CSE 260M / ESE 260
Intro. To Digital Logic & Computer Design

Bill Siever
&
Michael Hall

This week

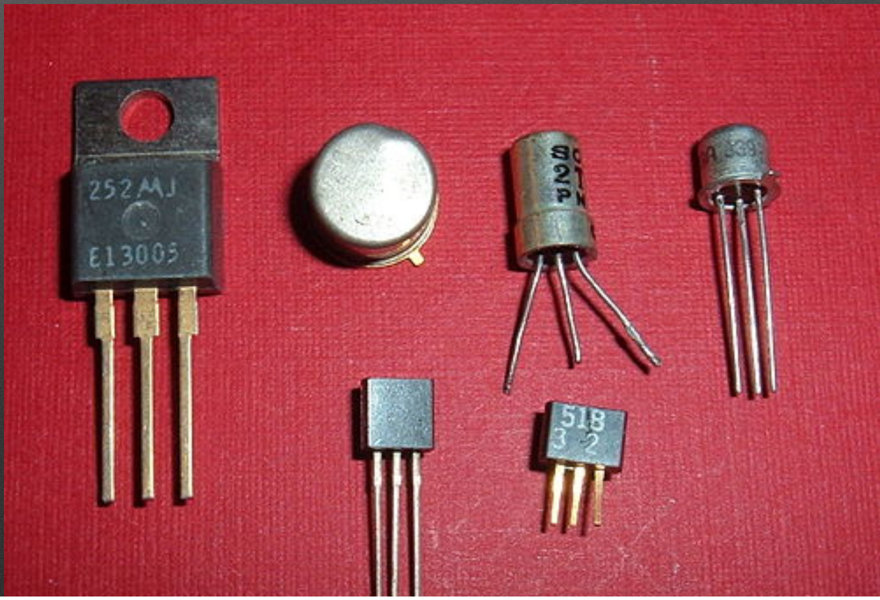
- Today: Review
- Hw 7A completion / Demos
- Exam 2 on Thursday @ classtime
- Optional Extra Credit Homework Posted! (Draft: Additional video tonight)
  - Up to 2% extra credit: Compensate for homework, studio, prep.
  - Due by Tuesday, April 29th

Studio 7B Review

Course Review
Digital Logic: At the heart of modern machines…

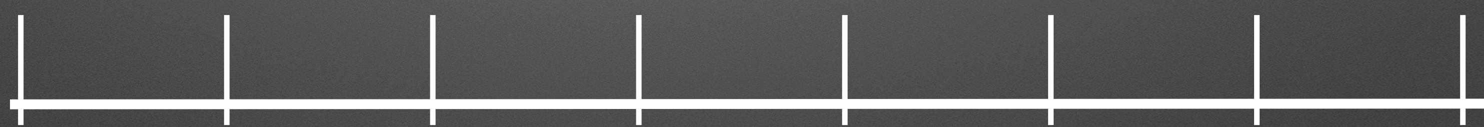
Back to Basics: Binary

- Binary: (0/1; false/true; Off/On; 0v/3v; No/Yes; …)
    - Why? Easy and cheap to build *reliable, fast* machines



Back to Basics: Binary

- Can encode info
  - Can have one-to-one correspondence to number lines



Decimal:	0	1	2	3	4	5	6	7
Binary:	000	001	010	011	100	101	110	111

- Also easy to convert between convenient forms:
  Binary, Decimal, Hexadecimal

Back to Basics: Binary

- Behaving like negatives is easy (with fixed-width numbers)

Decimal:	0	1	2	3	4	5	6	7
Binary:	000	001	010	011	100	101	110	111

$$n+7 == n-1$$

$$(n+8 = n \text{ for 3-bit numbers})$$

The Magic of Fixed Width numbers (modular arithmetic): Addition can emulate subtraction!

Decimal:	0	1	2	3	4	5	6	7
Binary:	000	001	010	011	100	101	110	111
2's comp behavior:					-4	—3	-2	-1

Back to Basics: Binary

- Can encode other discrete/enumerable values too

Table B.4 Register names and numbers

Name	Register Number	Use
zero	x0	Constant value 0
ra	x1	Return address
sp	x2	Stack pointer
gp	x3	Global pointer
tp	x4	Thread pointer
t0–2	x5–7	Temporary registers
s0/fp	x8	Saved register / Frame pointer
s1	x9	Saved register
a0–1	x10–11	Function arguments / Return values
a2–7	x12–17	Function arguments
s2–11	x18–27	Saved registers
t3-6	x28–31	Temporary registers

31:25		24:20	19:15	14:12	11:7	6:0	
funct7		rs2	rs1	funct3	rd	op	R-Type
$imm_{11:0}$			rs1	funct3	rd	op	I-Type
$imm_{11:5}$		rs2	rs1	funct3	$imm_{4:0}$	op	S-Type
$imm_{12,10:5}$		rs2	rs1	funct3	$imm_{4:1,11}$	op	B-Type
$imm_{31:12}$					rd	op	U-Type
$imm_{20,10:1,11,19:12}$					rd	op	J-Type
fs3	funct2	fs2	fs1	funct3	fd	op	R4-Type
5 bits	2 bits	5 bits	5 bits	3 bits	5 bits	7 bits	

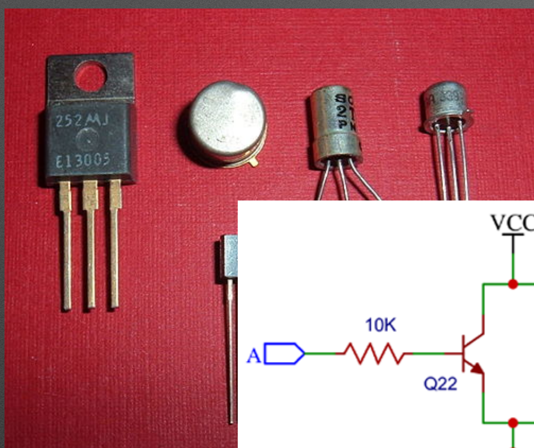
Figure B.1 RISC-V 32-bit instruction formats

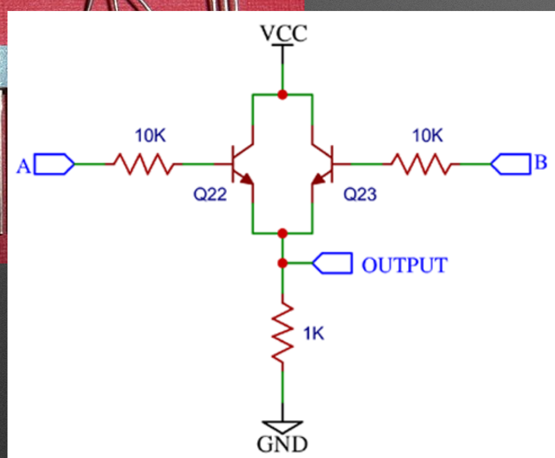
Back to Basics: Binary

- Including “states”:
  - Ex: Idle, Wash, Rinse, Dry

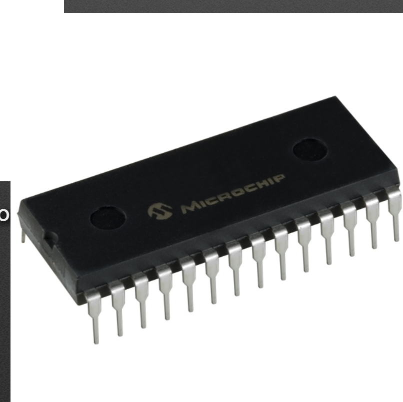
STATE	BINARY COUNTING ENCODING	ANOTHER BIN. ENC.	A ONE HOT
IDLE	00	10	0001
WASH	01	11	0010
RINSE	10	01	0100
DRY	11	00	1000

Assume we can build basic blocks…

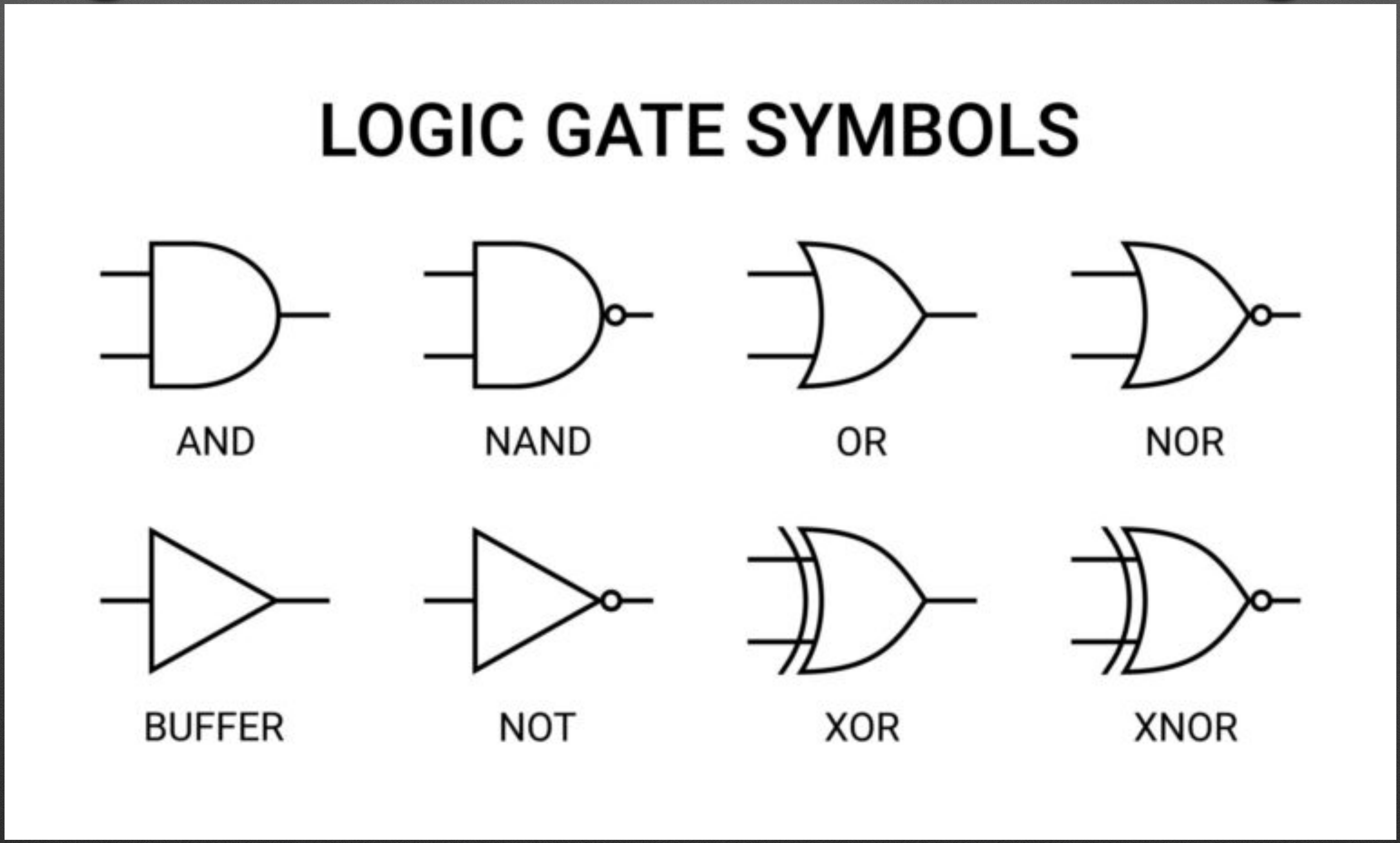




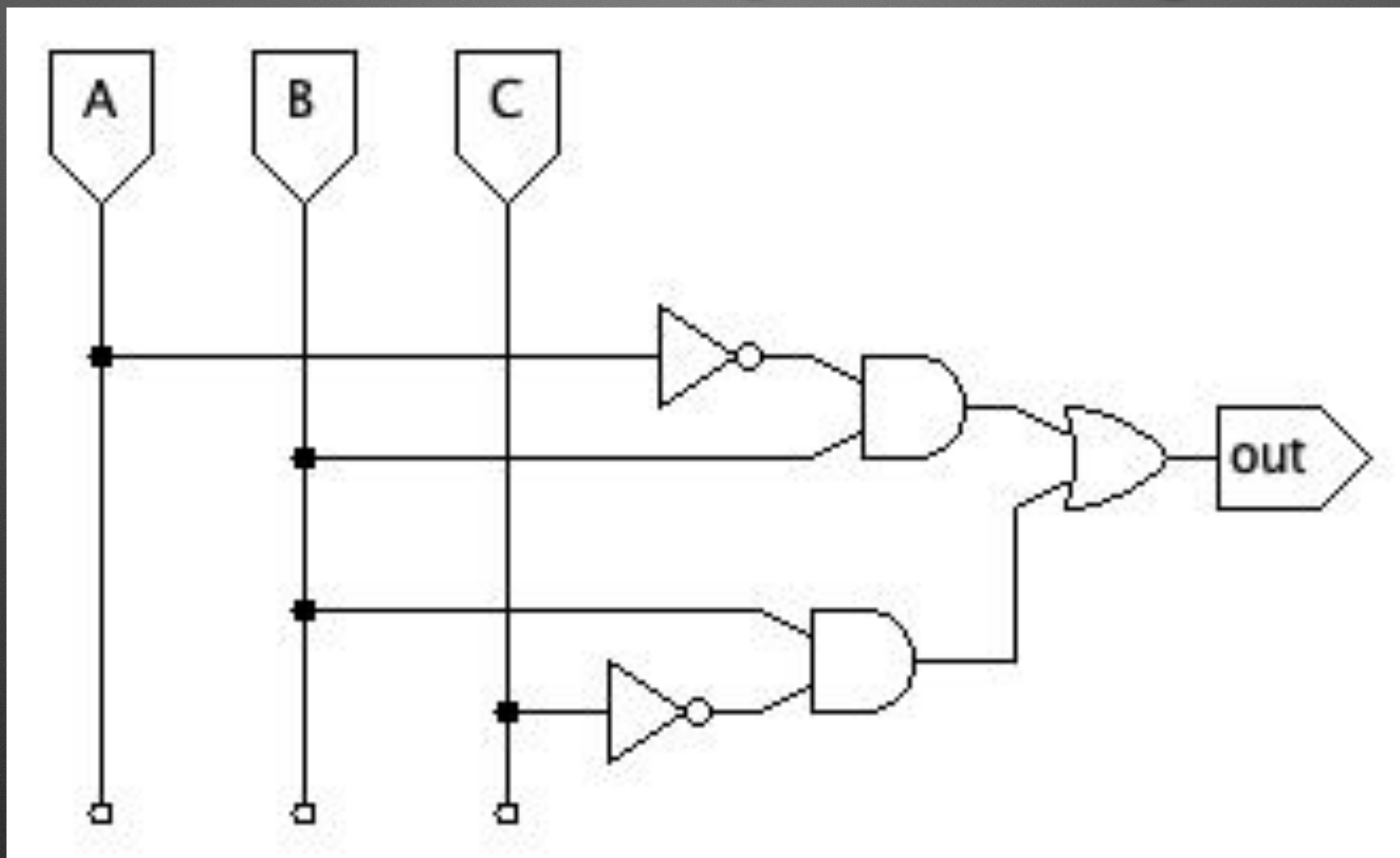
https://circuitdigest.com/electronic-circuits/designing-o



