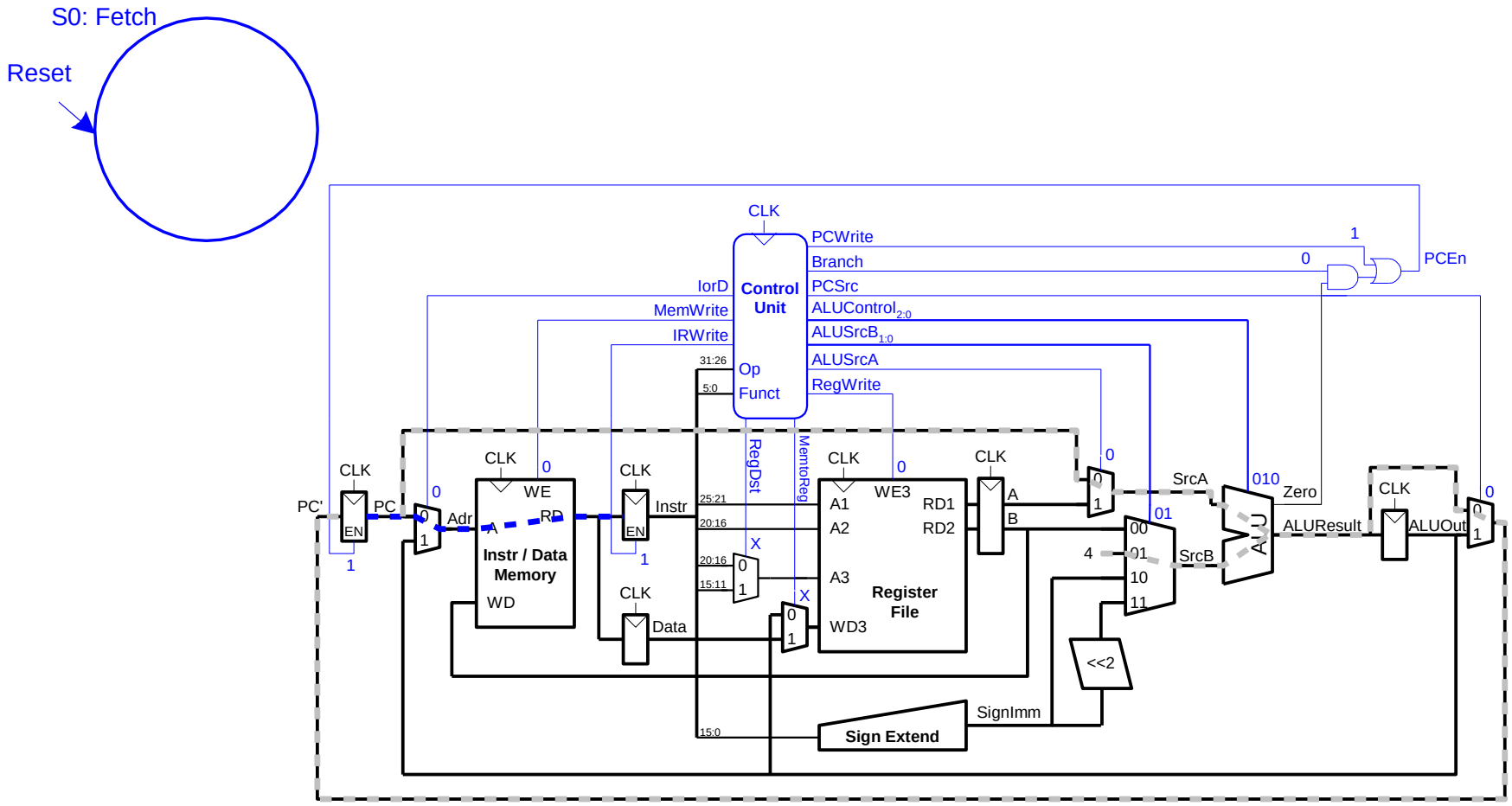
Instruction	$Op_{5:0}$	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	$ALUOp_{1:0}$
R-type	000000	1	1	0	0	0	0	10
`lw`	100011	1	0	1	0	0	1	00
`sw`	101011	0	X	1	0	1	X	00
`beq`	000100	0	X	0	1	0	X	01

$ALUOp_{1:0}$	Meaning
00	Add
01	Subtract
10	Look at Funct
11	Not Used

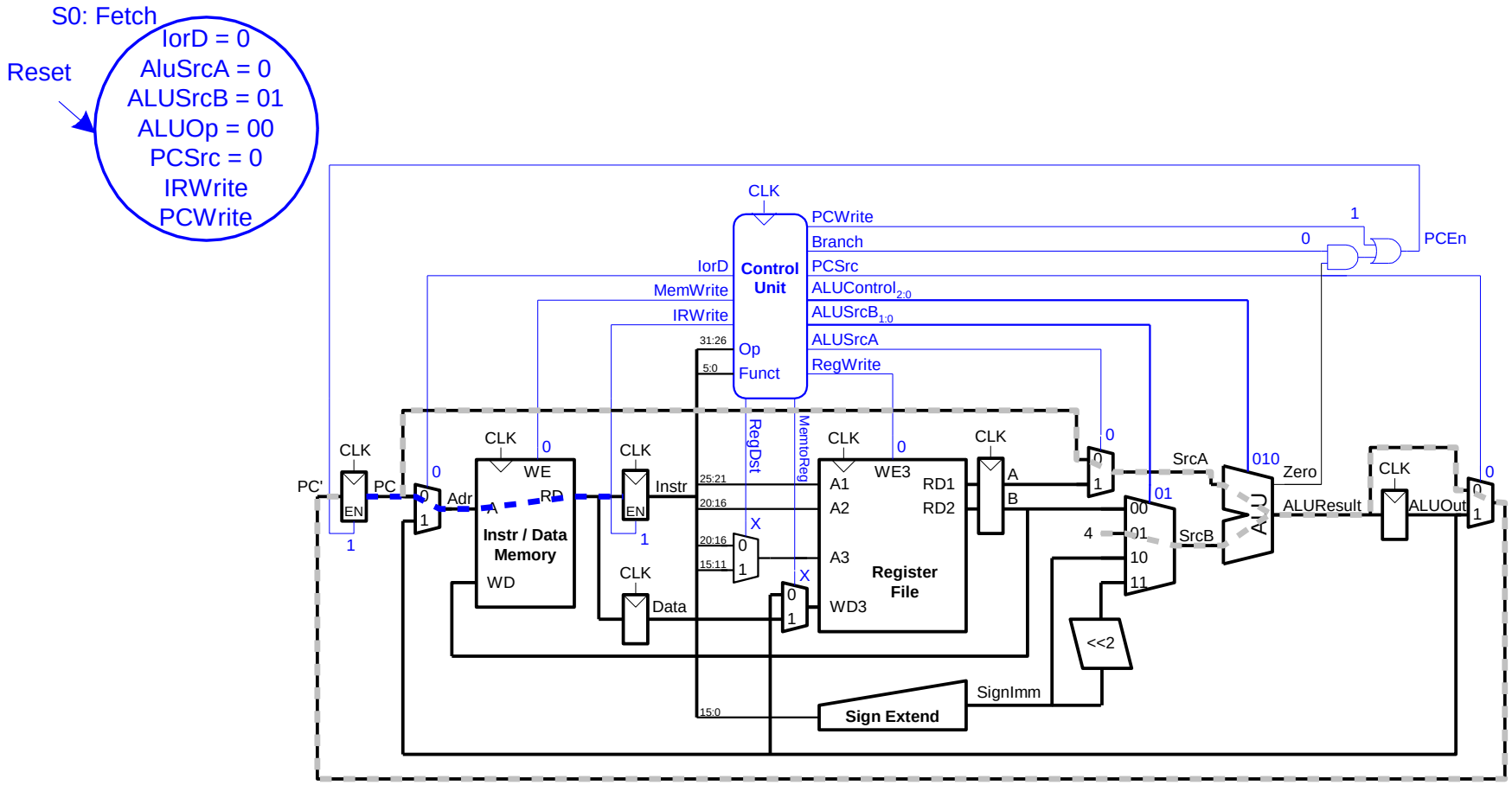
Pipelined MIPS Processor

- Temporal parallelism
- Divide single-cycle processor into 5 stages:
  - Fetch
  - Decode
  - Execute
  - Memory
  - Writeback
- Add pipeline registers between stages