Big Idea: Schematics for Designs



Can build complex things…



Brings some new ideas

- Can be represented (and manipulated)
  - Boolean Algebra
  - Truth table
- Implementation operates in real-world and takes time
  - I.e., Propagation Delay

$$out = \overline{A} \cdot B + B \cdot \overline{C}$$

A	B	C	OUT
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

Brings some new ideas

- Truth table can be converted to equations and vice versa

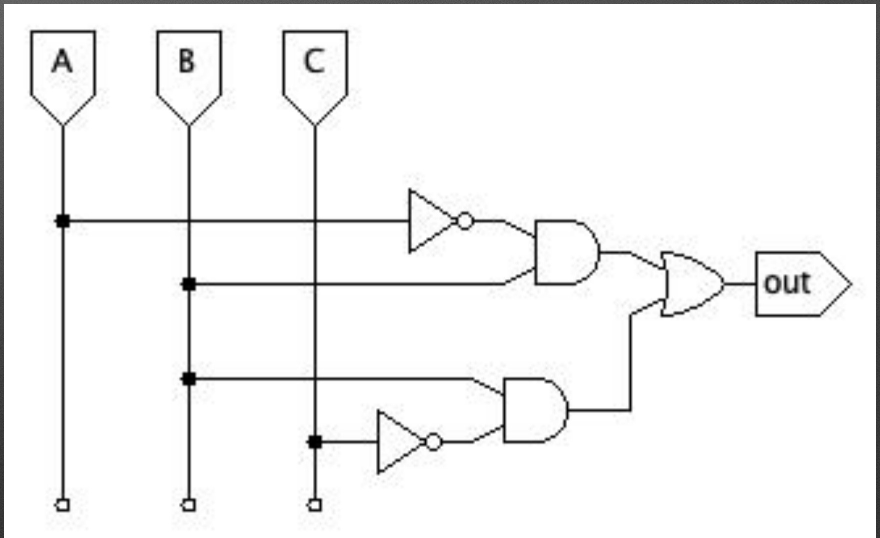
$$out = \overline{A} \cdot B + B \cdot \overline{C}$$

A	B	C	OUT
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

Brings some new ideas

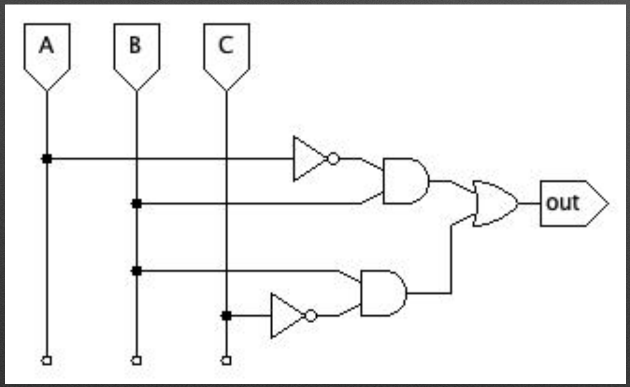
- Equations can be converted to schematics too

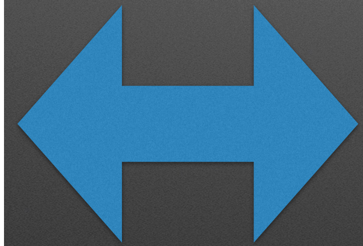
$$out = \overline{A} \cdot B + B \cdot \overline{C}$$

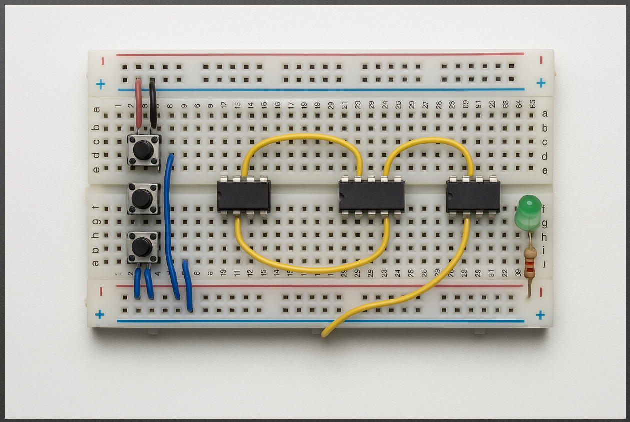


Brings some new ideas

- And machines can be build from schematics…







ChatGPT Generated

Concepts

- Equations can be manipulated via formal rules

Table 2.1 Axioms of Boolean algebra

	Axiom		Dual	Name
A1	$B = 0$ if $B \neq 1$	A1′	$B = 1$ if $B \neq 0$	Binary field
A2	$\overline{0} = 1$	A2′	$\overline{1} = 0$	NOT
A3	$0 \bullet 0 = 0$	A3′	$1 + 1 = 1$	AND/OR
A4	$1 \bullet 1 = 1$	A4′	$0 + 0 = 0$	AND/OR
A5	$0 \bullet 1 = 1 \bullet 0 = 0$	A5′	$1 + 0 = 0 + 1 = 1$	AND/OR

Table 2.2 Boolean theorems of one variable

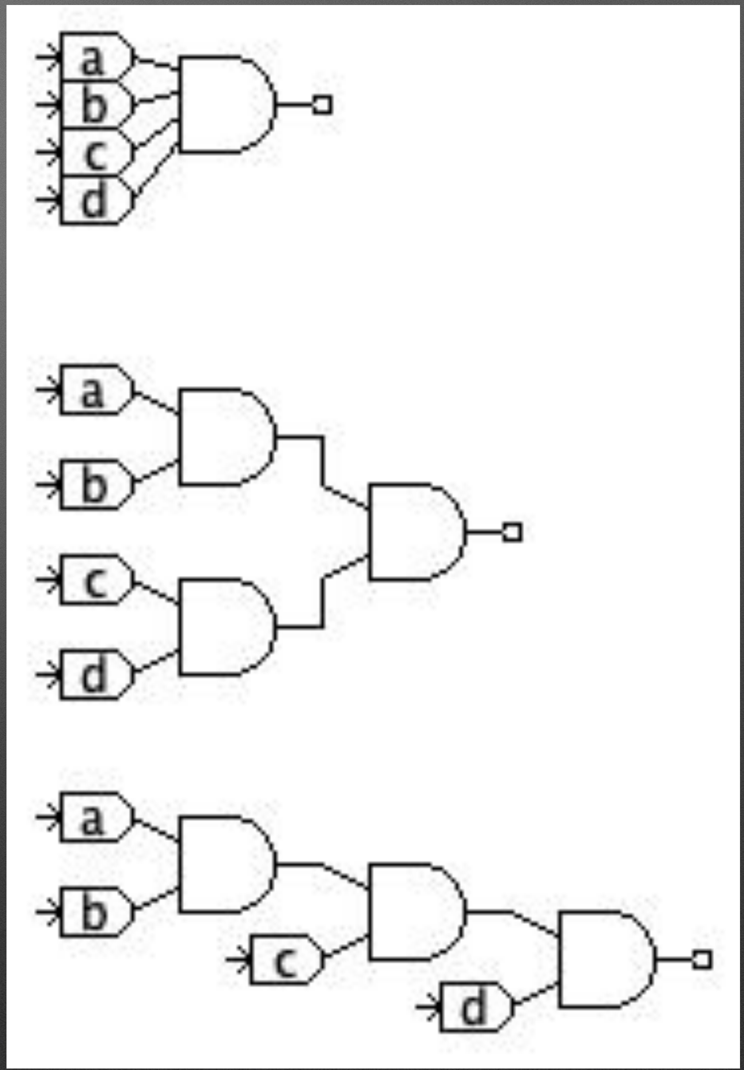
	Theorem		Dual	Name
T1	$B \bullet 1 = B$	T1′	$B + 0 = B$	Identity
T2	$B \bullet 0 = 0$	T2′	$B + 1 = 1$	Null Element
T3	$B \bullet B = B$	T3′	$B + B = B$	Idempotency
T4		$\overline{\overline{B}} = B$		Involution
T5	$B \bullet \overline{B} = 0$	T5′	$B + \overline{B} = 1$	Complements

Table 2.3 Boolean theorems of several variables

	Theorem		Dual	Name
T6	$B \bullet C = C \bullet B$	T6′	$B + C = C + B$	Commutativity
T7	$(B \bullet C) \bullet D = B \bullet (C \bullet D)$	T7′	$(B + C) + D = B + (C + D)$	Associativity
T8	$(B \bullet C) + (B \bullet D) = B \bullet (C + D)$	T8′	$(B + C) \bullet (B + D) = B + (C \bullet D)$	Distributivity
T9	$B \bullet (B + C) = B$	T9′	$B + (B \bullet C) = B$	Covering
T10	$(B \bullet C) + (B \bullet \overline{C}) = B$	T10′	$(B + C) \bullet (B + \overline{C}) = B$	Combining
T11	$(B \bullet C) + (\overline{B} \bullet D) + (C \bullet D)$ $= (B \bullet C) + (\overline{B} \bullet D)$	T11′	$(B + C) \bullet (\overline{B} + D) \bullet (C + D)$ $= (B + C) \bullet (\overline{B} + D)$	Consensus
T12	$\overline{B_0 \bullet B_1 \bullet B_2 \ldots}$ $= (\overline{B}_0 + \overline{B}_1 + \overline{B}_2 \ldots)$	T12′	$\overline{B_0 + B_1 + B_2 \ldots}$ $= (\overline{B}_0 \bullet \overline{B}_1 \bullet \overline{B}_2 \ldots)$	De Morgan's Theorem

Implementation Matters

- $o = a \cdot b \cdot c \cdot d$



Generality

- Truth Table
  - n input variables (n columns of input)
  - 2^n rows: represent all possible combos of input values
  - k columns, 1 for each output bit
- Any such truth table can be converted into an equation (and circuit)
  - Sum-of-products (or product-of-sum): Fool proof (kinda), but may be inefficient

Simple Optimization

- Karnaugh Maps
  - Use “Gray code order” (not counting order)
  - Visual way to combine terms like: $A \cdot \overline{B} \cdot C + A \cdot B \cdot C$
    - Adjacent cells represent a change in a single variable
    - Theorem:

T10	$(B \bullet C) + (B \bullet \overline{C}) = B$

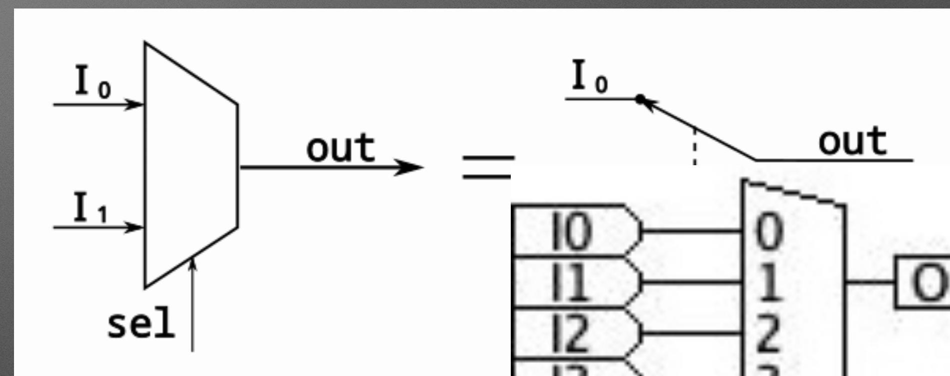
A changes

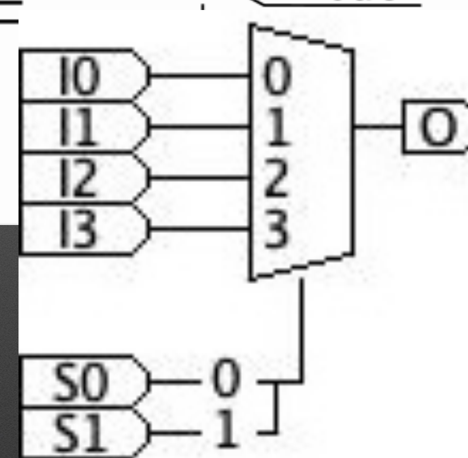
		A B			
		00	01	11	10
C	0	0	1	1	0
	1	0	1	0	0

C changes

Combinational Logic

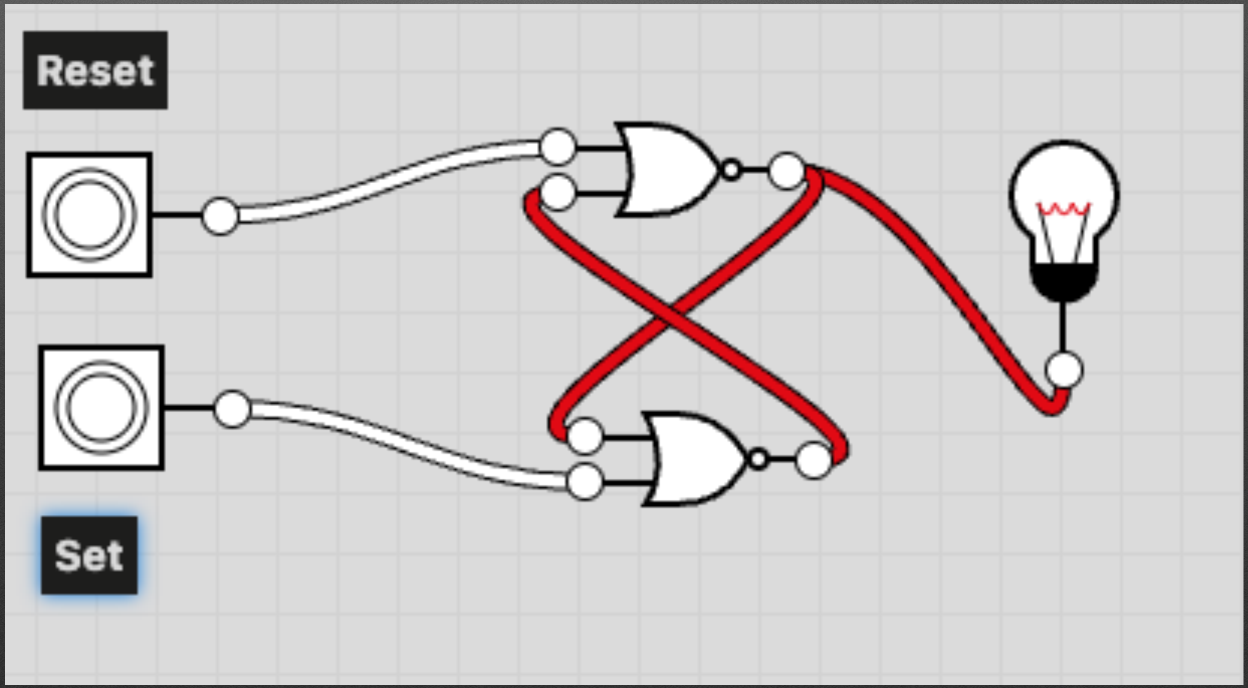
- Can be represented by tables
  - Combine inputs to produce output
  - No concept of memory
- Can represent bigger ideas
  - Selection (multiplexor)
    - May be hierarchical (rather than Sum-of-Products)
  - Encoding / Decoding (2^n inputs to n outputs or n inputs to 2^n outputs)
  - Addition/subtraction, even multi-bit





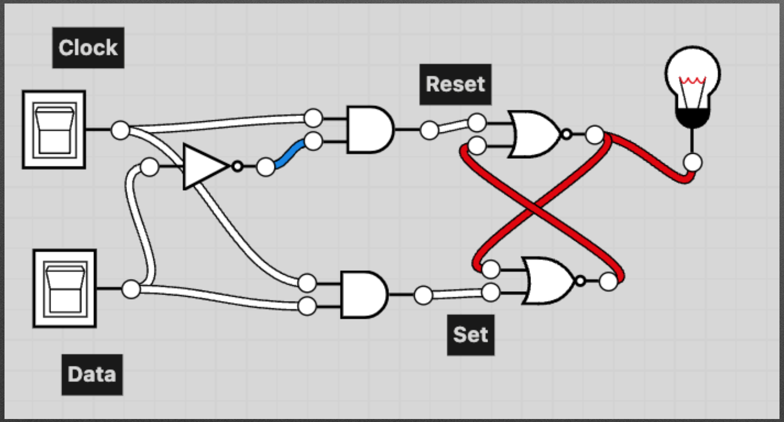
Sequential Stuff & Feedback Paths

- Basic feedback leads to stable storage / memory



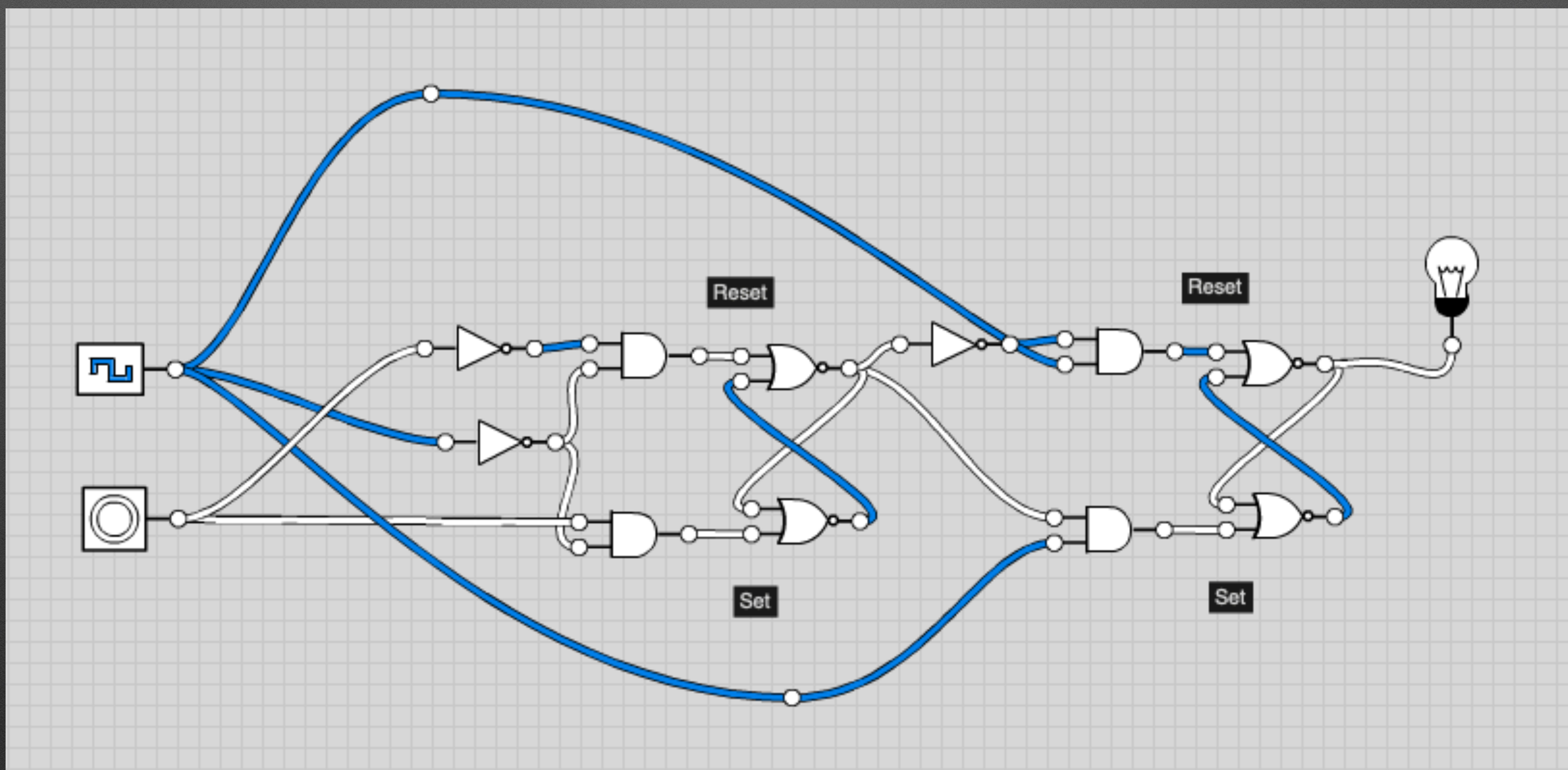
Can combine ideas to shape behavior

D Latch (transparent when Clock high)

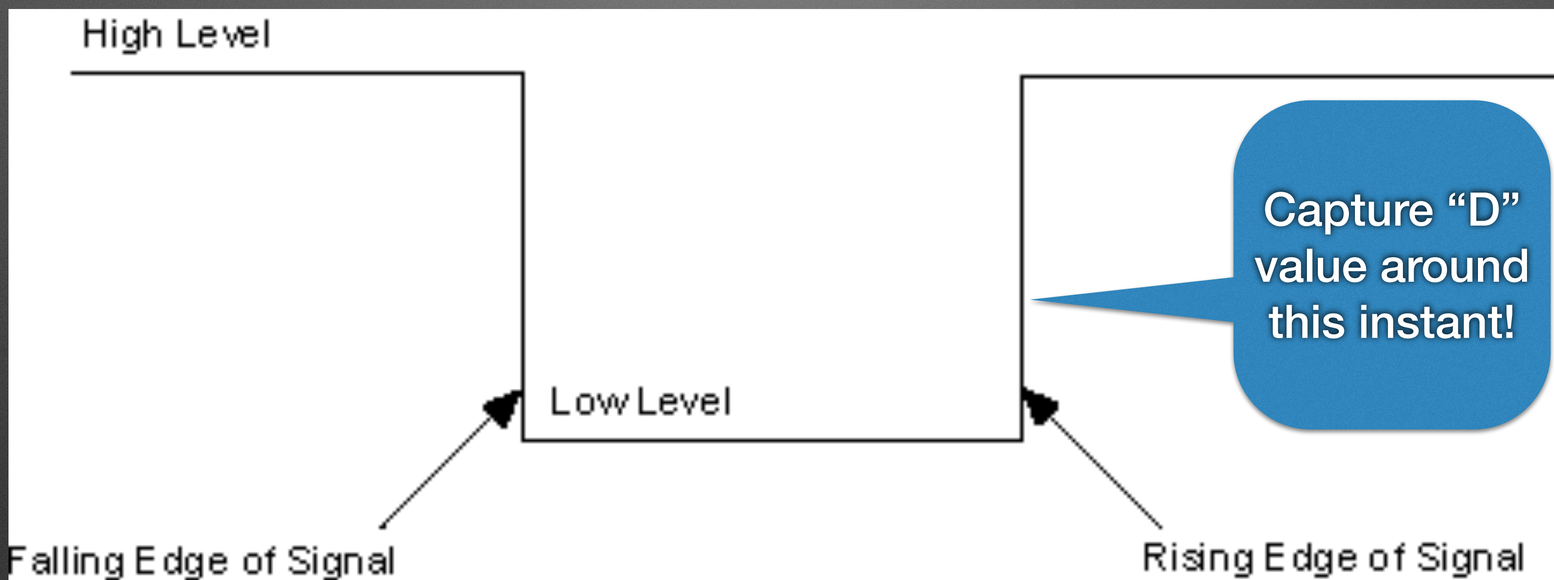


CLOCK	DATA	Q
0	0	(Unchanged)
0	1	(Unchanged)
1	0	0
1	1	1

D Flip-Flop



D-Flip-Flop Clock

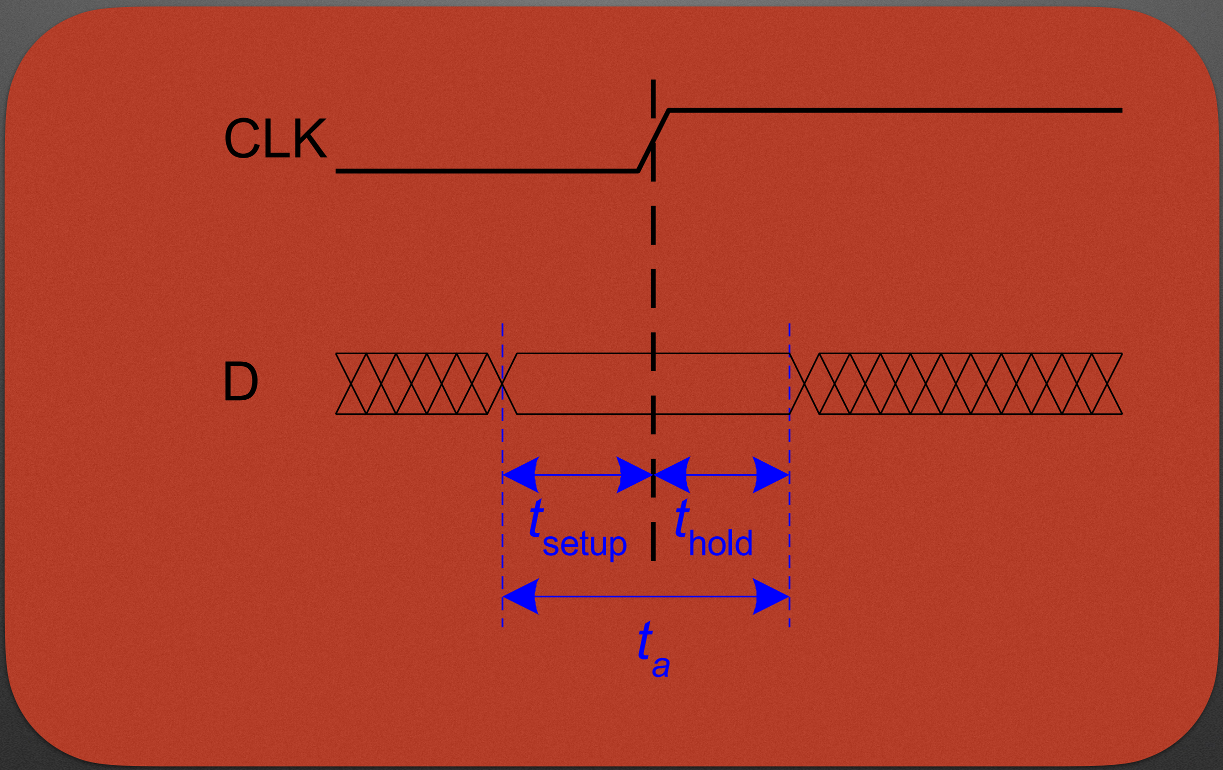


https://www.ni.com/docs/en-US/bundle/ni-hsdio/page/hsdio/fedge_trigger.html

D Flip Flop is built from SR Latches

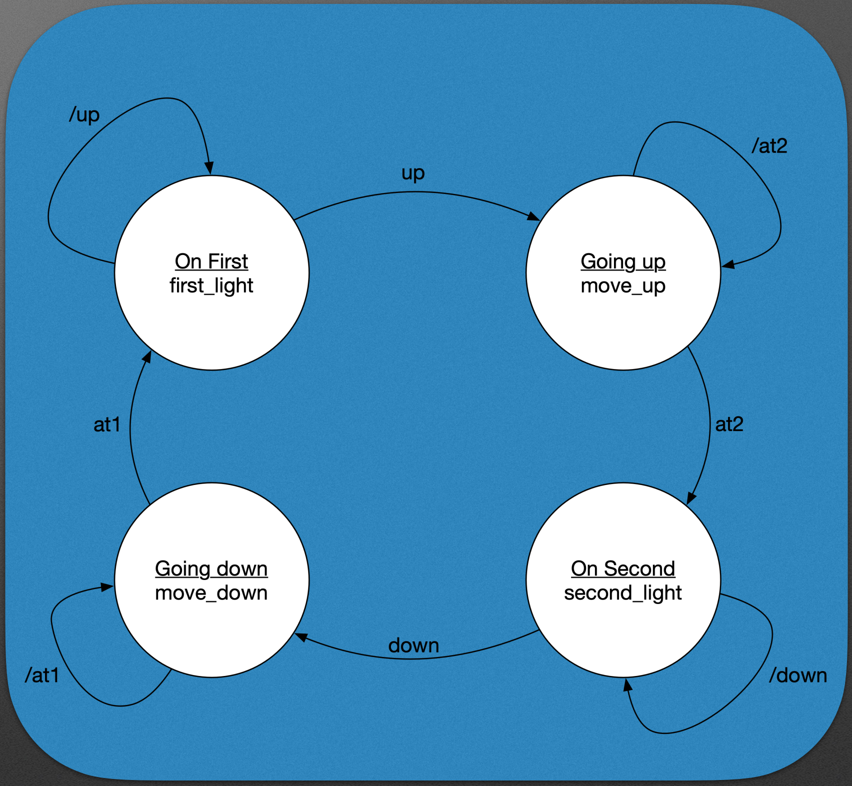
- Internal latch
  - Fails if pulses too short
  - Unstable if R/S drop at same time / too close
- Setup Time: Time needed before clock to ensure stable capture
- Hold Time: Time After clock edge value must be “held”

Dff: Setup & Hold Time

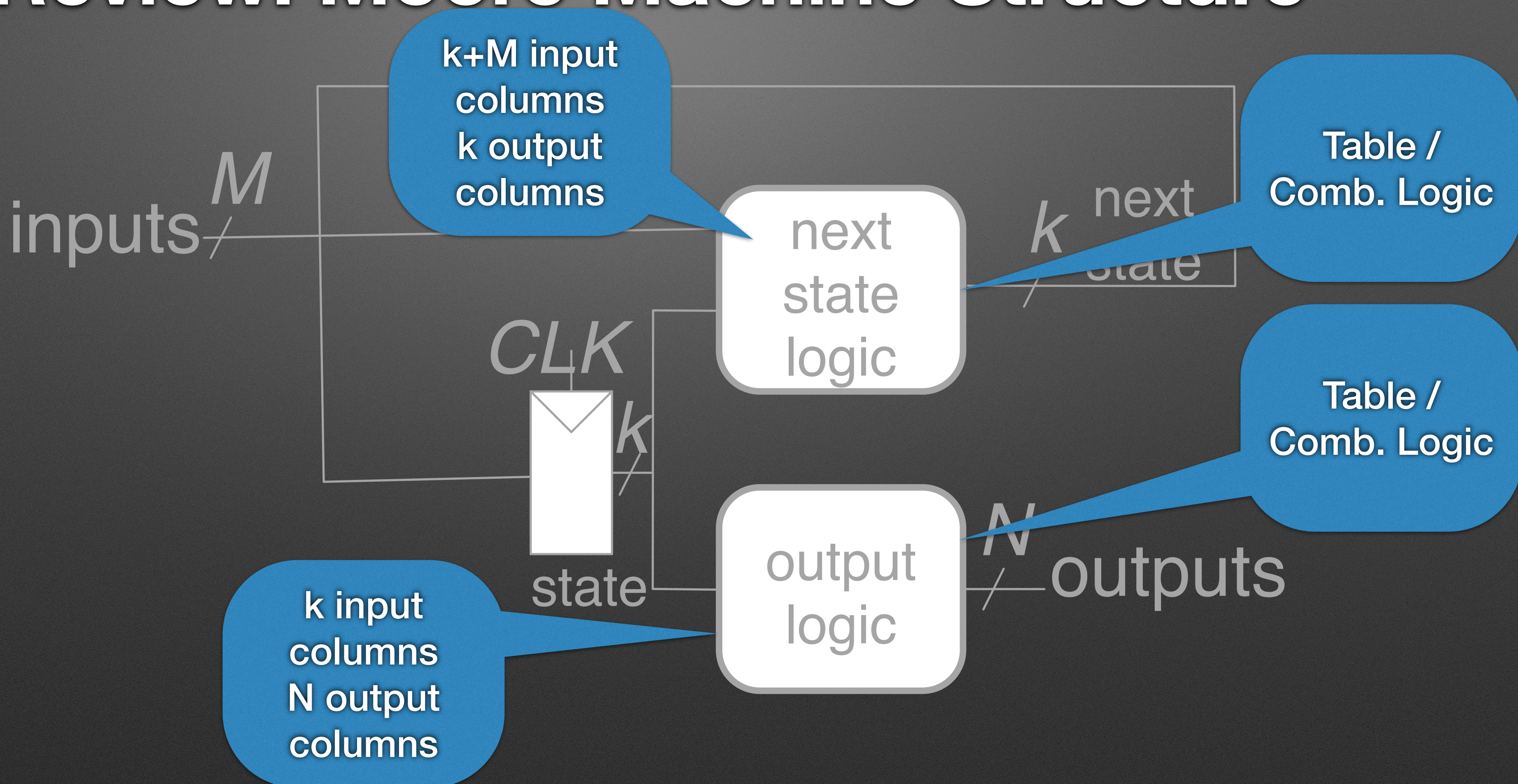


State Machine Diagram

- State: Condition of system
  - Inputs: Used by arcs / outputs
  - Arrows/arcs: When/why state changes
  - Outputs: Actions in the world...