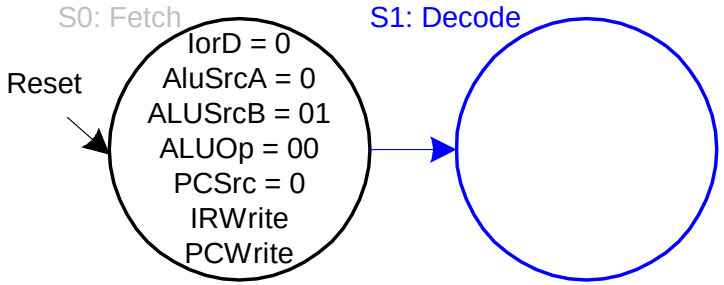
Main Controller FSM: Fetch

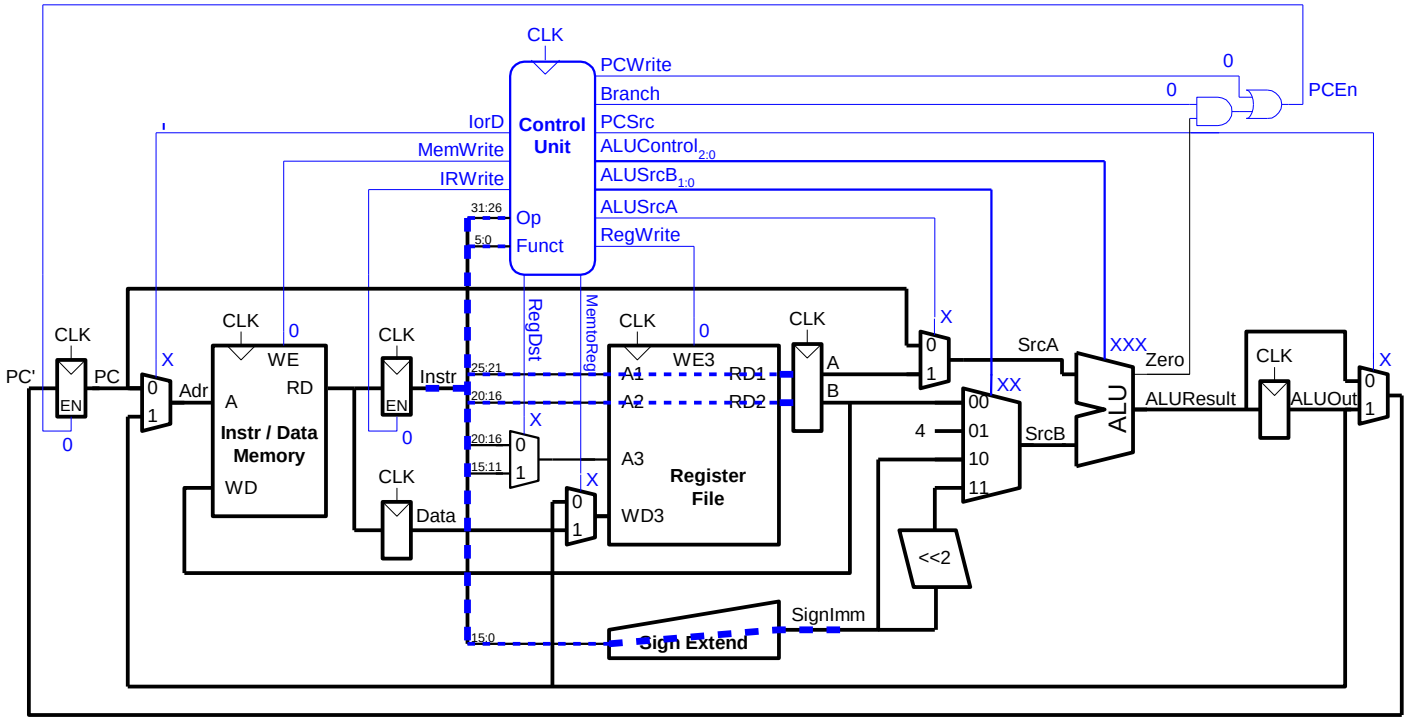


Main Controller FSM: Fetch

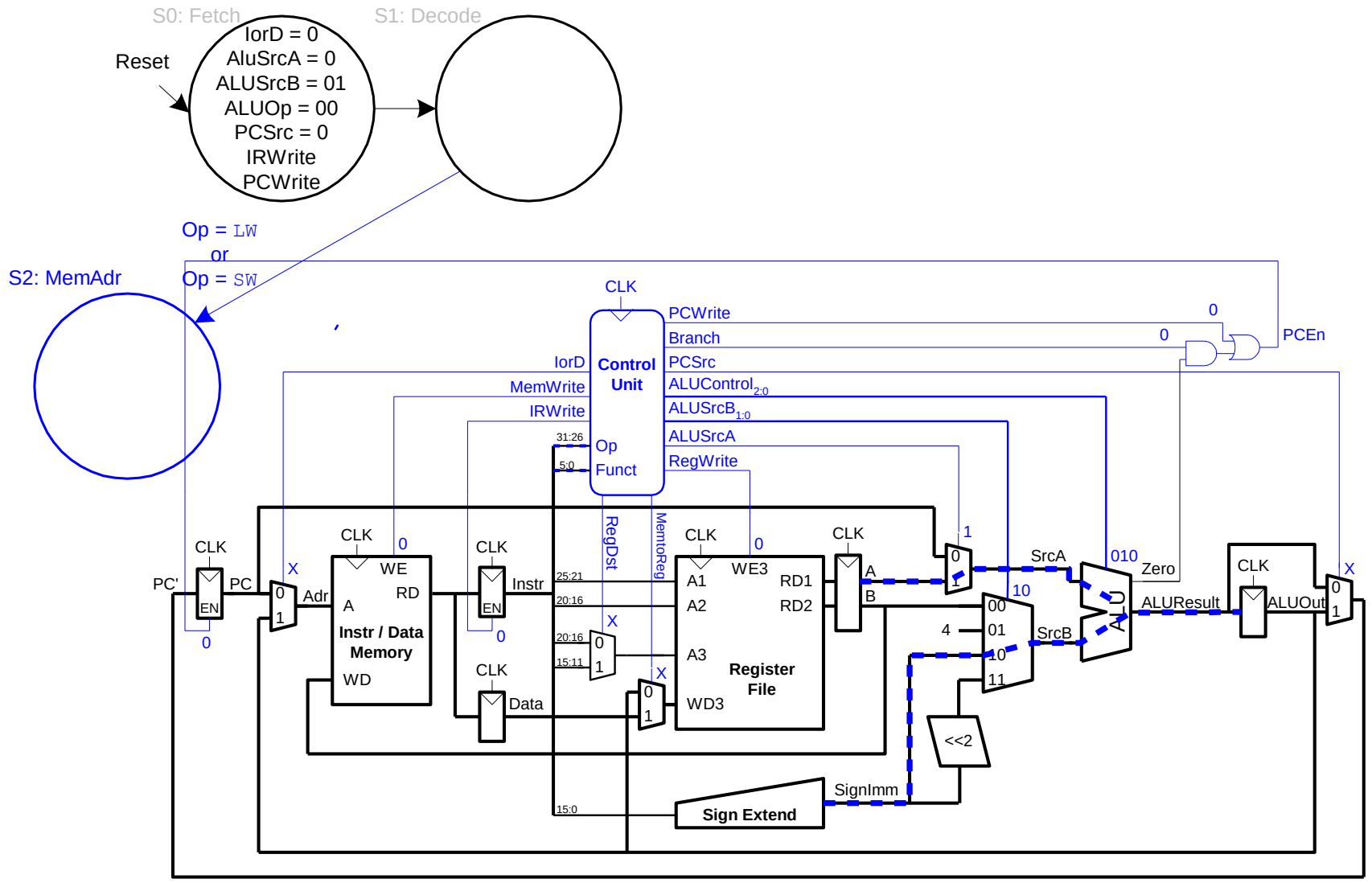


Main Controller FSM: Decode

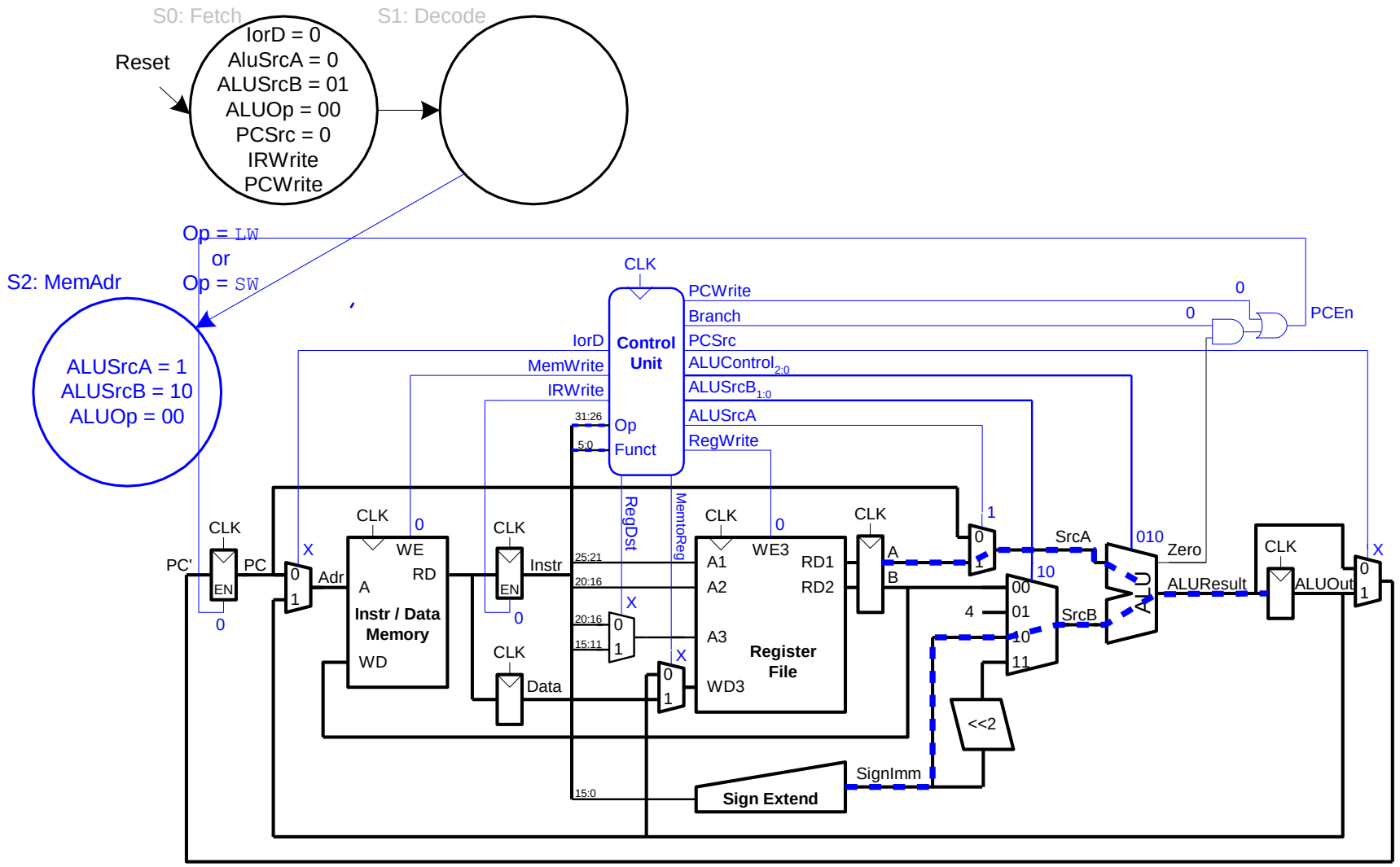




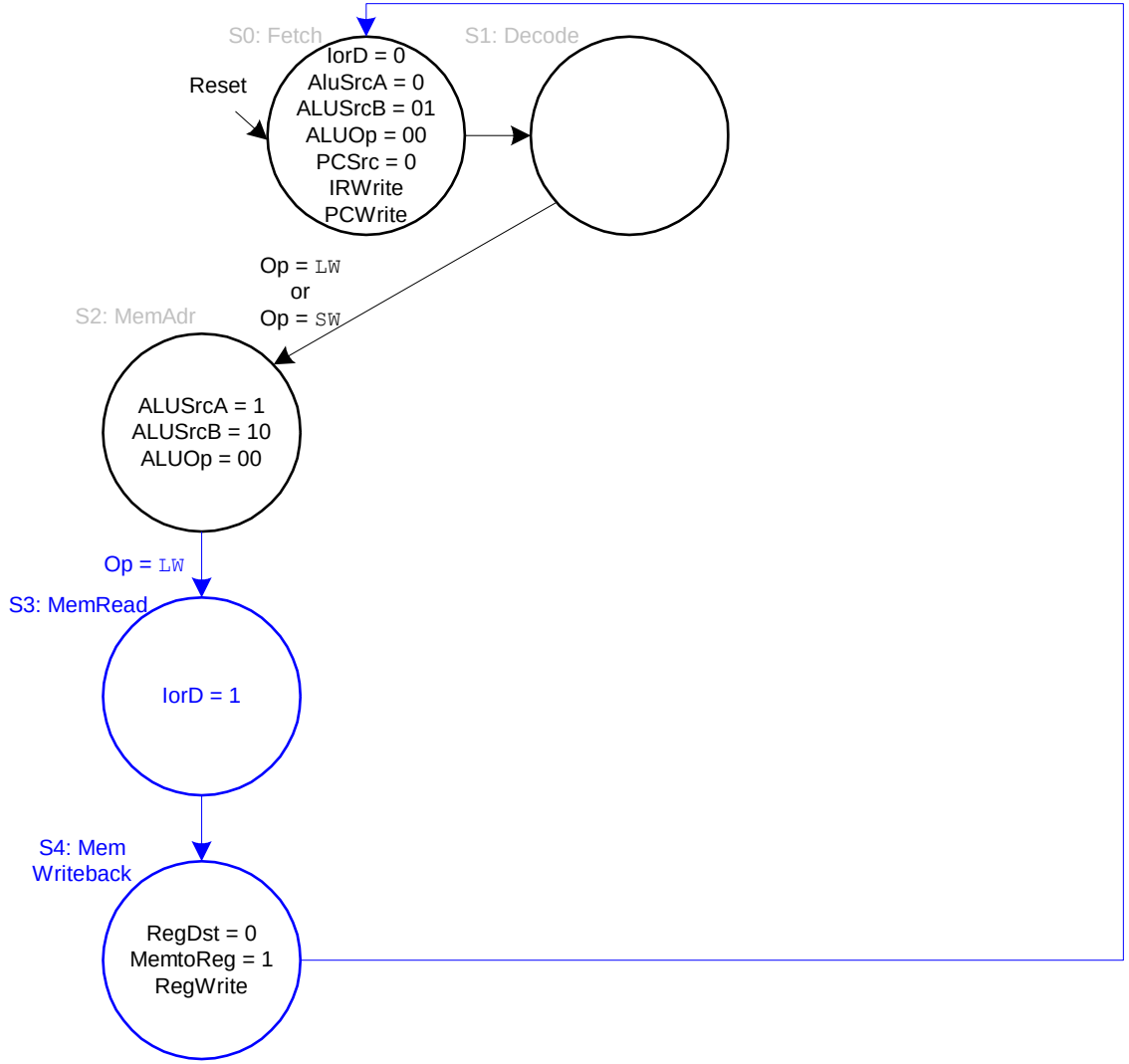