Review: Moore Machine Structure



Chapter 4-7: Exam 2 Focus

(System) Verilog

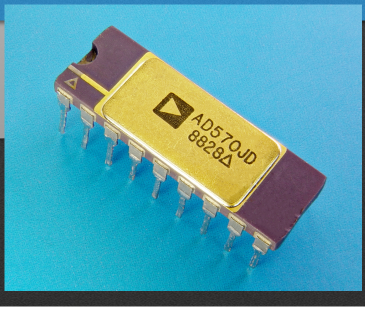
- Developed in 1984 by Gateway Design Automation
- IEEE standard (1364) in 1995; Extended in 2005 (IEEE STD 1800-2009)
- HDL
  - A HDL is not a computer program.
  - A HDL is *not* a computer program!
  - A HDL is *not* a computer program!
  - A HDL is ***NOT*** a computer program!

(System) Verilog Module Example

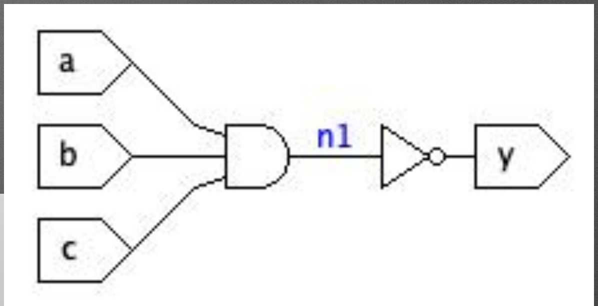
(System) Verilog Module Example


```
module example(input  logic a, b, c,
               output logic y);
  // module body goes here
endmodule
```

Input & Output are like the Pins On chips or in JLS



HDL: ***Structural*** (Verilog)




```
module nand3(input  logic a, b, c
             output logic y);
  logic n1;                        // internal signal

  and  my_and(n1, a, b, c);       // instance of and3
  not  my_inverter(y,n1);         // instance of inv
endmodule
```

Operator Precedence

~	NOT
*, /, %	mult, div, mod
+, -	add, sub
<<, >>	shift
<<<, >>>	arithmetic shift
<, <=,	comparison
==, !=	equal, not equal
&, ~&	AND, NAND
^, ~^	XOR, XNOR
|, ~|	OR, NOR
?:	ternary

Number Formats

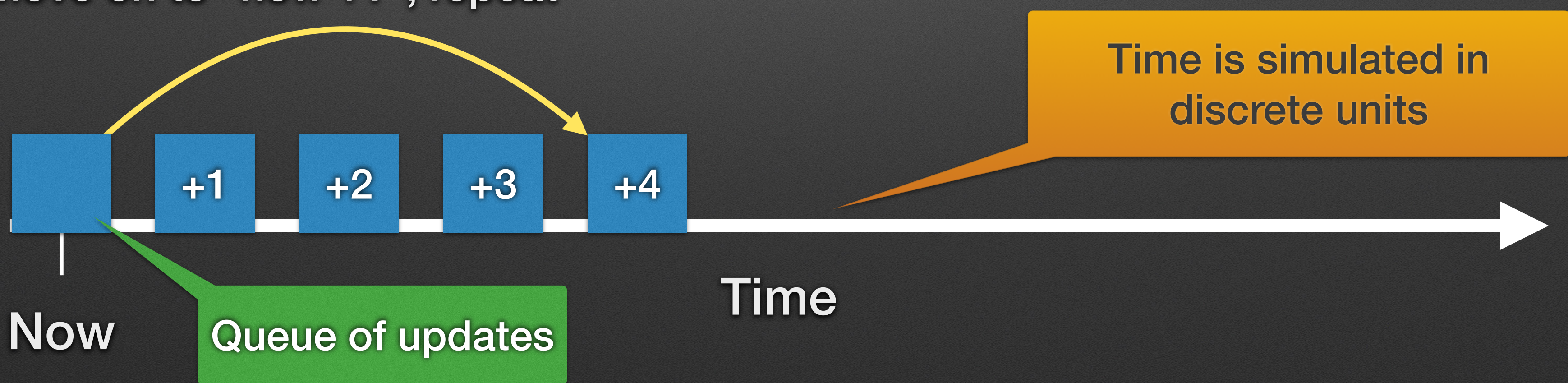
Number	# Bits	Base	Decimal	Stored
3'b101	3	binary	5	101
'b11	unsized	binary	3	00…0011
8'b11	8	binary	3	00000011
8'b1010_1011	8	binary	171	10101011
3'd6	3	decimal	6	110
6'o42	6	octal	34	100010
8'hAB	8	hexadecimal	171	10101011
42	Unsized	decimal	42	00…0101010

`always`: **Based on *Events***

- Concept of “event” is related to simulation and “event driven programming”
- JLS uses events: An OR gate “reacts” to events and schedules an update
  See here

Discrete Time Event Simulator

- Computes all activities / updates for “now”
  - They cause new activities that need to be handled in the future (at: now + prop delay). Those are put in a queue at for that time. Ex: Update an or-gate’s output at now+4
- Move on to “now +1”, repeat



Discrete Time Event Simulator

- Updating values in current turn: Incrementally or all at once at end of turn
  - Ex: Assume x is 1 and y is 0
  - Incremental:
    x = 0
    y = x
  - x's final value is 0
    y's final value is 0 too
  - All at once / end of turn
    x <= 0
    y <= x
  - x's final value is 0
    y's final value is <u>1</u>

`always` in 260

- Form 1: Comb logic
```
always_comb
     statement;
```
Use blocking assignment (=)

  - Statement(s) are (complex) combinational logic. Like if/else or case. Updates when any (relevant/used) input changes

- Form 2: Registered (synchronous, synthesize able, sequential) logic
```
always_ff @(sensitivity list)
    statement;
```
  - Often `@(posedge clock)` used

Use non-blocking assignment (<=)

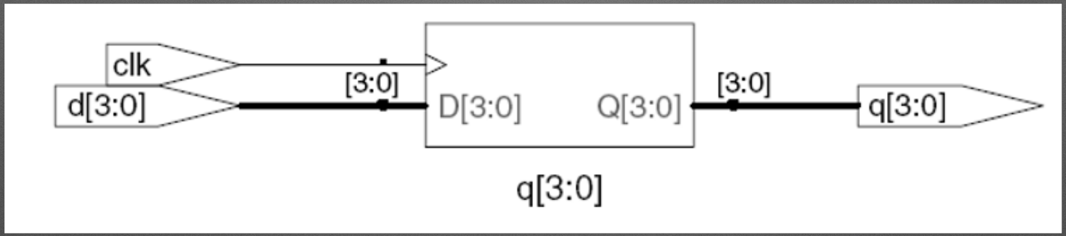
Assignments

- Form 1: Continuous Assignment
  `assign var = expression;`
  - Continuously assigned! Largely a wired connection
- Forms 2 & 3 *in Procedures* (in some form of `always*`) :
  - Blocking (=): Will be "instant" in terms of simulation
    - `always_comb`
  - Non-Blocking (`<=`): Will occur at end of turn all at once
    - `always_ff`

Rules for Assignments

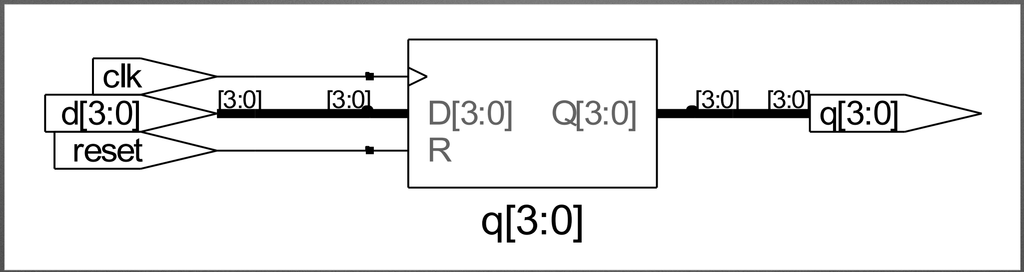
- Synchronous sequential logic
  use `always_ff @(posedge clk)` and nonblocking assignments (<=)
```
always_ff @(posedge clk)
    q <= d; // nonblocking
```
- *Simple* combinational logic
  use continuous assignments (`assign`)
```
assign y = a & b;
```
- Complex Combinational `Logic`
  use `always_comb` and blocking assignments (=)
- Assign signals in only one `always` or `assign` statement!

Verilog: D Flip-Flop



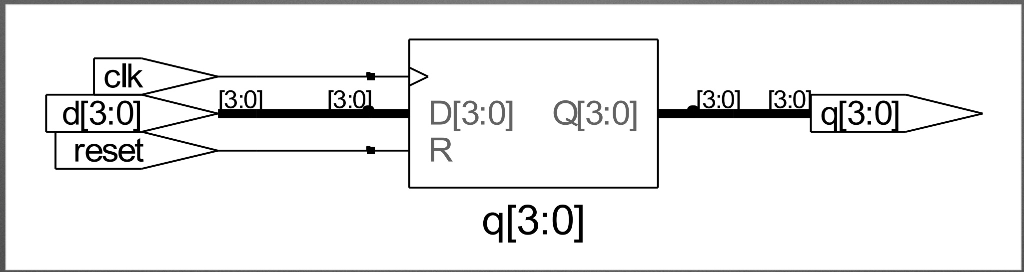

```
module flop(input  logic       clk,
            input  logic [3:0] d,
            output logic [3:0] q);
  always_ff @(posedge clk)
    q <= d;             // pronounced “q gets d”
endmodule
```

Resettable D-Flip-Flop 1



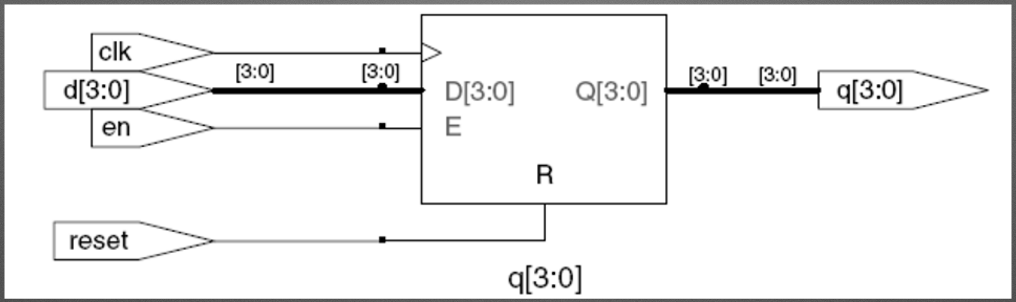

```
module flopr(input  logic       clk,
             input  logic       reset,
             input  logic [3:0] d,
             output logic [3:0] q);

  always_ff @(posedge clk)
    if (reset) q <= 4'b0;
    else       q <= d;
endmodule
```

Resettable D-Flip-Flop 2




```
module flopr(input  logic       clk,
             input  logic       reset,
             input  logic [3:0] d,
             output logic [3:0] q);

  always_ff @(posedge clk, posedge reset)
    if (reset) q <= 4'b0;
    else       q <= d;
endmodule
```

Resettable D-Flip-Flop 3




```
module flopr(input  logic       clk,
             input  logic       reset,
             input  logic       en,
             input  logic [3:0] d,
             output logic [3:0] q);

  always_ff @(posedge clk, posedge reset)
    if (reset)    q <= 4'b0;
    else if (en)  q <= d;
endmodule
```

`always` and Combinational Logic

```
always_comb
   begin
      y = a & b
      …
   end
```

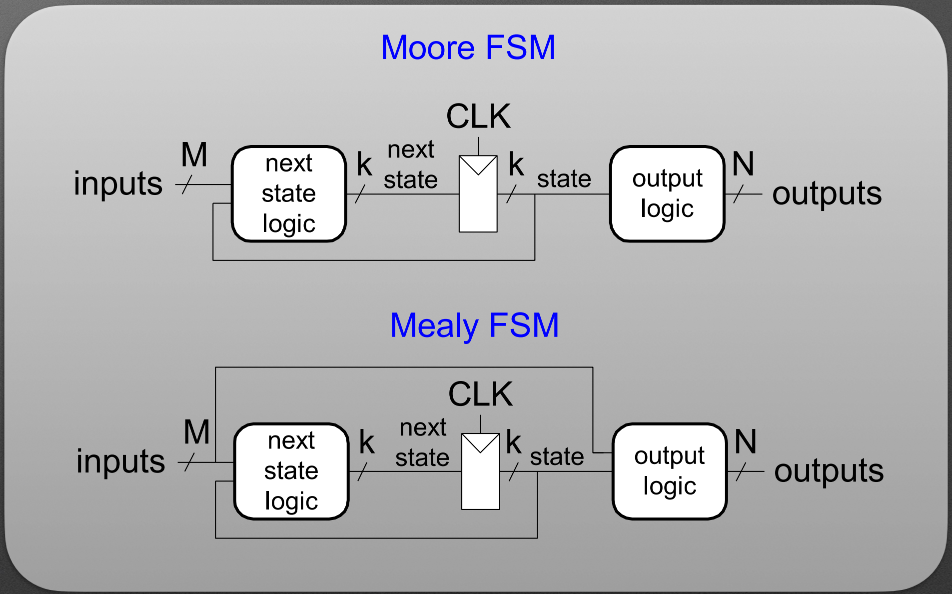
Block of assignments

Could have been done with individual `assigns`

Notice = ("blocking assignment"), not <= ("non-blocking assignment")

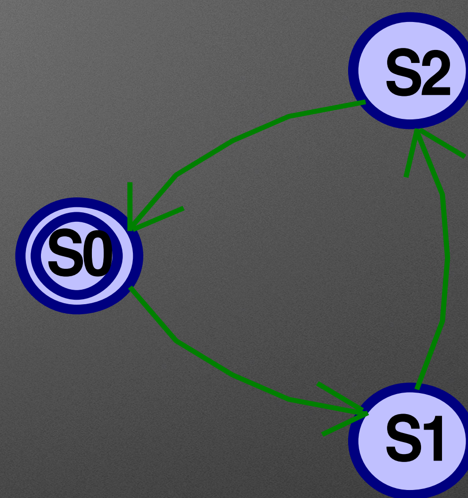
Verilog FSMs

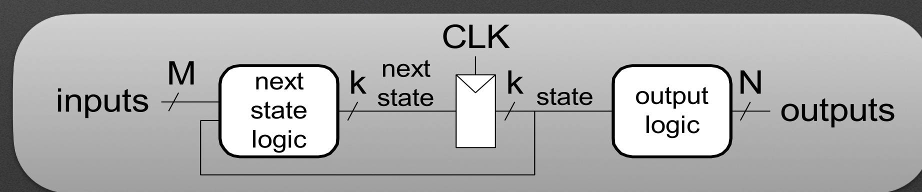
- Three parts
  - Next state logic
    (arrows / next state table)
  - State register (active bubble)
  - Output logic (output equations)



Verilog

```
module divideby3FSM(input  logic clk,
                    input  logic reset,
                    output logic q);

   typedef enum logic [1:0] {S0, S1, S2} statetype;
   statetype state, nextstate;

   // state register
   always_ff @(posedge clk, posedge reset)
      if (reset) state <= S0;
      else       state <= nextstate;

   // next state logic
   always_comb
     case (state)
        S0:      nextstate = S1;
        S1:      nextstate = S2;
        S2:      nextstate = S0;
        default: nextstate = S0;
     endcase

   // output logic
   assign q = (state == S0);
endmodule
```





`always_comb` has nice features

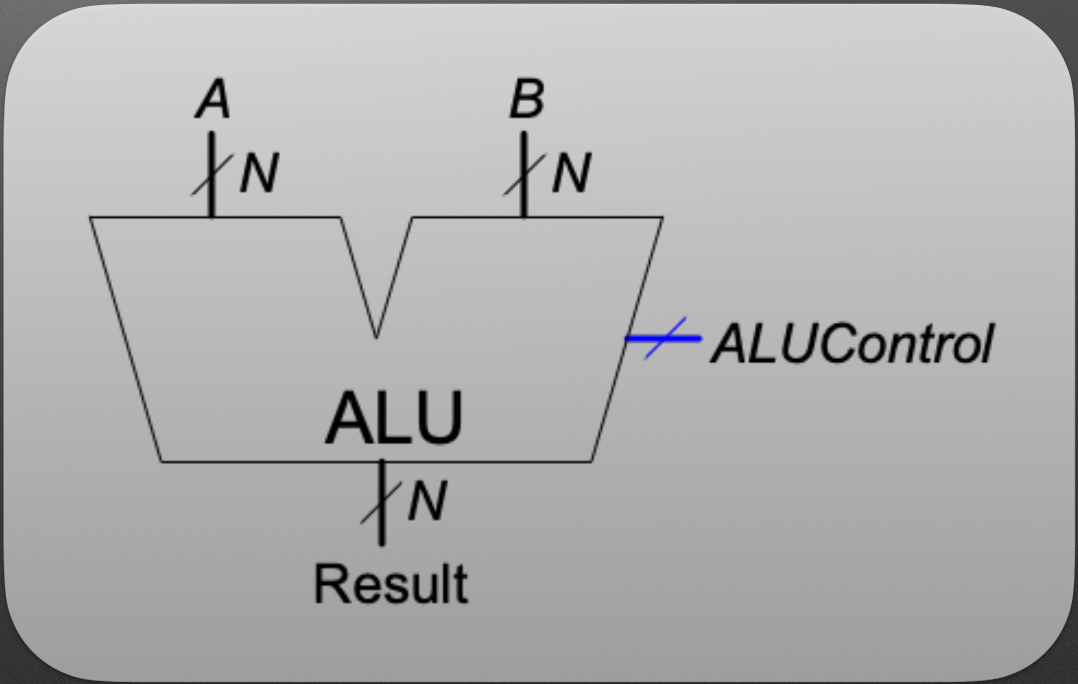
- `case` : Selection between several options
  Great for state machines!

- Must describe all possible combinations to be comb logic. Use `default`

```
case (state)
  soap:                 hot = 1;
  highPressureWarm: hot = 1;
  …
  default: hot = 0;
 endcase
```

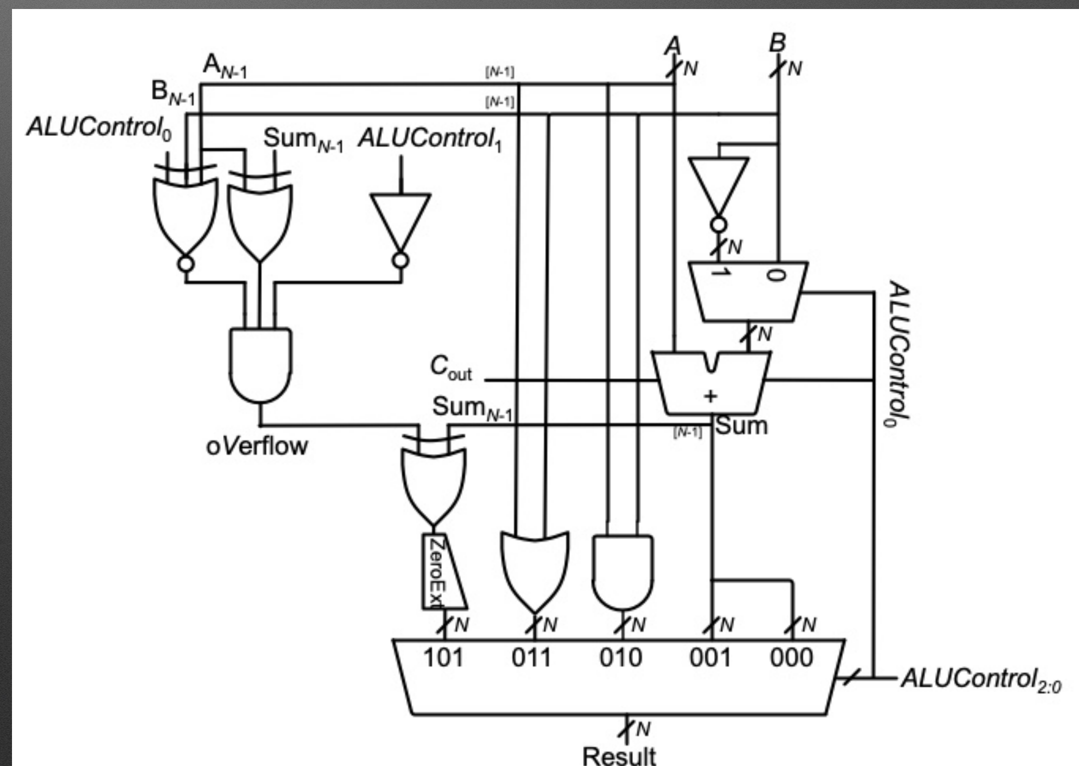
ALU: Arithmetic Logic Unit

- “Heart” of CPU: Does the computation stuff.
  - Basic operations
    - Addition
    - Subtraction
    - Bitwise AND
    - Bitwise OR
    - Comparison (<)



ALU: Arithmetic Logic Unit

- “Heart” of CPU: Does the computation stuff.
  - Basic operations
    - Addition: 000
    - Subtraction: 001
    - Bitwise AND: 010
    - Bitwise OR: 011
    - Comparison (<): 101



Memory / Storage

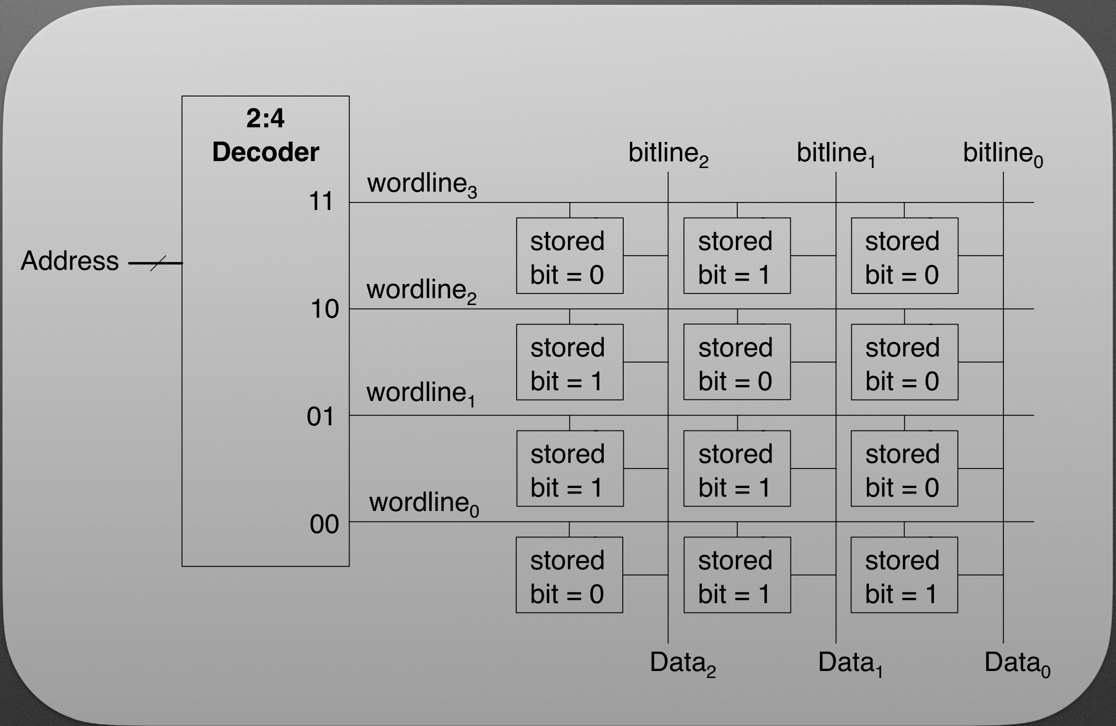
- Common types
  - Static Random Access Memory (SRAM)
  - Dynamic Random Access Memory
  - Read Only Memory (contents can’t be easily changed)

Memory / Storage

- General Approach
  - Store in a 2D grid of elements
  - Call each row a “word”
  - Each row has an index to access the content of the entire row

Memory Structure

- One approach
  - Bits are “enabled” to connect to shared output line

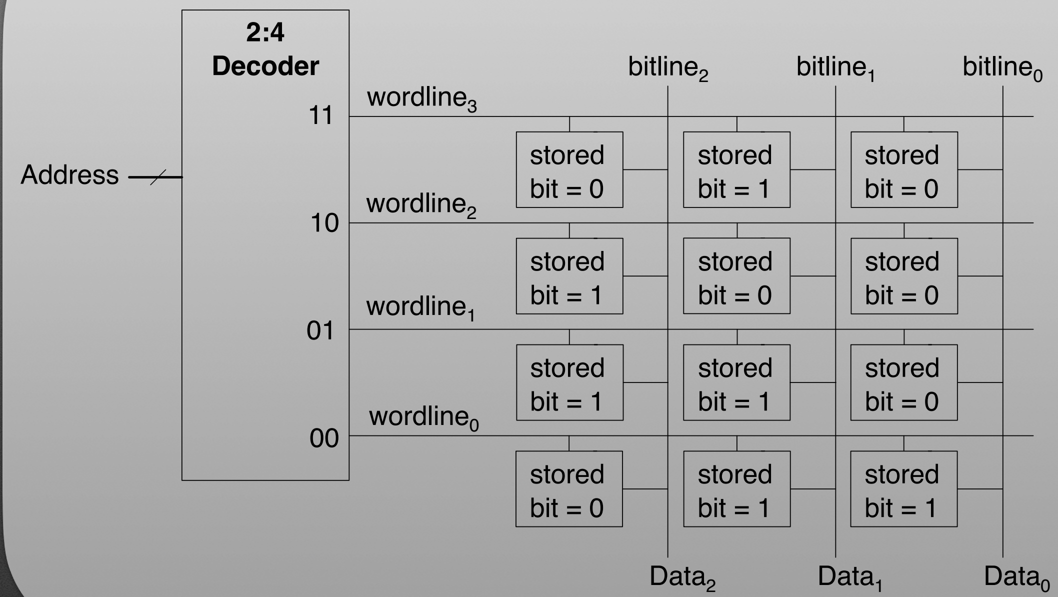


Memory / Storage

- Concept: Computer programming
  - An Array (List) is a representation of memory
- “Random” : Largely about time to access
  - The “random” means the location doesn’t have much impact on access time
  - Vs. “Sequential”

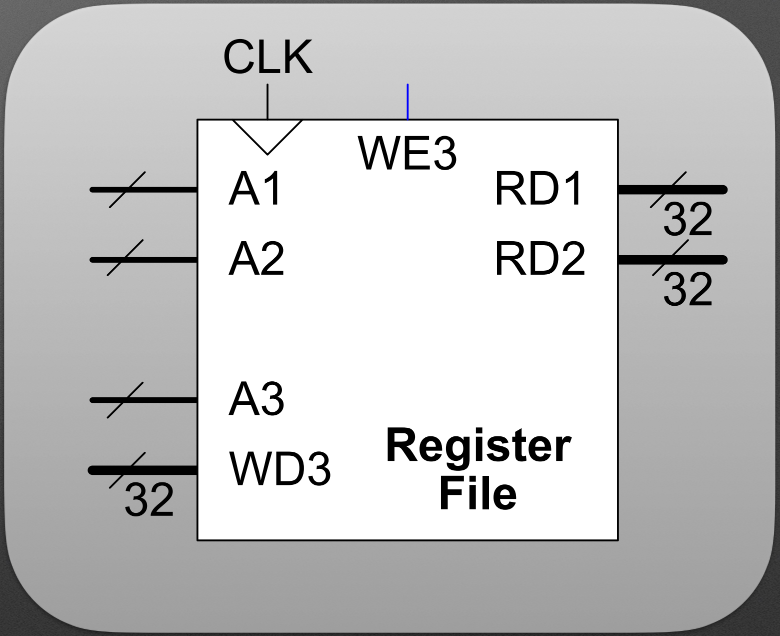
Memory Structure

- One approach
  - Bits are "enabled" to connect to shared output line



ALU Operations

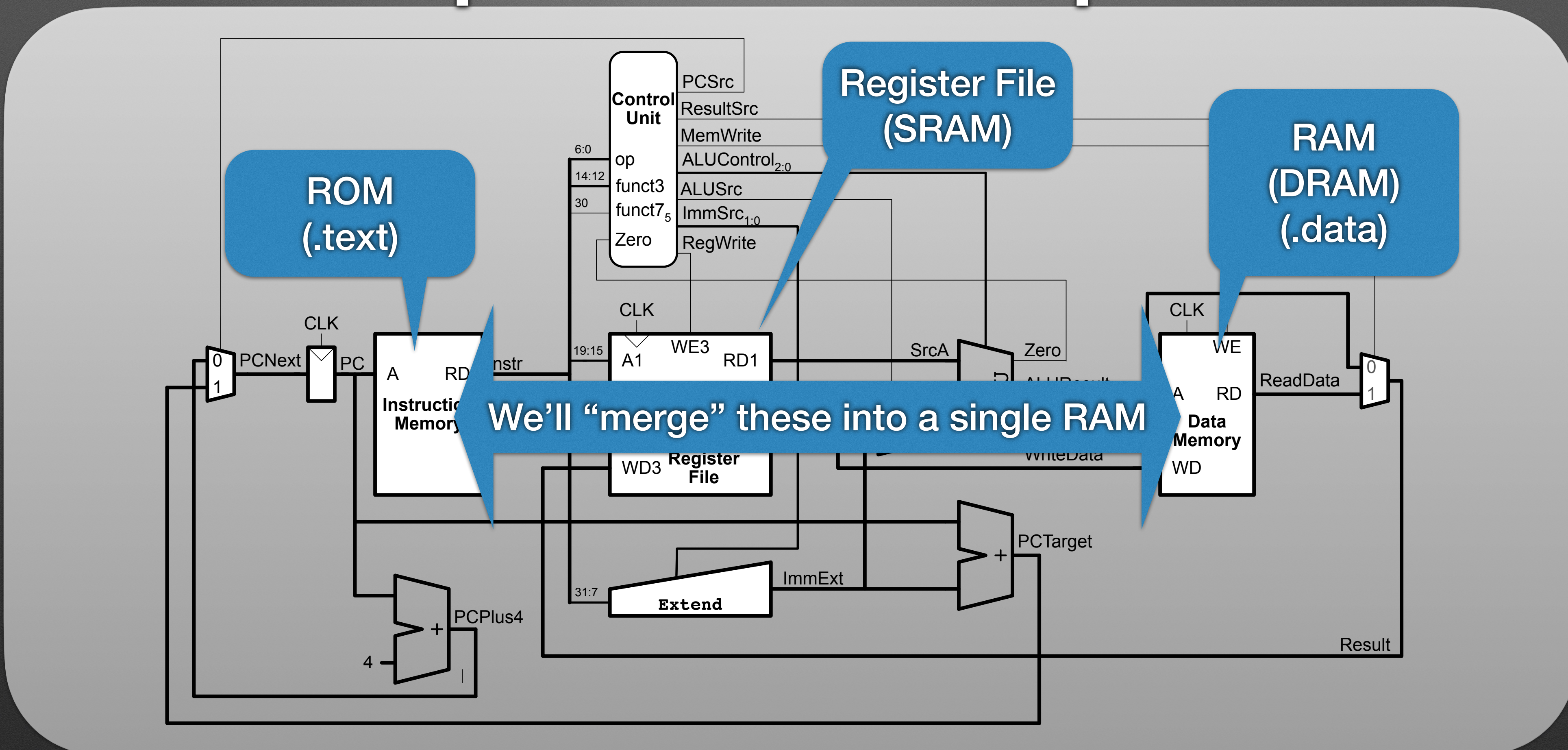
- Context: X=Y+Z
  - We need places to hold Y, Z, and X.
- Need TWO inputs:
  - need a memory structure that provides 2 values (I.e. dual output ports)
- The “Register File”
- Also supports writing (updating)