Main Controller FSM: Address



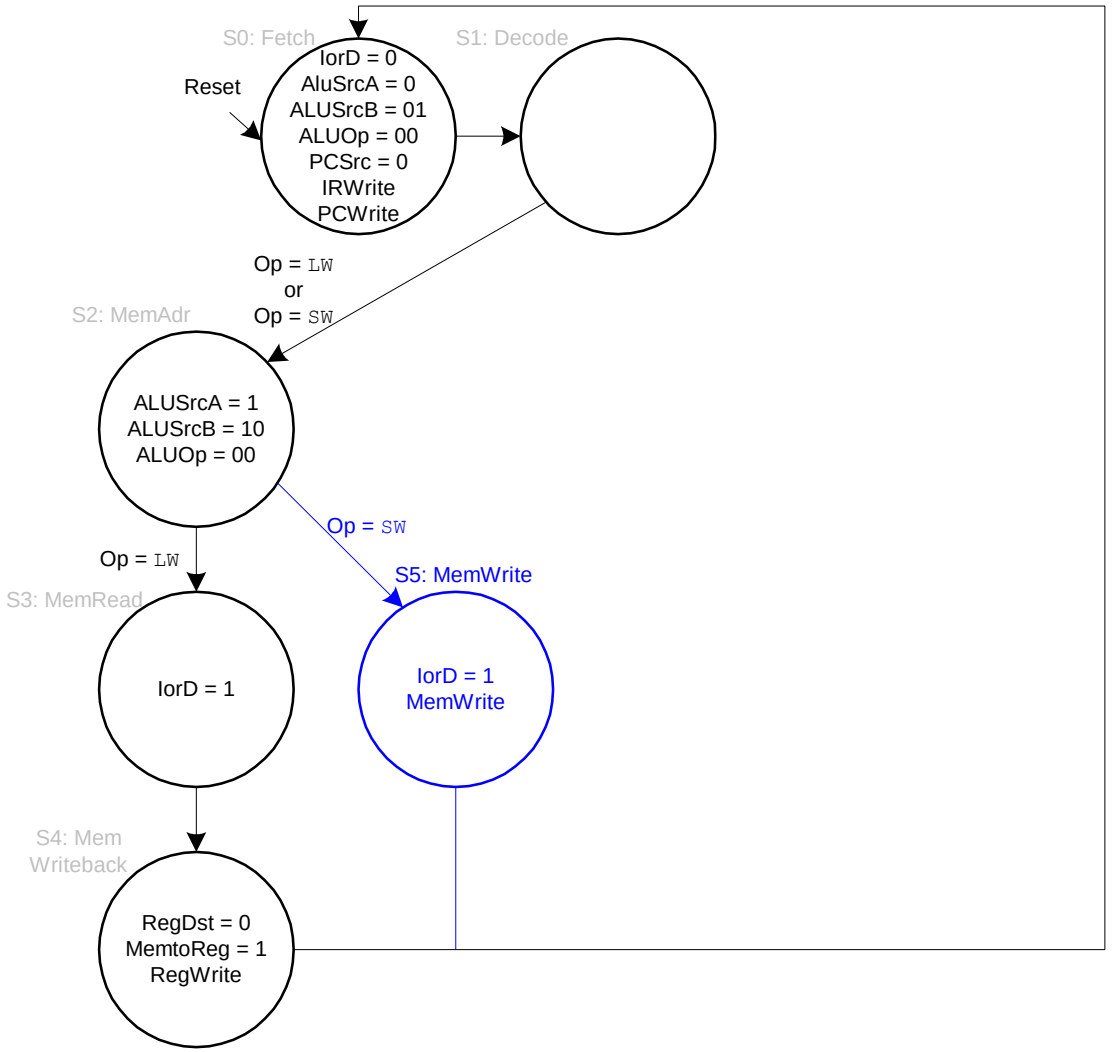
Main Controller FSM: Address



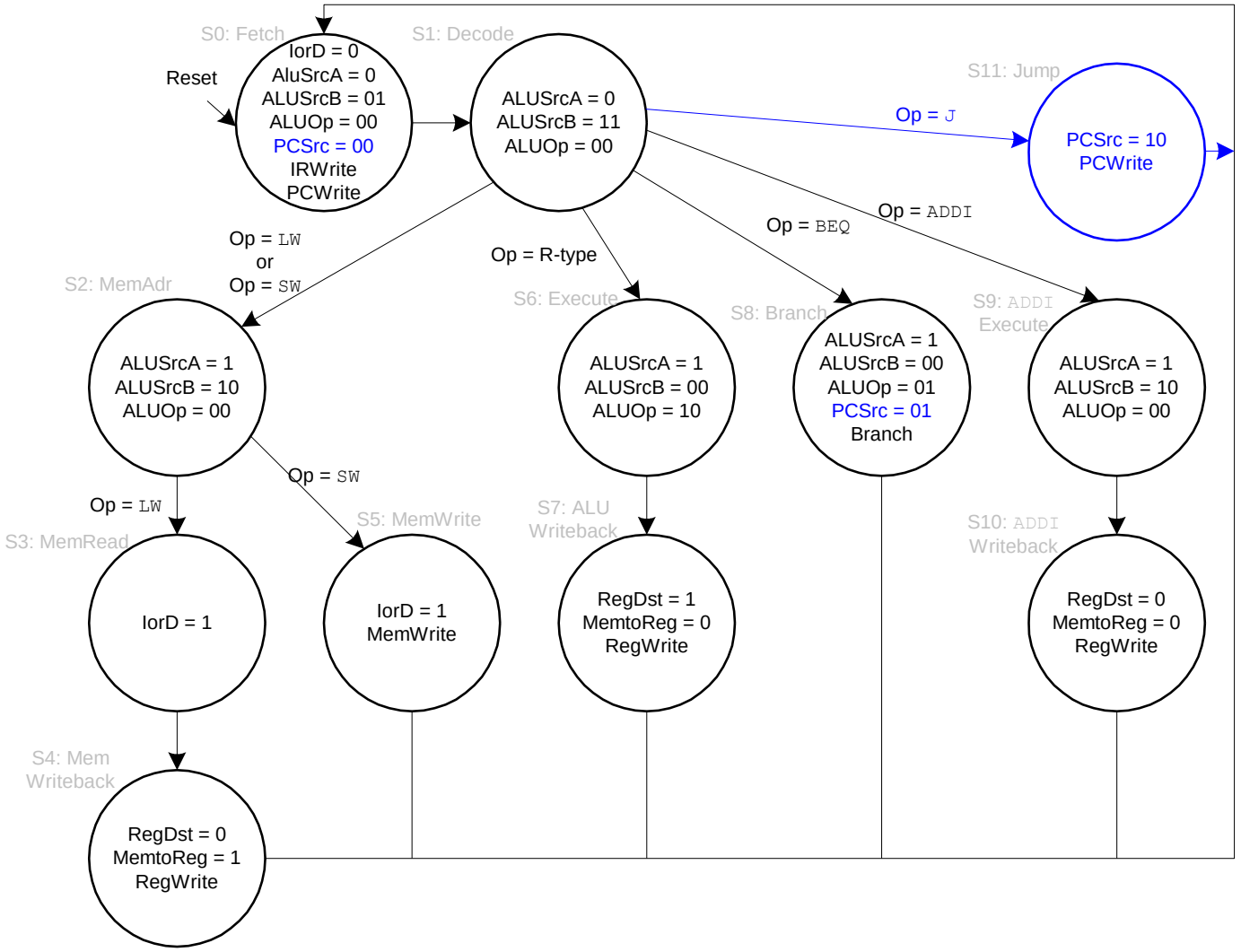
Main Controller FSM: `lw`



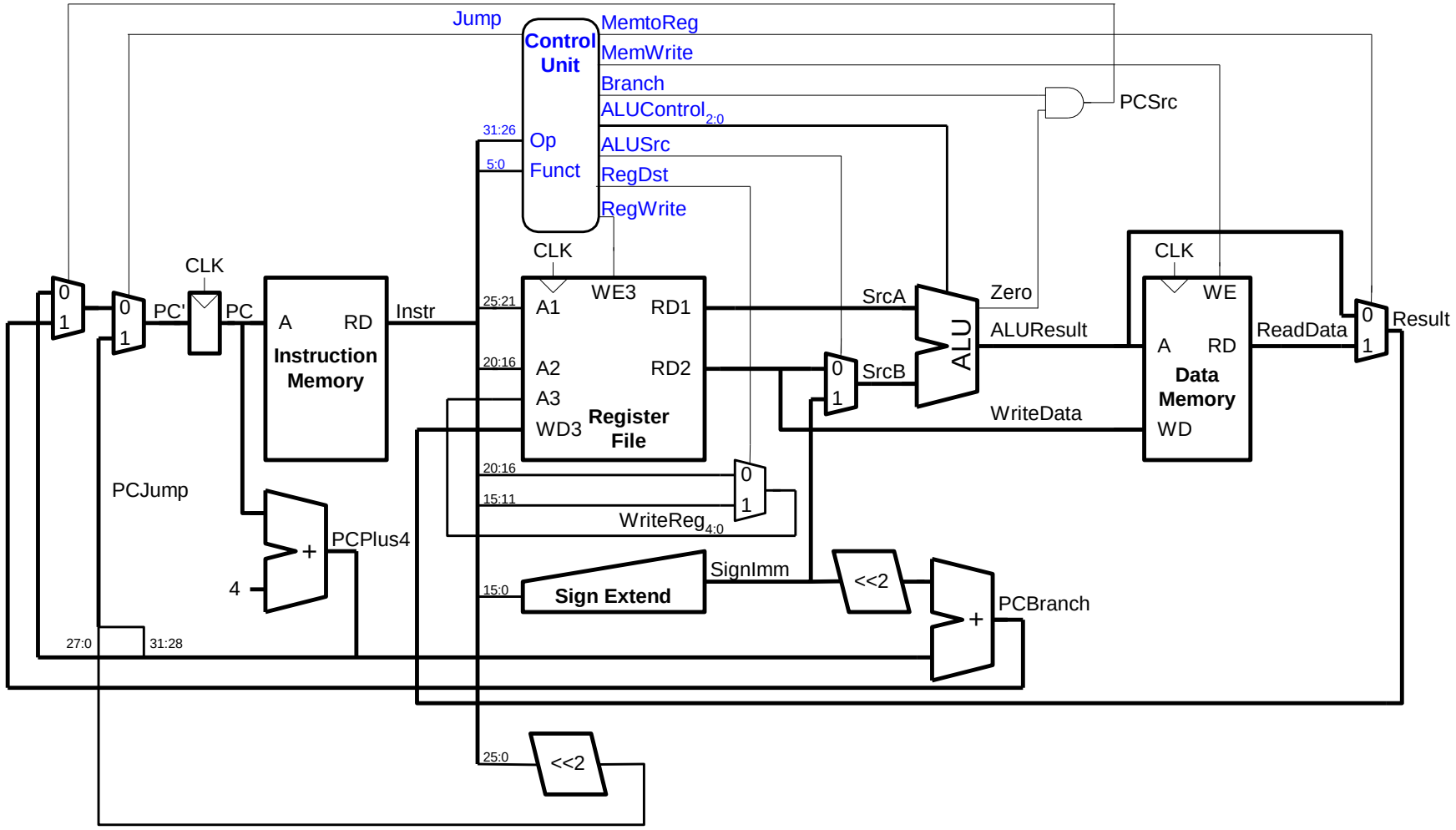
Main Controller FSM: SW



Main Controller FSM: j

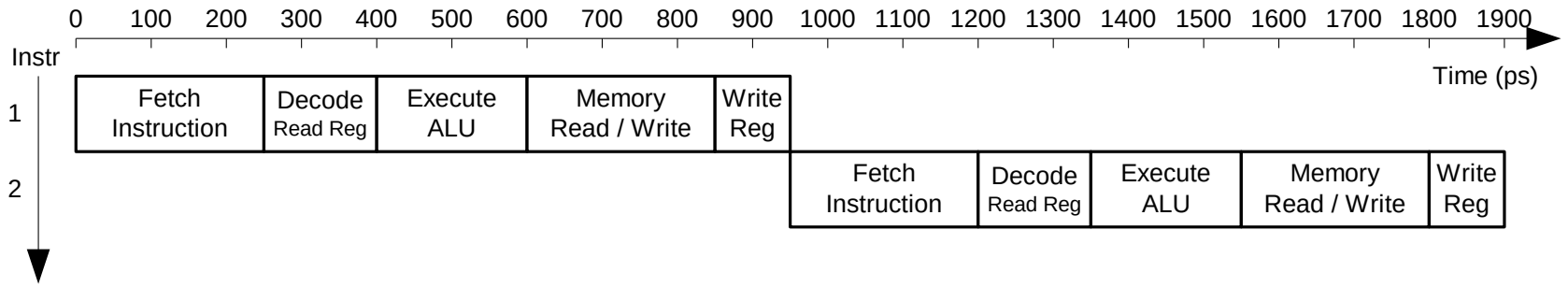


Review: Single-Cycle Processor

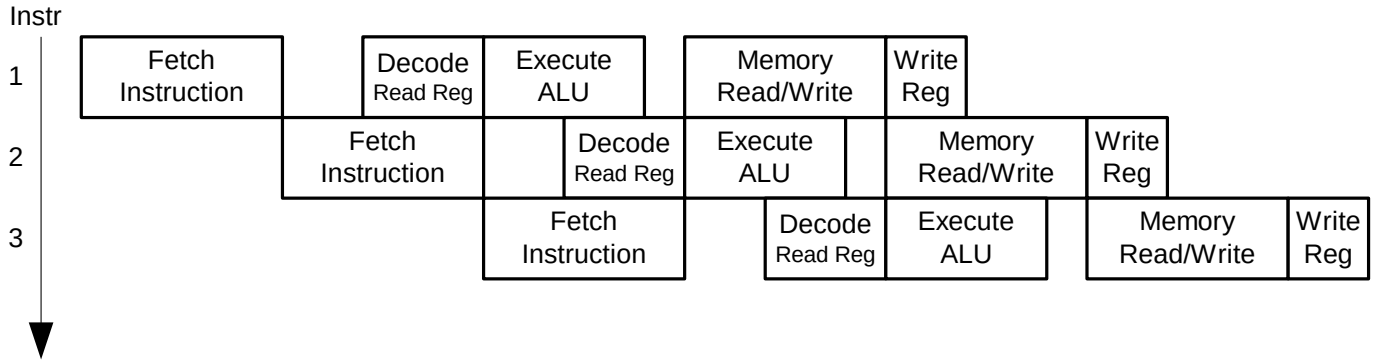


Single-Cycle vs. Pipelined

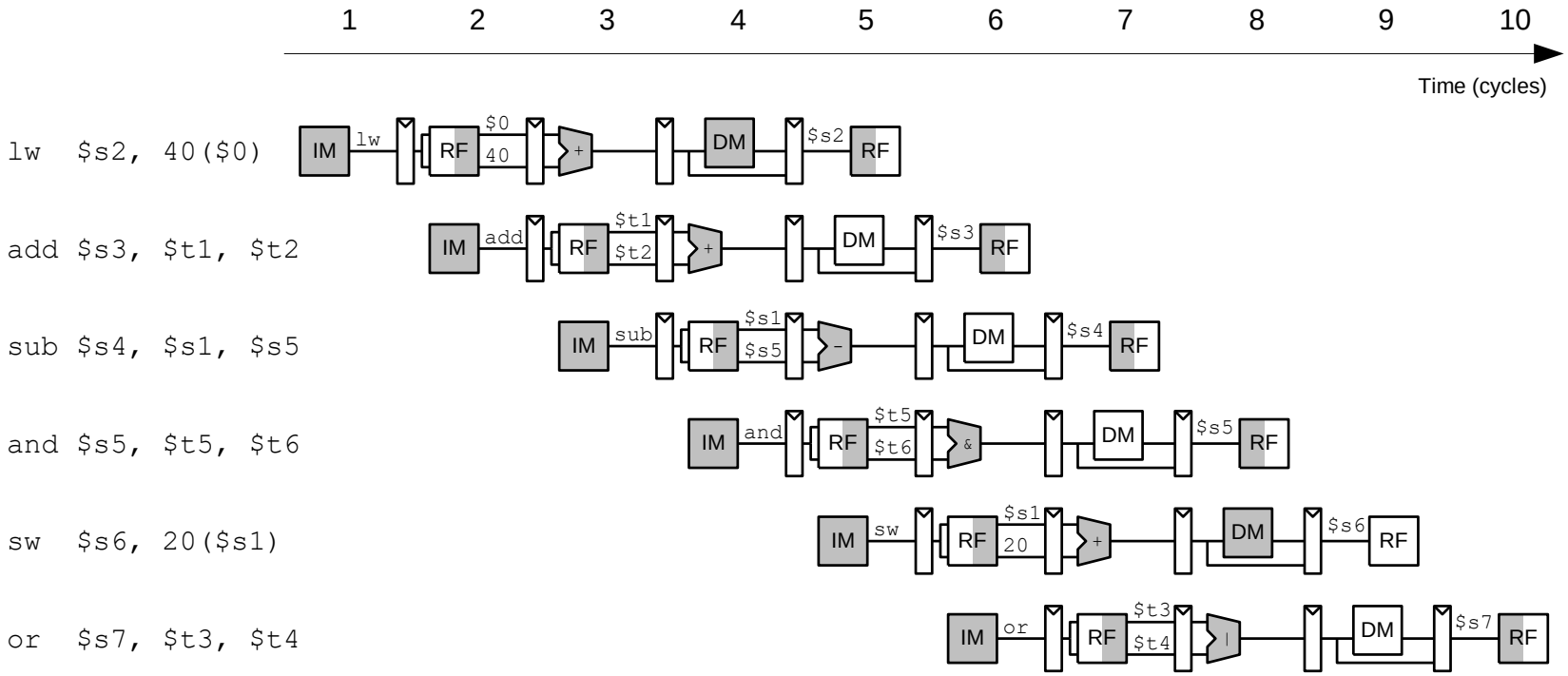
Single-Cycle



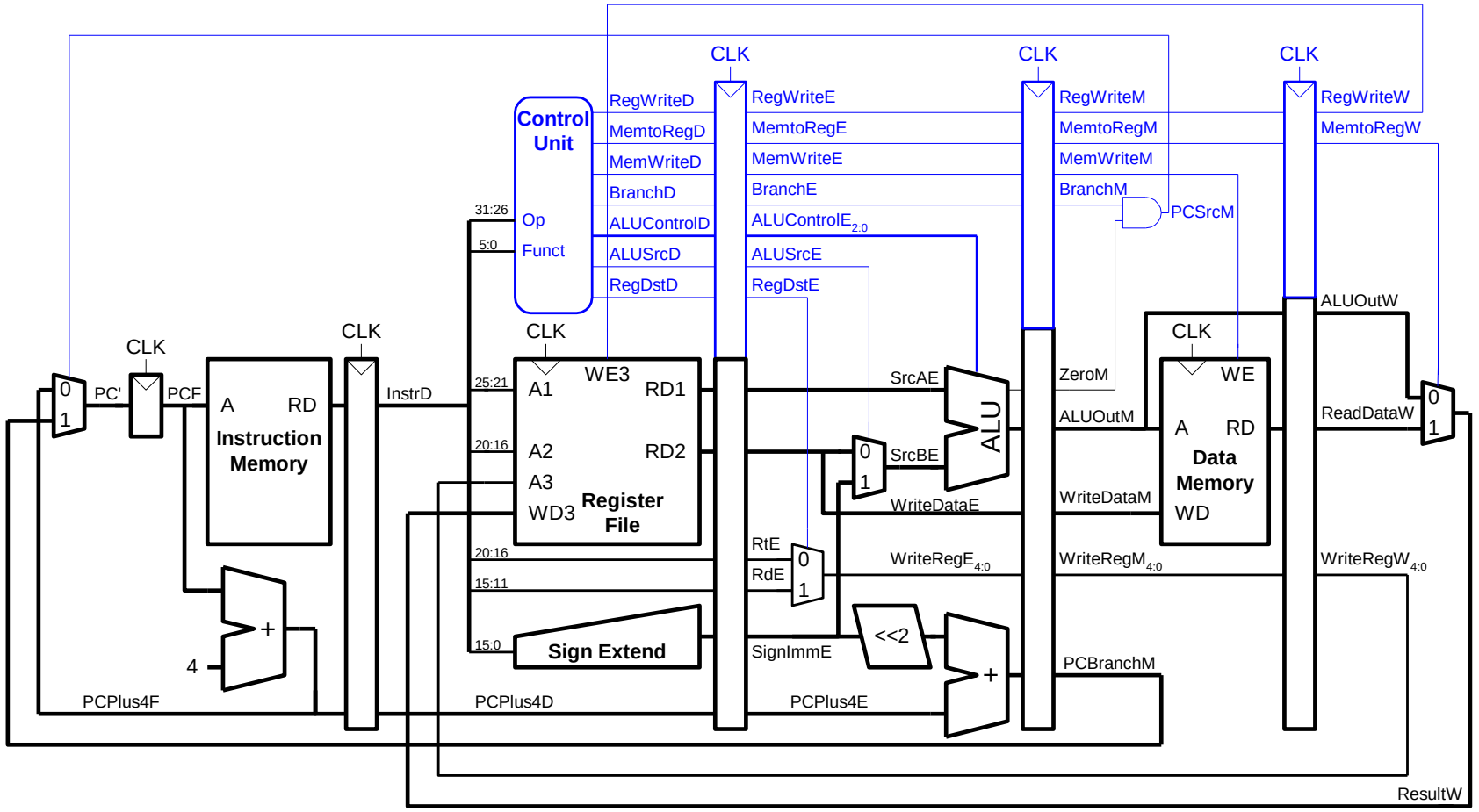
Pipelined



Pipelined Processor Abstraction



Pipelined Processor Control



- **Same control unit as single-cycle processor**
- **Control delayed to proper pipeline stage**

Pipeline Hazards

- When an instruction depends on result from instruction that hasn't completed
- Types:
  - **Data hazard:** register value not yet written back to register file
  - **Control hazard:** next instruction not decided yet (caused by branches)