Verilog: RISC-V Register File

```
// 32 x 32 register file with 2 read, 1 write port
module regfile(input  logic        clk,
               input  logic        we3,
               input  logic [4:0]  ra1, ra2, wa3,
               input  logic [31:0] wd3,
               output logic [31:0] rd1, rd2);

  logic [31:0] rf[31:0];

  always_ff @(posedge clk)
    if (we3) rf[wa3] <= wd3;

  assign rd1 = rf[ra1];
  assign rd2 = rf[ra2];
endmodule
```

Simple RISC-V Computer



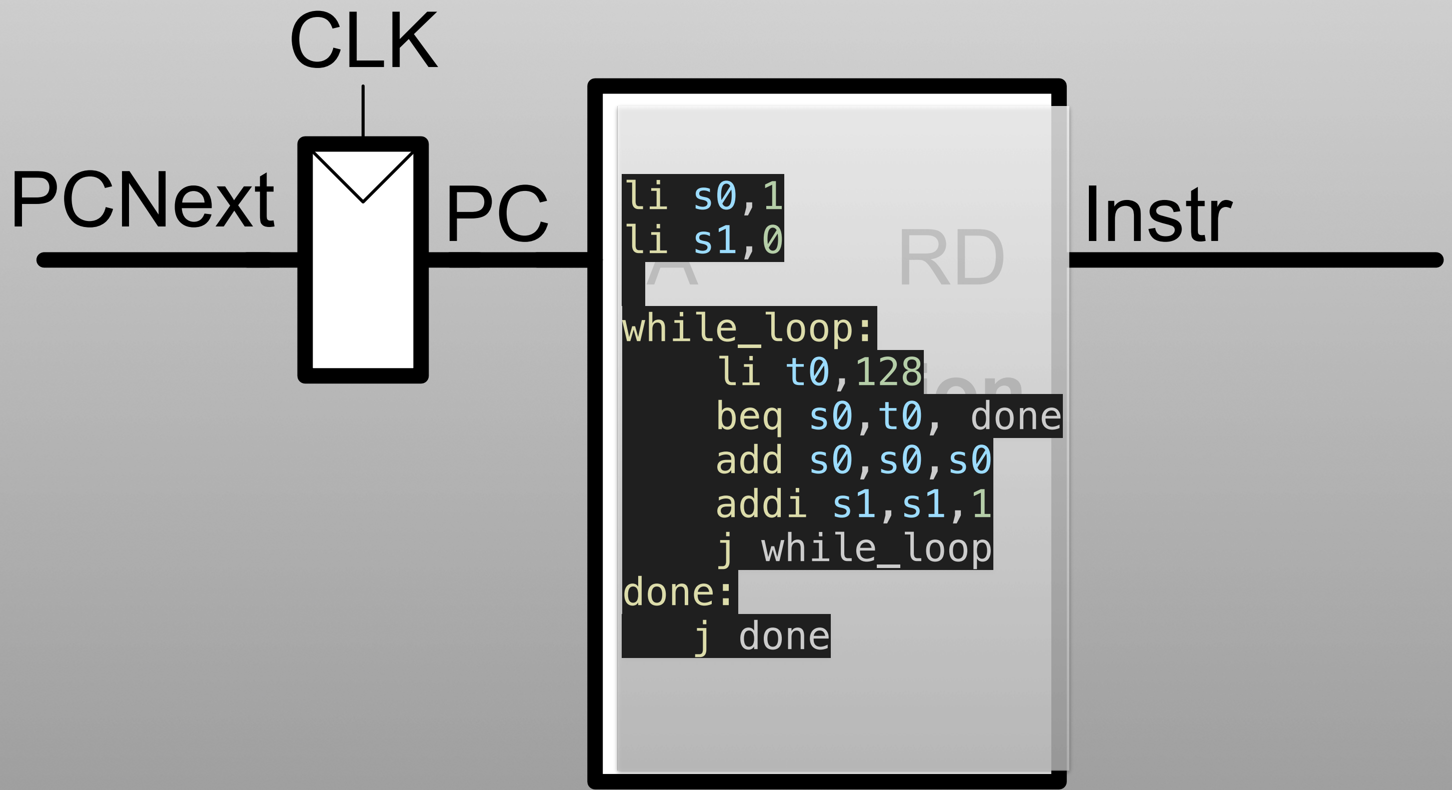
Basic Model

- Machine is basically 2-3 memories + CPU
  - Registers (small, easy to use; temporary/ephemeral)
    - Ex: You have 31, 32-bit data registers = 124 ***Bytes***
  - RAM: Place for most data (Gigabytes!)
  - Program Memory: Possible in RAM or some additional “program memory”

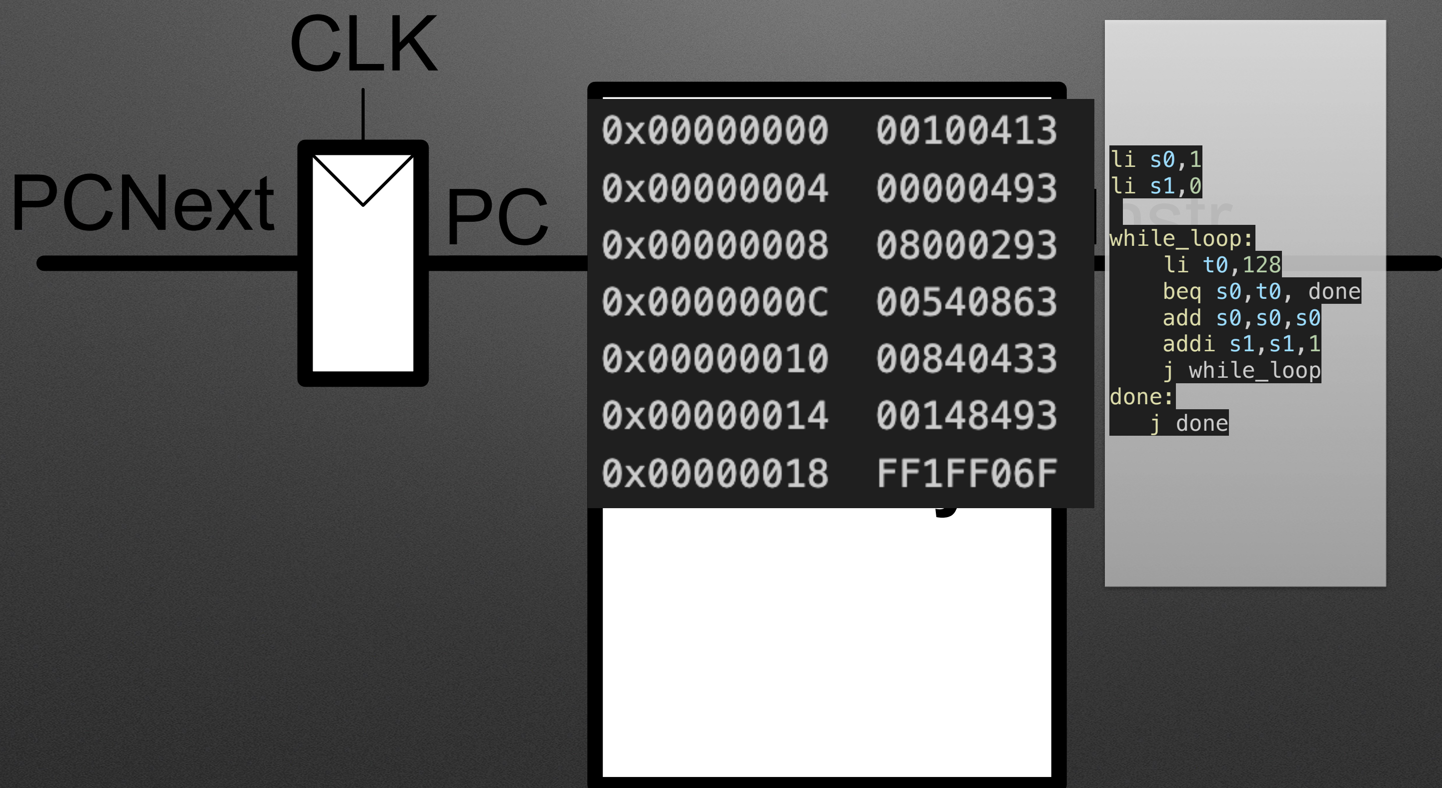
Problem: Find x such that $2^x = 128$

```
// determines the power
// of x such that 2x = 128
int pow = 1;
int x = 0;

while (pow != 128) {
   pow = pow * 2;
   x = x + 1;
}
```

Behavior: Parts of CPU Model



Behavior: Parts of CPU Model



0x00000000	00100413
0x00000004	00000493
0x00000008	08000293
0x0000000C	00540863
0x00000010	00840433
0x00000014	00148493
0x00000018	FF1FF06F

```
li s0,1
li s1,0
while_loop:
    li t0,128
    beq s0,t0, done
    add s0,s0,s0
    addi s1,s1,1
    j while_loop
done:
   j done
```

Exam Qs

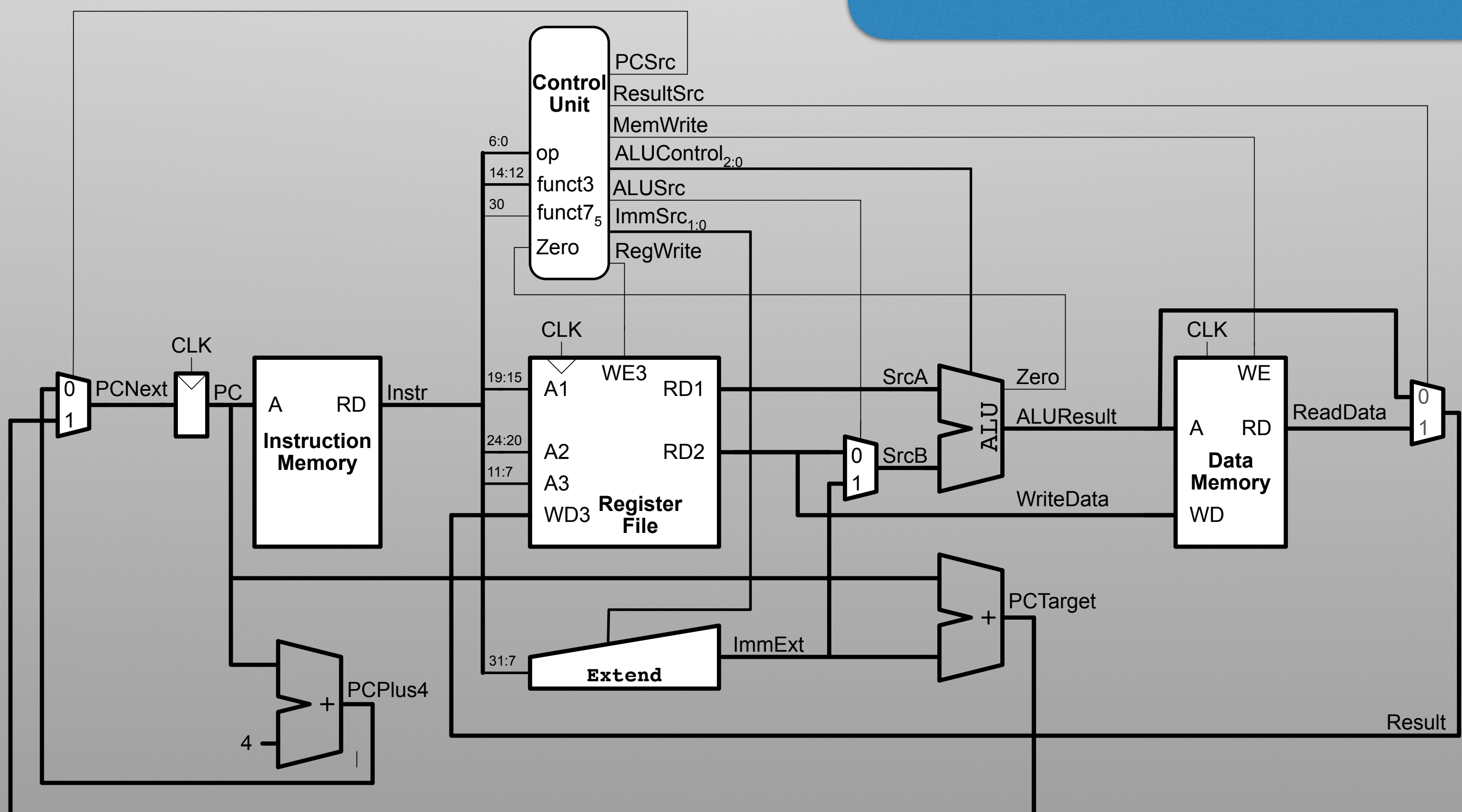
- Given Machine Code:
  - What format of instruction? (J? I? R? Etc.)
  - What is the content of the rd field?
  - What operation is being performed?
- Given Assembly Language (like lw t4, 0(a0))
  - What is the value of the Machine language rd field?

Exam Qs

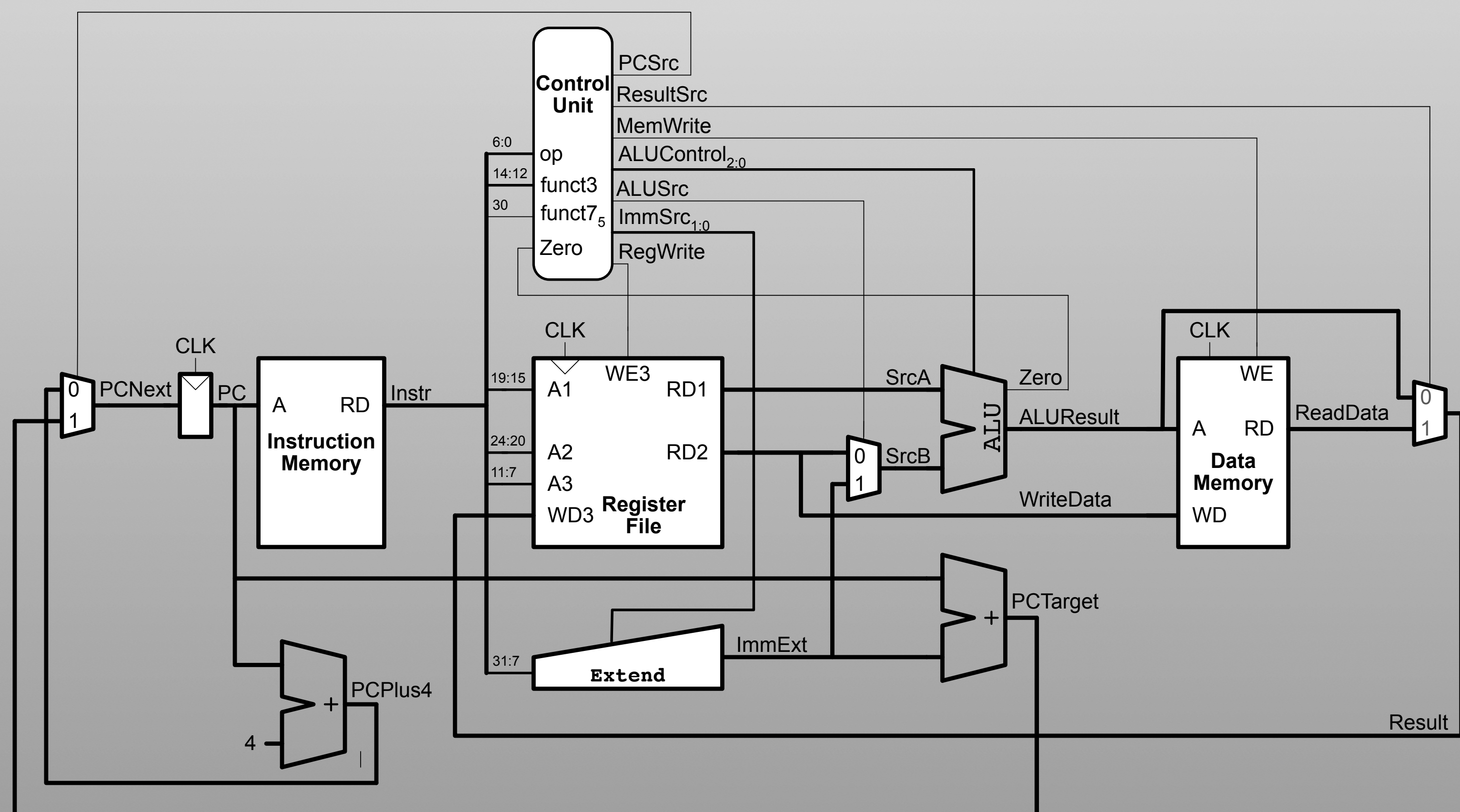
- Given Assembly Language Code (no loop) and initial register values:
  - What will the value of register X be when done?
- Given Assembly Language Code (with loop) and initial register values:
  - What will the value of register X be when done?
  - How many times will instruction Y run?

Simple, Single-Cycle P

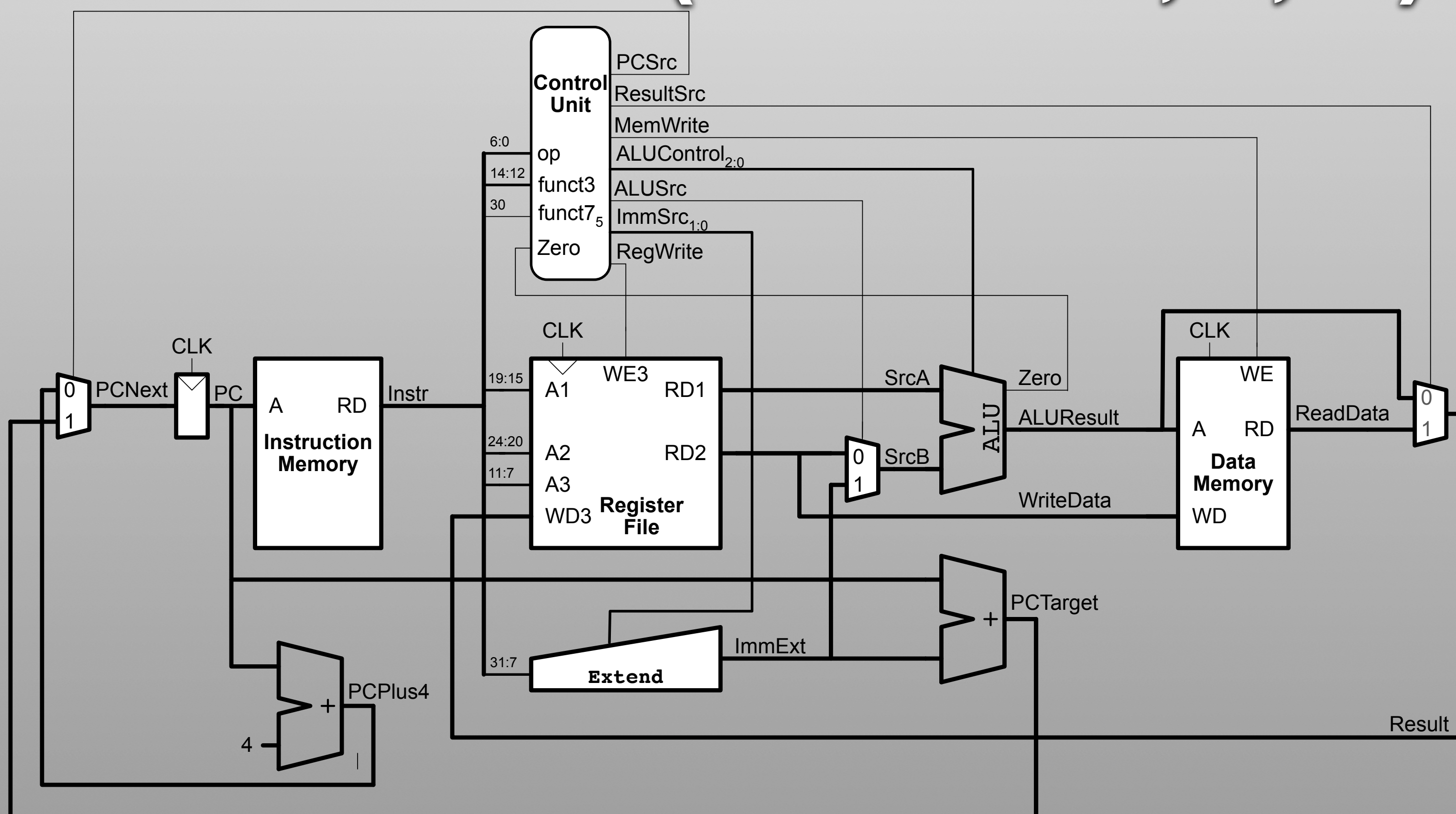
Describe behavior of all elements and any required control signals for
`lw t0,4(a0)`



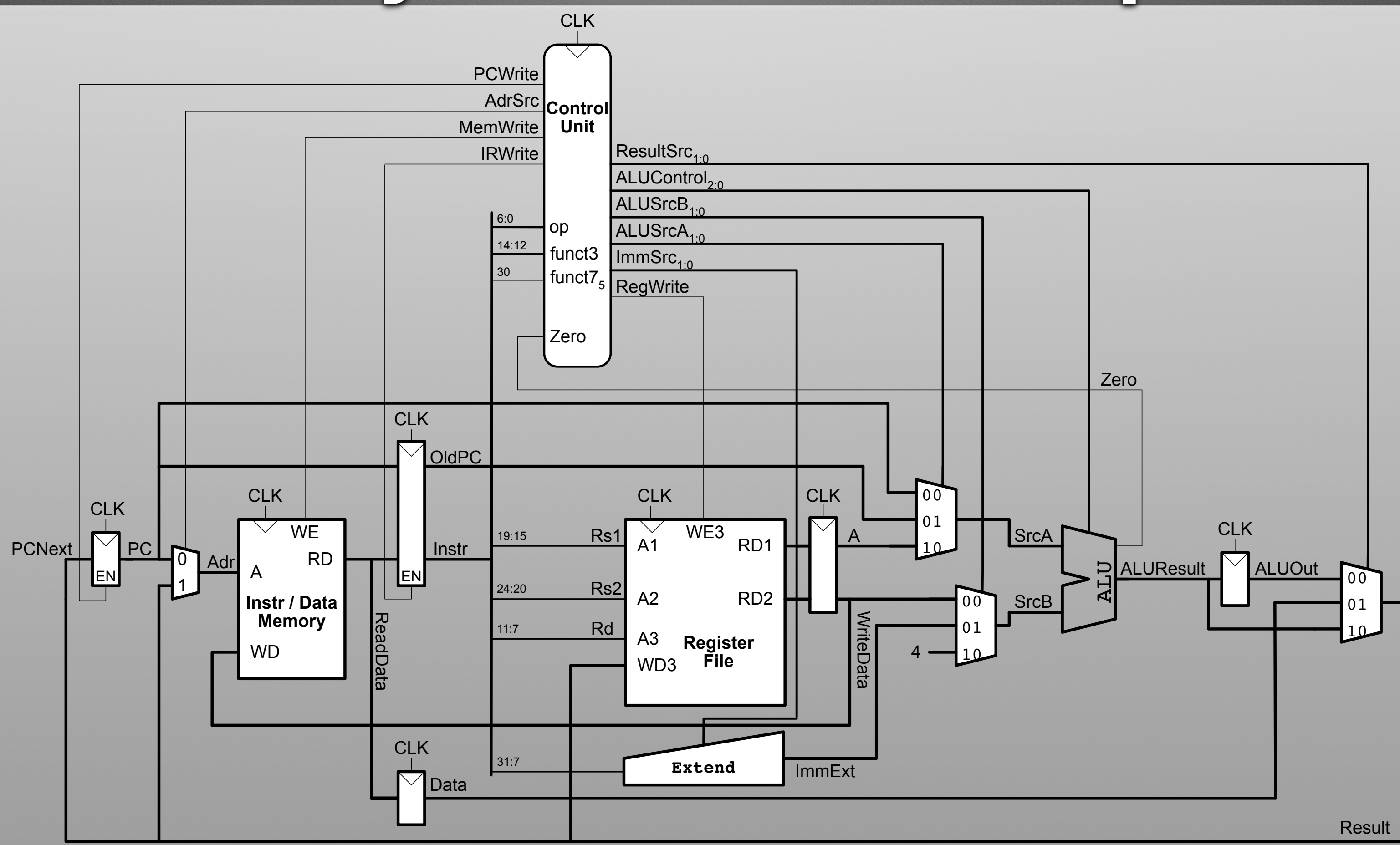
Simple (Single-Cycle) Control vs. Datapath



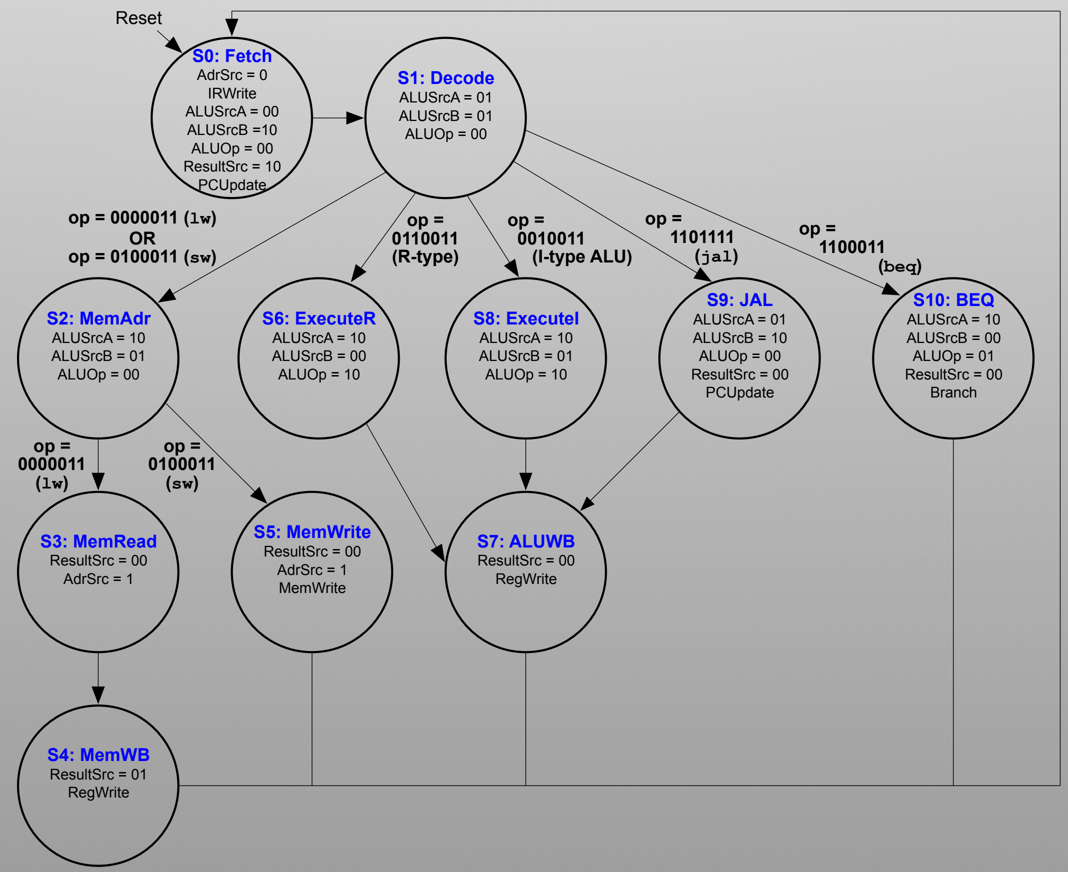
Exam Q: What are the control signals for instruction… (ex: addi t1,t0,32)