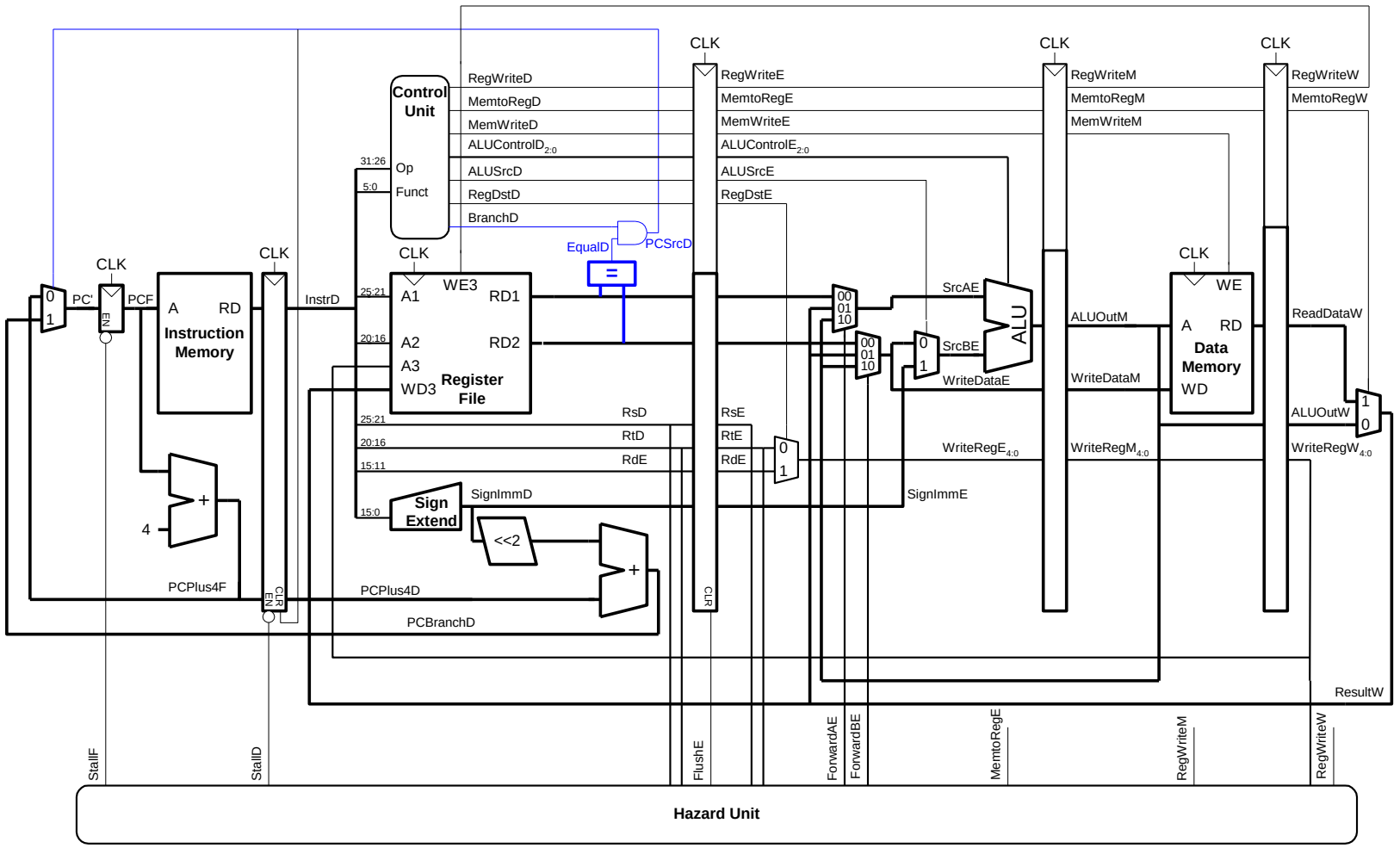
Handling Data Hazards

- Insert `nop`s in code at compile time
- Rearrange code at compile time
- Forward data at run time
- Stall the processor at run time

Handling Control Hazards

- Branch Prediction
  - Guess whether branch will be taken
    - Backward branches are usually taken (loops)
    - Consider history to improve guess
  - Good prediction reduces fraction of branches requiring a flush
  - **Static branch prediction:**
    - Check direction of branch (forward or backward)
    - If backward, predict taken
    - Else, predict not taken
  - **Dynamic branch prediction:**
    - Keep history of last (several hundred) branches in *branch target buffer*, record:
      - Branch destination
      - Whether branch was taken

Early Branch Resolution



Processor Performance

Program Execution Time

= (#instructions)(cycles/instruction)(seconds/cycle)

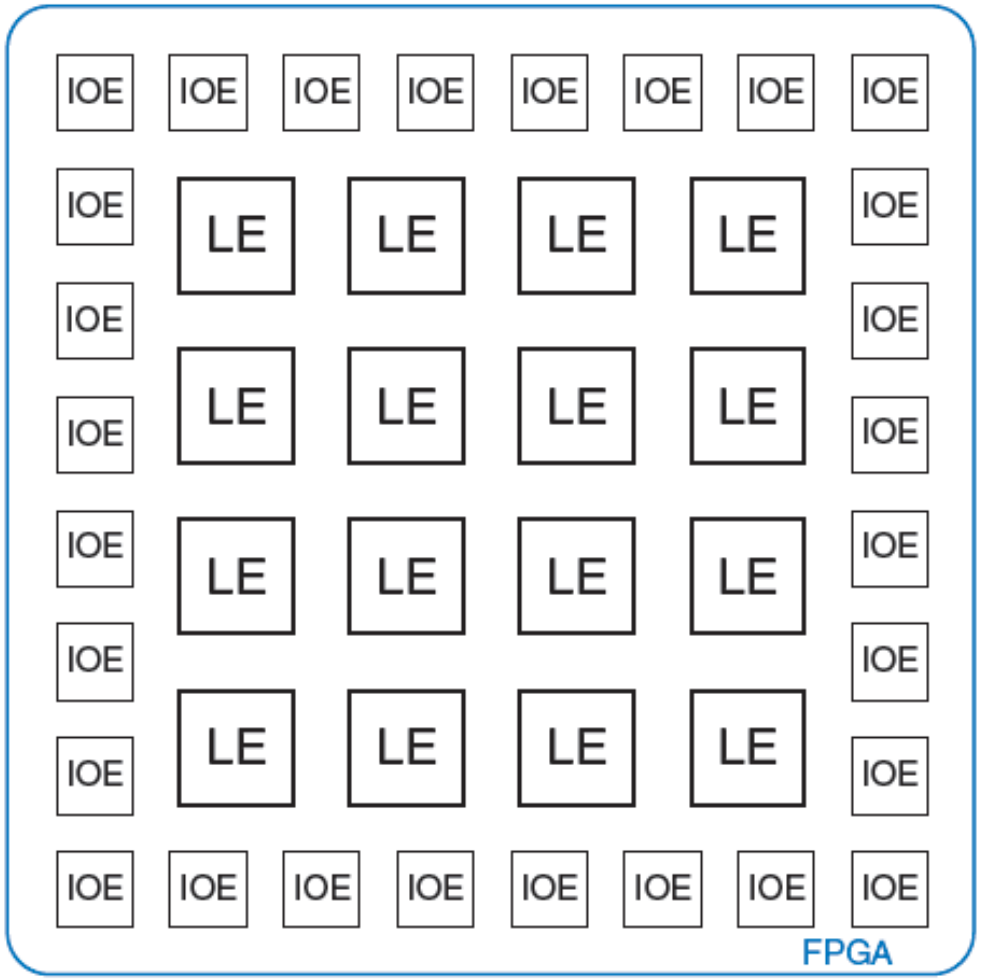
= # instructions x CPI x T_C

Processor	Execution Time (seconds)	Speedup (single-cycle as baseline)
Single-cycle	92.5	1
Multicycle	133	0.70
Pipelined	63	1.47

FPGA: Field Programmable Gate Array

- Composed of:
  - **LEs** (Logic elements): perform logic
  - **IOEs** (Input/output elements): interface with outside world
  - **Programmable interconnection:** connect LEs and IOEs
  - Some FPGAs include other building blocks such as multipliers and RAMs

General FPGA Layout



LE: Logic Element

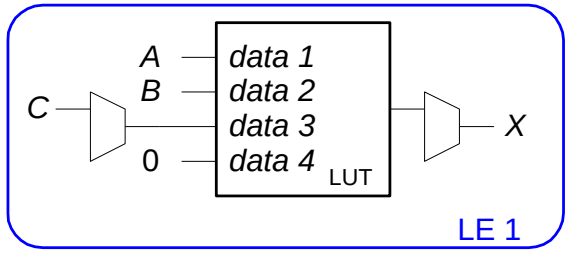
- Composed of:
  - **LUTs** (lookup tables): perform combinational logic
  - **Flip-flops:** perform sequential logic
  - **Multiplexers:** connect LUTs and flip-flops

LE Configuration Example

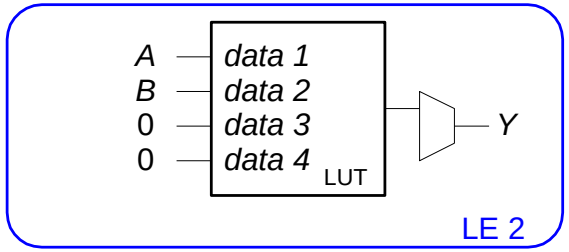
Show how to configure a Cyclone IV LE to perform the following functions:

- $X = \overline{A}\,\overline{B}C + AB\overline{C}$
- $Y = A\overline{B}$

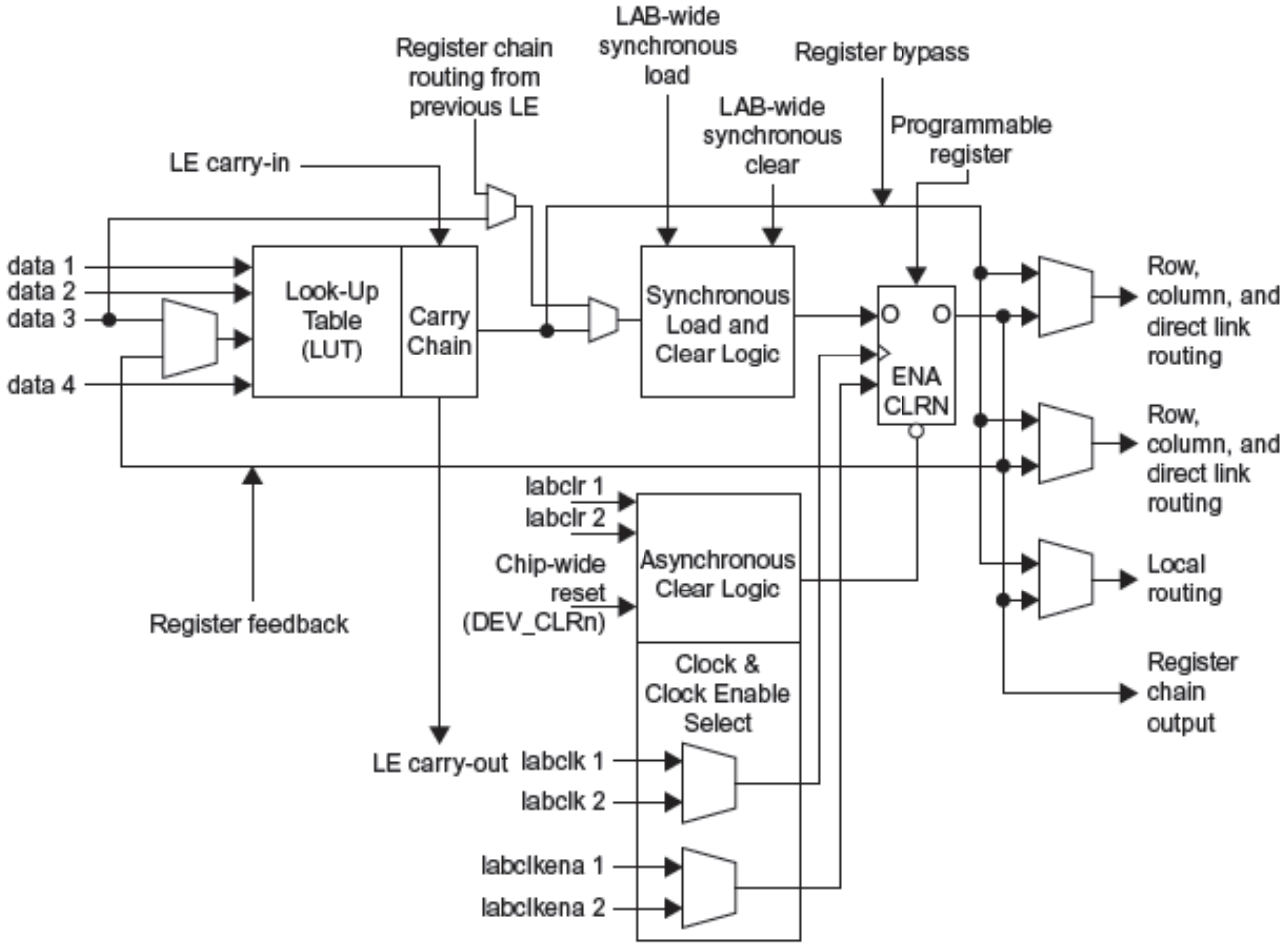
(A) *data 1*	(B) *data 2*	(C) *data 3*	*data 4*	(X) LUT output
0	0	0	X	0
0	0	1	X	1
0	1	0	X	0
0	1	1	X	0
1	0	0	X	0
1	0	1	X	0
1	1	0	X	1
1	1	1	X	0



(A) *data 1*	(B) *data 2*	*data 3*	*data 4*	(Y) LUT output
0	0	X	X	0
0	1	X	X	0
1	0	X	X	1
1	1	X	X	0



Altera Cyclone IV LE

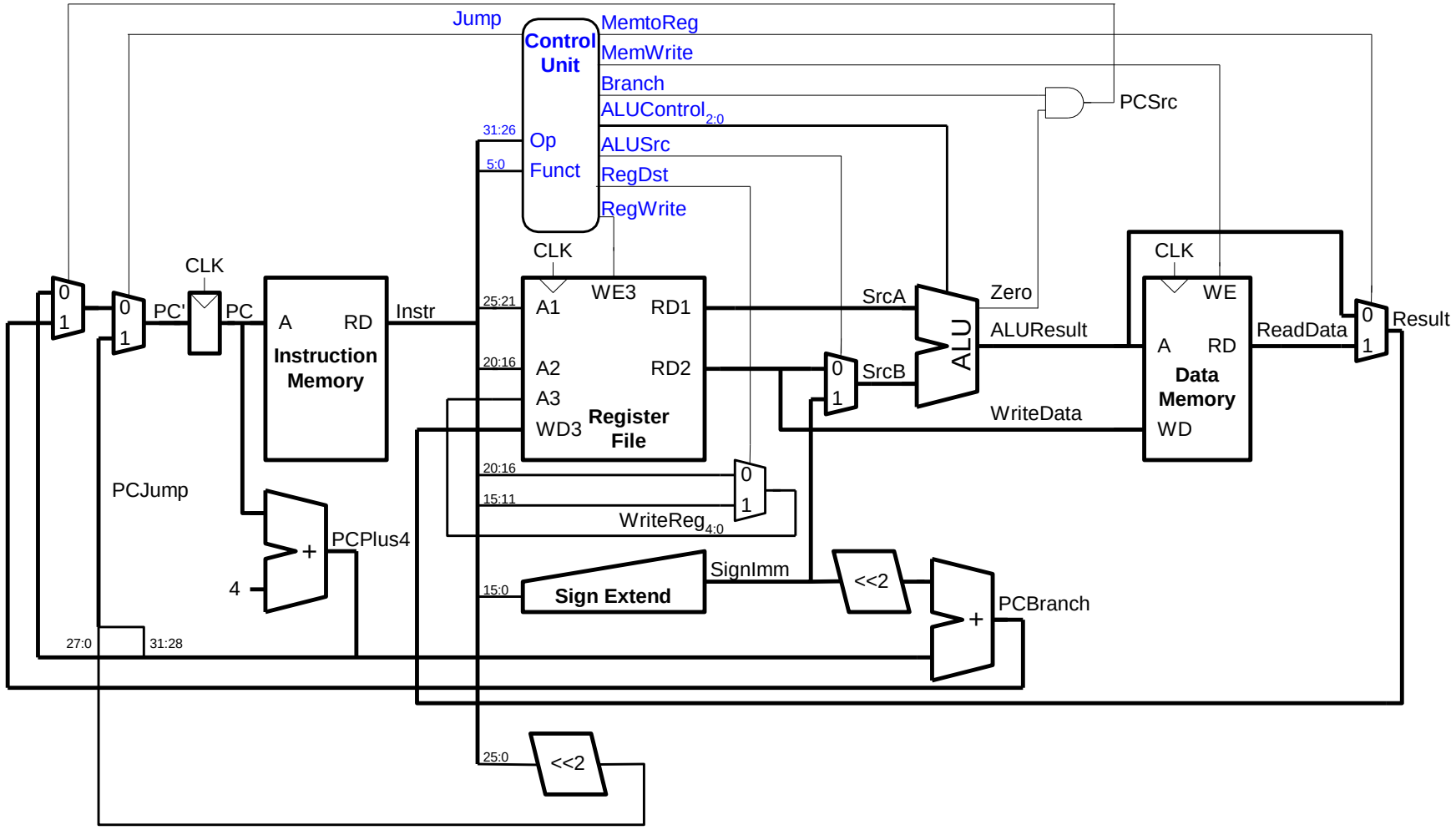


FPGA Design Flow

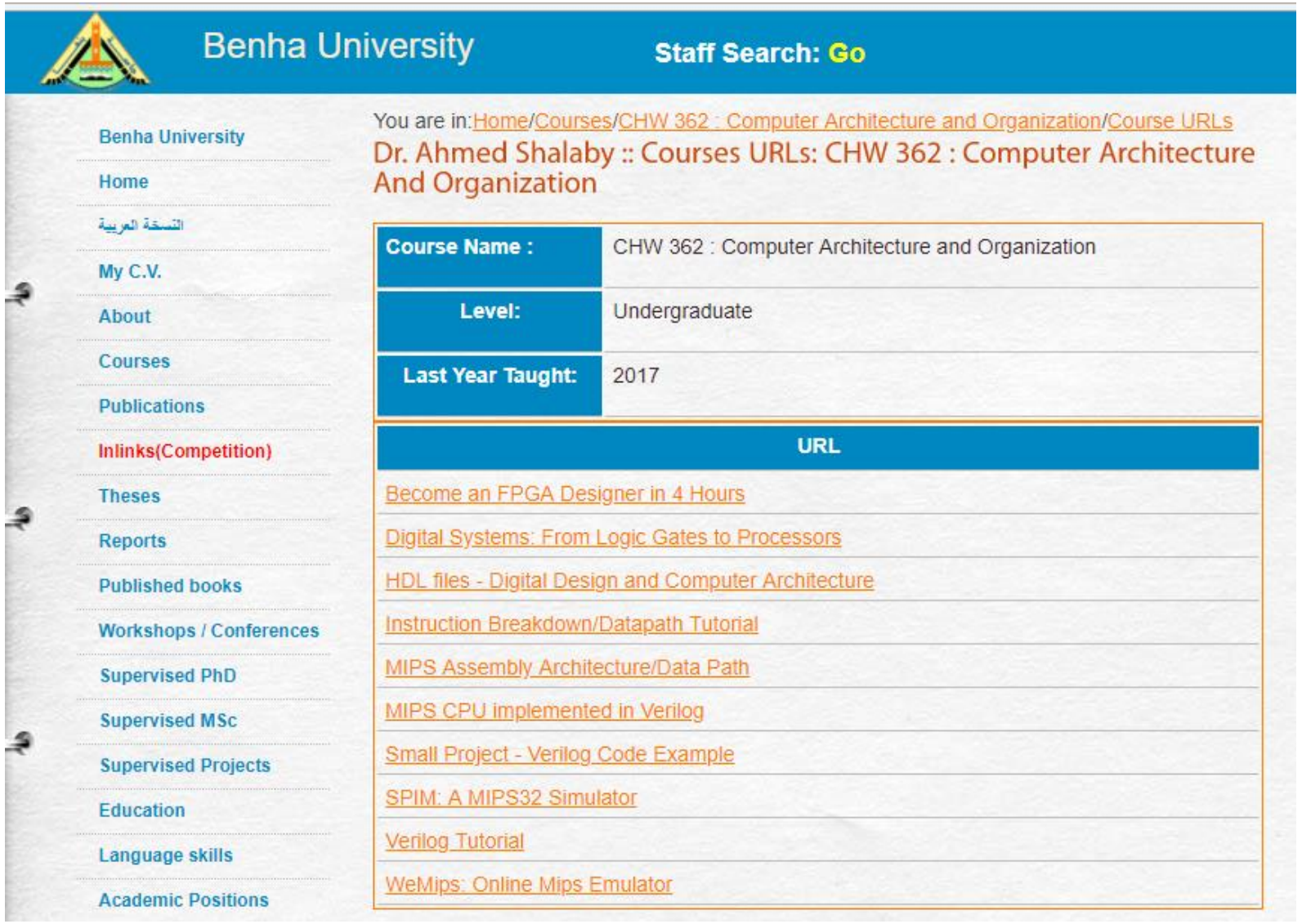
Using a CAD tool (such as Altera's Quartus II)

- **Enter the design** using schematic entry or an HDL
- **Simulate** the design
- **Synthesize** design and map it onto FPGA
- **Download the configuration** onto the FPGA
- **Test** the design

Review: Single-Cycle Processor



Thank You



Benha University Staff Search: Go

- Benha University
- Home
- النسخة العربية
- My C.V.
- About
- Courses
- Publications
- Inlinks(Competition)
- Theses
- Reports
- Published books
- Workshops / Conferences
- Supervised PhD
- Supervised MSc
- Supervised Projects
- Education
- Language skills
- Academic Positions

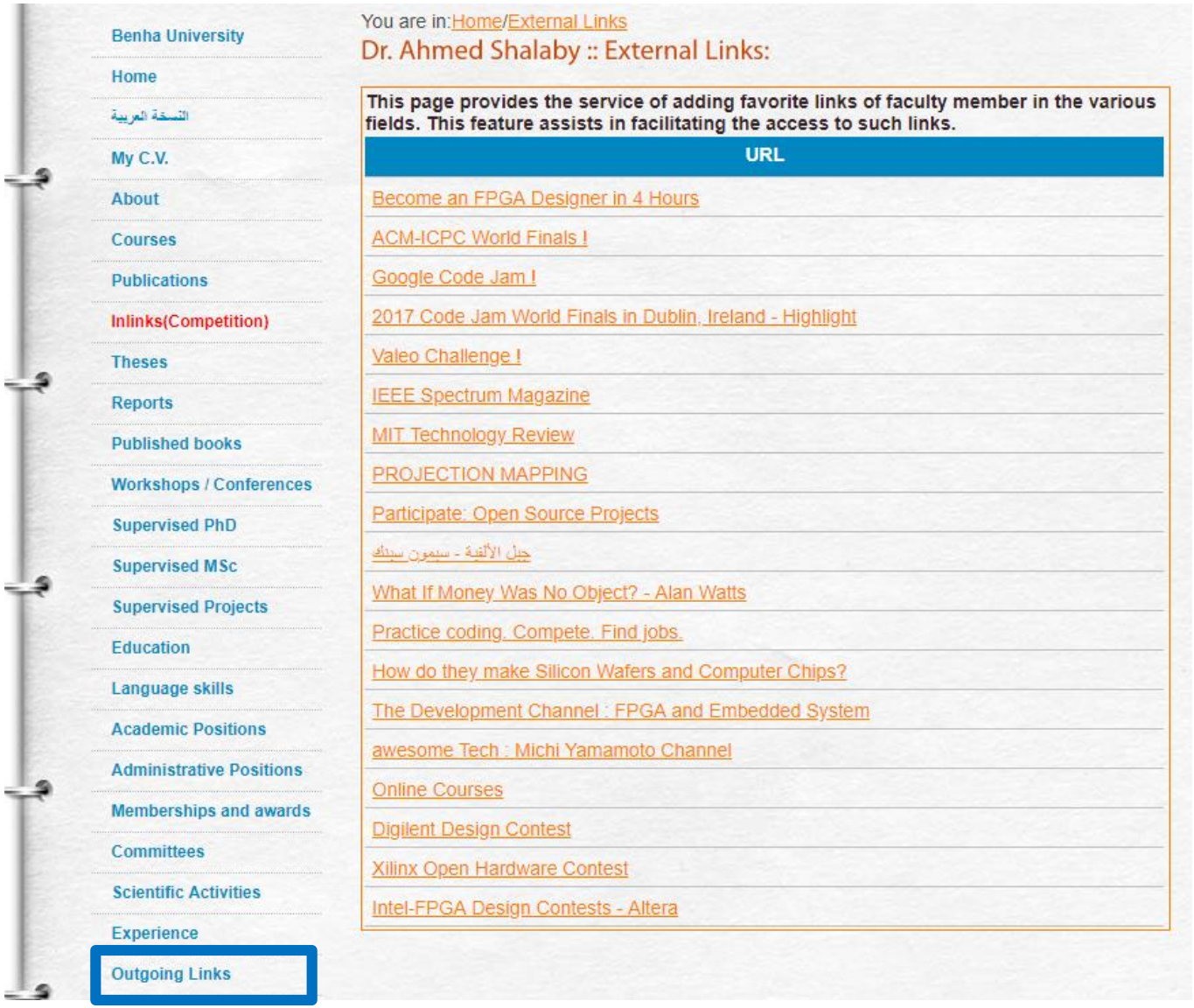
You are in:Home/Courses/CHW 362 : Computer Architecture and Organization/Course URLs

Dr. Ahmed Shalaby :: Courses URLs: CHW 362 : Computer Architecture And Organization

Course Name :	CHW 362 : Computer Architecture and Organization
Level:	Undergraduate
Last Year Taught:	2017

URL
Become an FPGA Designer in 4 Hours
Digital Systems: From Logic Gates to Processors
HDL files - Digital Design and Computer Architecture
Instruction Breakdown/Datapath Tutorial
MIPS Assembly Architecture/Data Path
MIPS CPU implemented in Verilog
Small Project - Verilog Code Example
SPIM: A MIPS32 Simulator
Verilog Tutorial
WeMips: Online Mips Emulator

Thank You


Benha University
Home
النسخة العربية
My C.V.
About
Courses
Publications
Inlinks(Competition)
Theses
Reports
Published books
Workshops / Conferences
Supervised PhD
Supervised MSc
Supervised Projects
Education
Language skills
Academic Positions
Administrative Positions
Memberships and awards
Committees
Scientific Activities
Experience
Outgoing Links
You are in:Home/External Links
Dr. Ahmed Shalaby :: External Links:
This page provides the service of adding favorite links of faculty member in the various fields. This feature assists in facilitating the access to such links.
URL
Become an FPGA Designer in 4 Hours
ACM-ICPC World Finals !
Google Code Jam !
2017 Code Jam World Finals in Dublin, Ireland - Highlight
Valeo Challenge !
IEEE Spectrum Magazine
MIT Technology Review
PROJECTION MAPPING
Participate: Open Source Projects
جيل الألفية - سيمون سينك
What If Money Was No Object? - Alan Watts
Practice coding. Compete. Find jobs.
How do they make Silicon Wafers and Computer Chips?
The Development Channel : FPGA and Embedded System
awesome Tech : Michi Yamamoto Channel
Online Courses
Digilent Design Contest
Xilinx Open Hardware Contest
Intel-FPGA Design Contests - Altera

Thank You

- **Oral Exam (Your Name)**
  - **Verilog Project**
  - **MIPS Project**
  - **Comparison between RISC-V and MIPS**
- **Final Exam**
- **Graduation Project (Idea, Job, Research)**