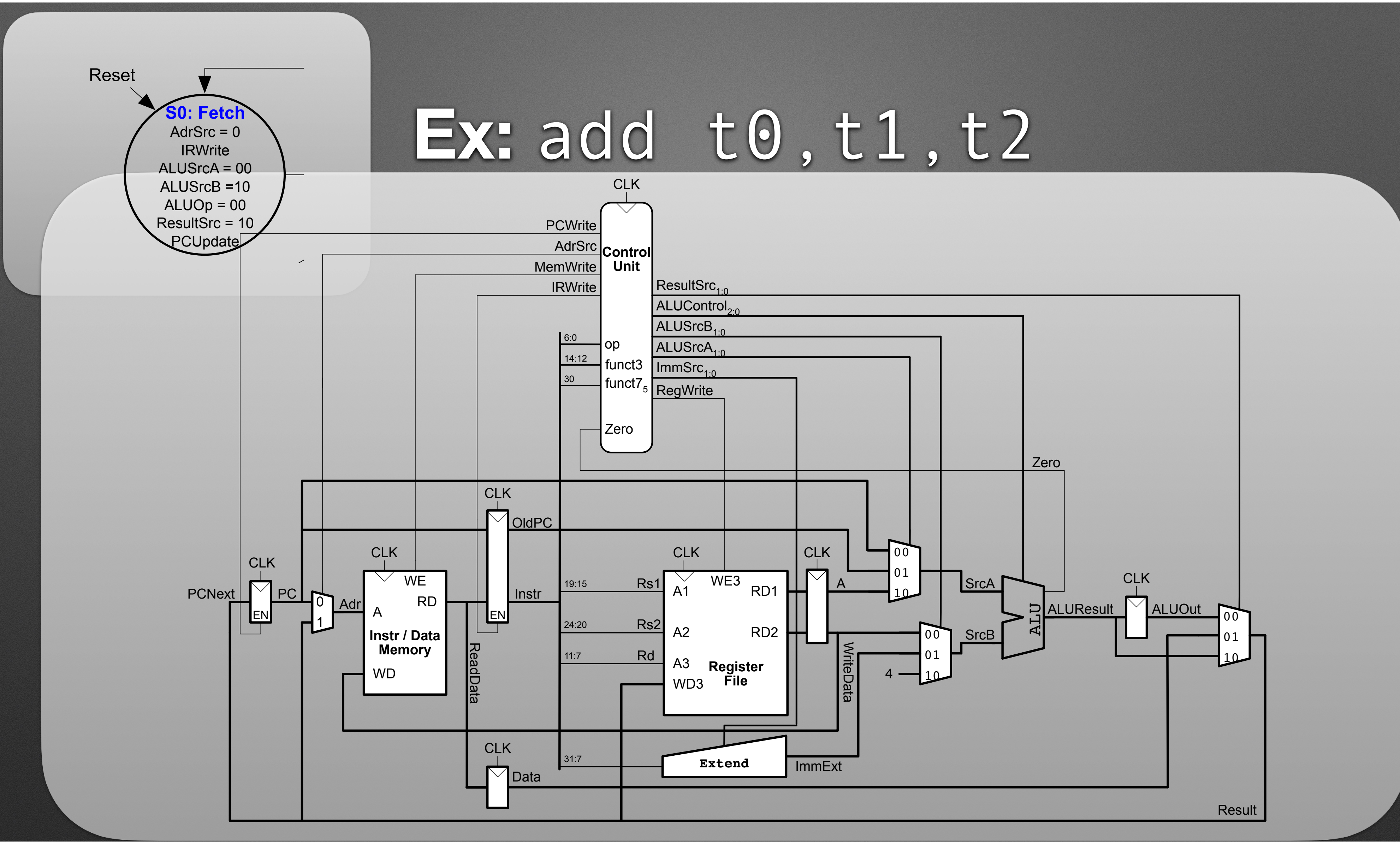
Multi-Cycle RISC-V Computer



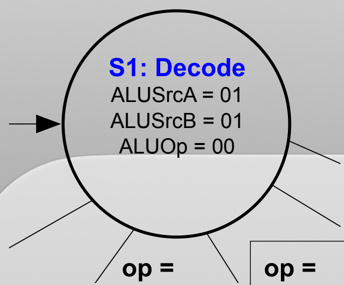
Process

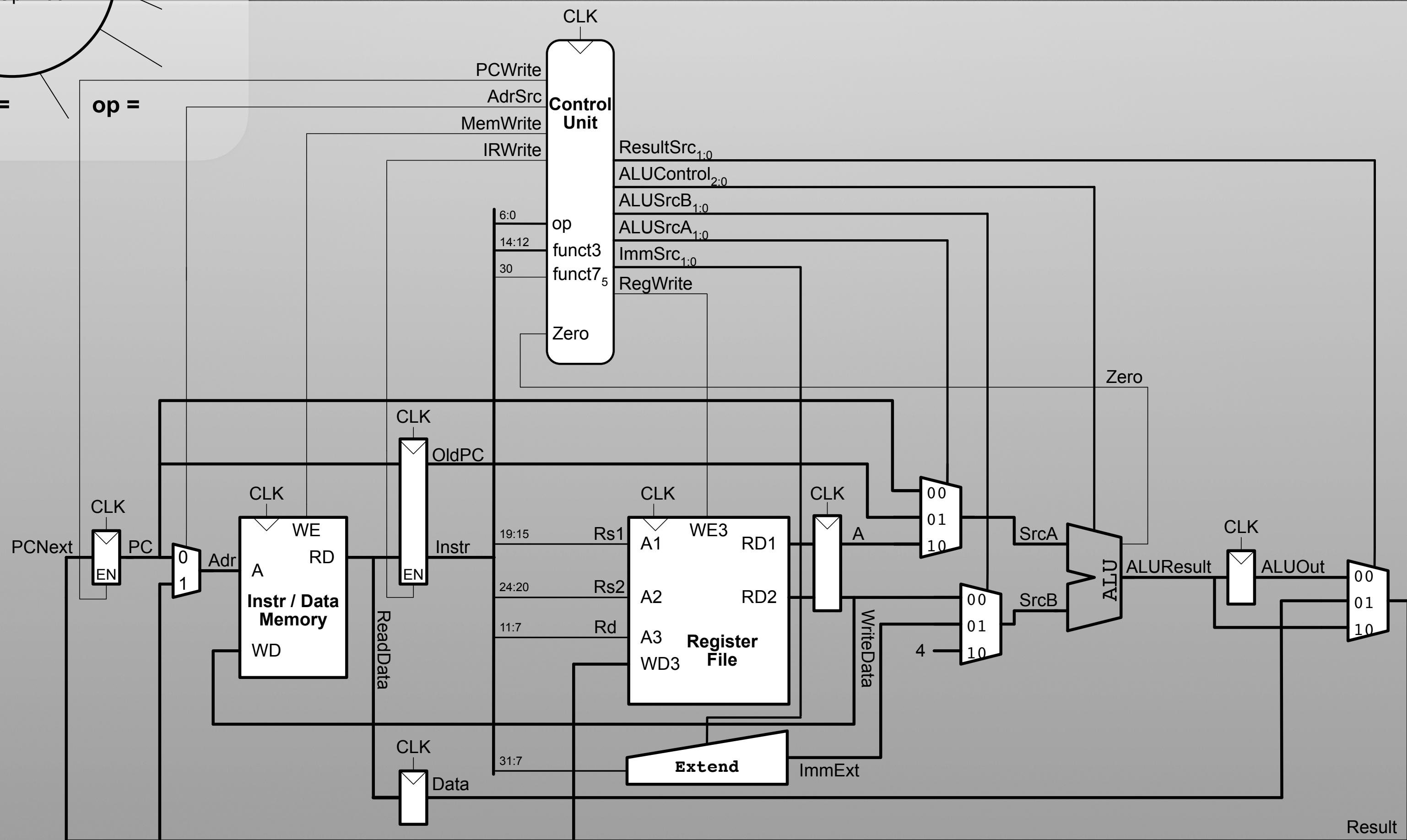

Reset
S0: Fetch
AdrSrc = 0
IRWrite
ALUSrcA = 00
ALUSrcB =10
ALUOp = 00
ResultSrc = 10
PCUpdate
S1: Decode
ALUSrcA = 01
ALUSrcB = 01
ALUOp = 00
op = 0000011 (lw)
OR
op = 0100011 (sw)
op = 0110011 (R-type)
op = 0010011 (I-type ALU)
op = 1101111 (jal)
op = 1100011 (beq)
S2: MemAdr
ALUSrcA = 10
ALUSrcB = 01
ALUOp = 00
S6: ExecuteR
ALUSrcA = 10
ALUSrcB = 00
ALUOp = 10
S8: ExecuteI
ALUSrcA = 10
ALUSrcB = 01
ALUOp = 10
S9: JAL
ALUSrcA = 01
ALUSrcB = 10
ALUOp = 00
ResultSrc = 00
PCUpdate
S10: BEQ
ALUSrcA = 10
ALUSrcB = 00
ALUOp = 01
ResultSrc = 00
Branch
op = 0000011 (lw)
op = 0100011 (sw)
S3: MemRead
ResultSrc = 00
AdrSrc = 1
S5: MemWrite
ResultSrc = 00
AdrSrc = 1
MemWrite
S7: ALUWB
ResultSrc = 00
RegWrite
S4: MemWB
ResultSrc = 01
RegWrite

Ex: add t0,t1,t2

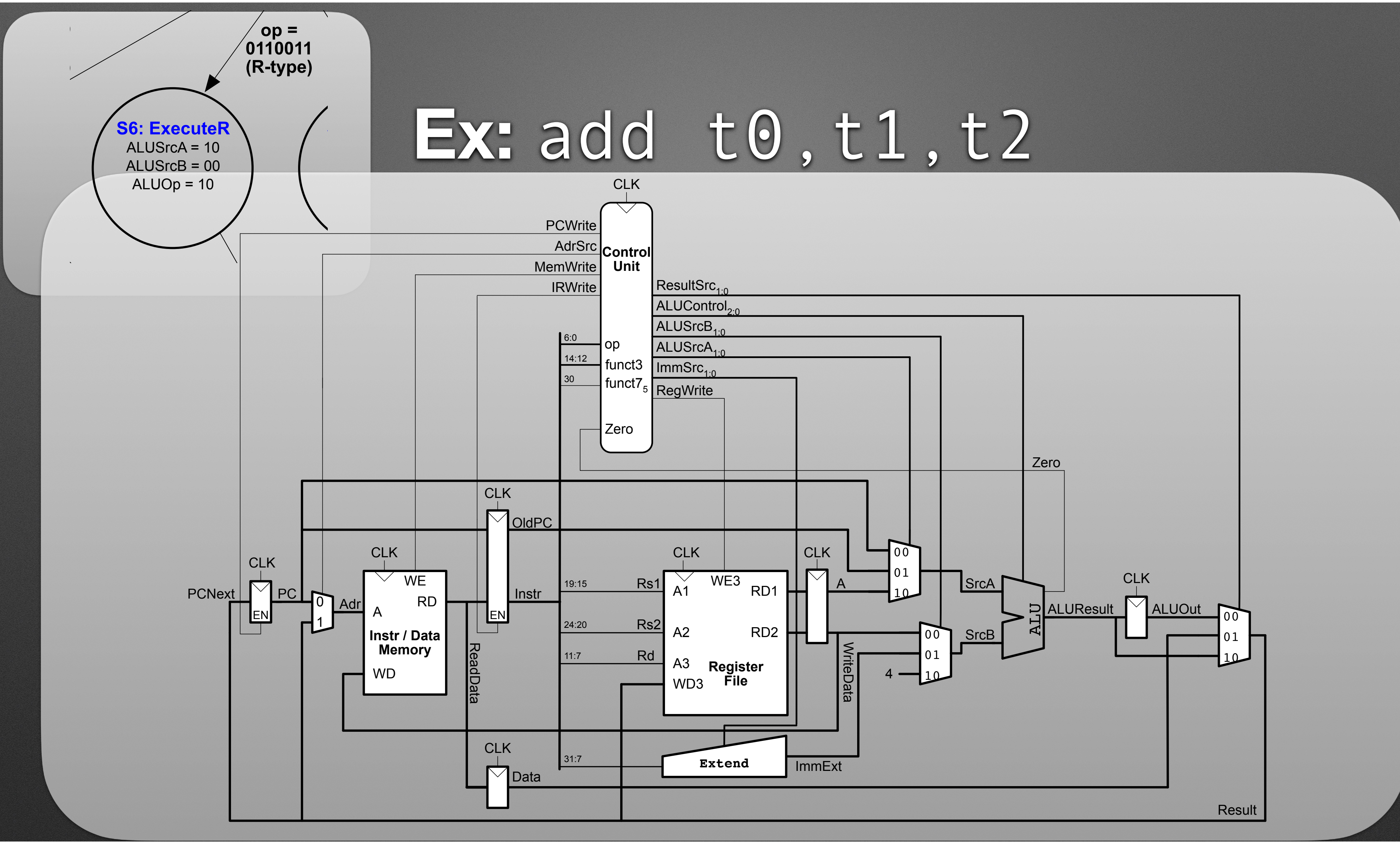


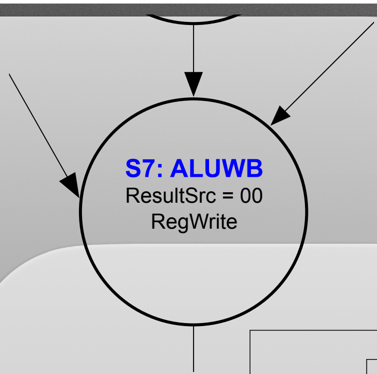
Ex: add t0,t1,t2



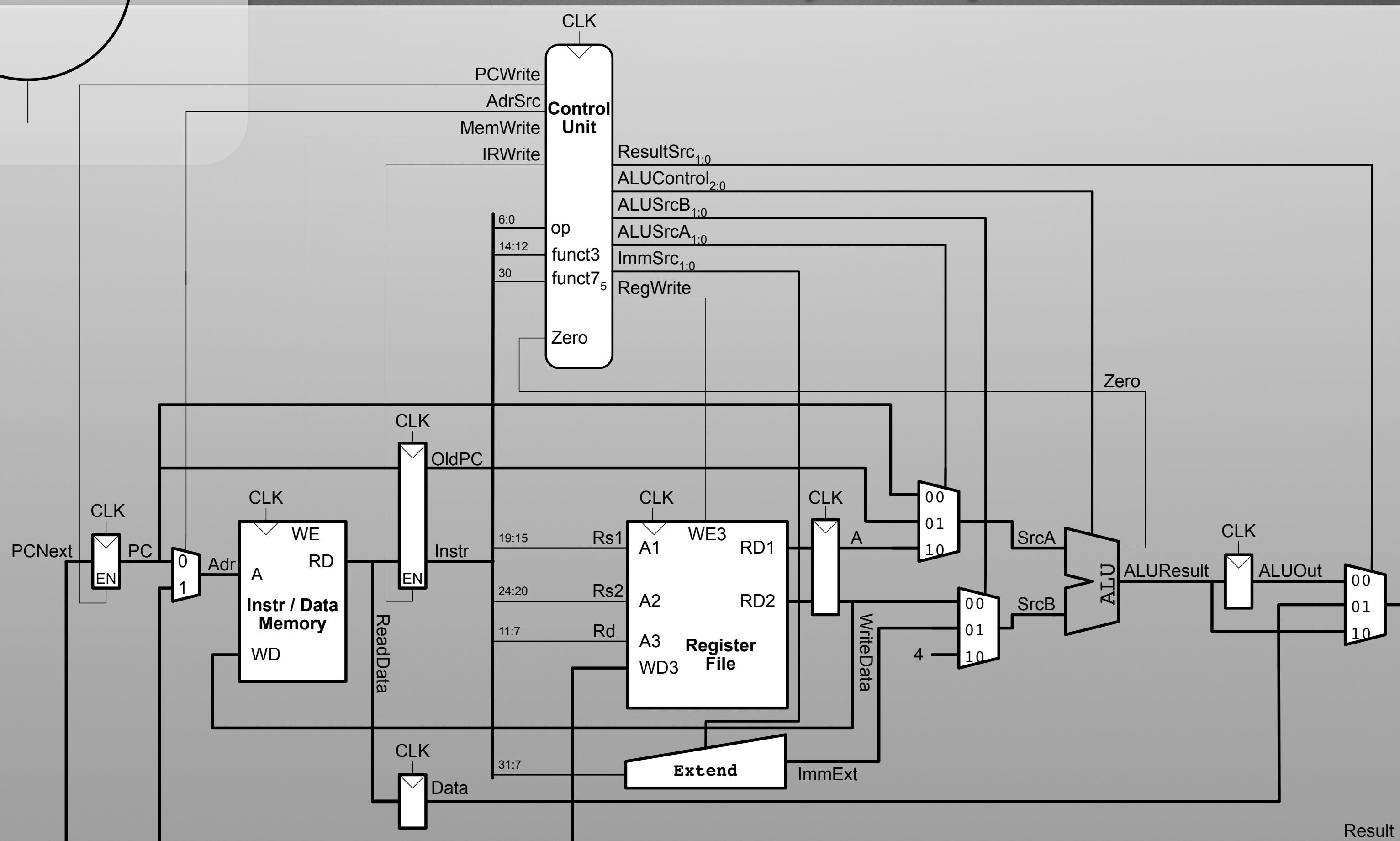


Ex: `add t0,t1,t2`

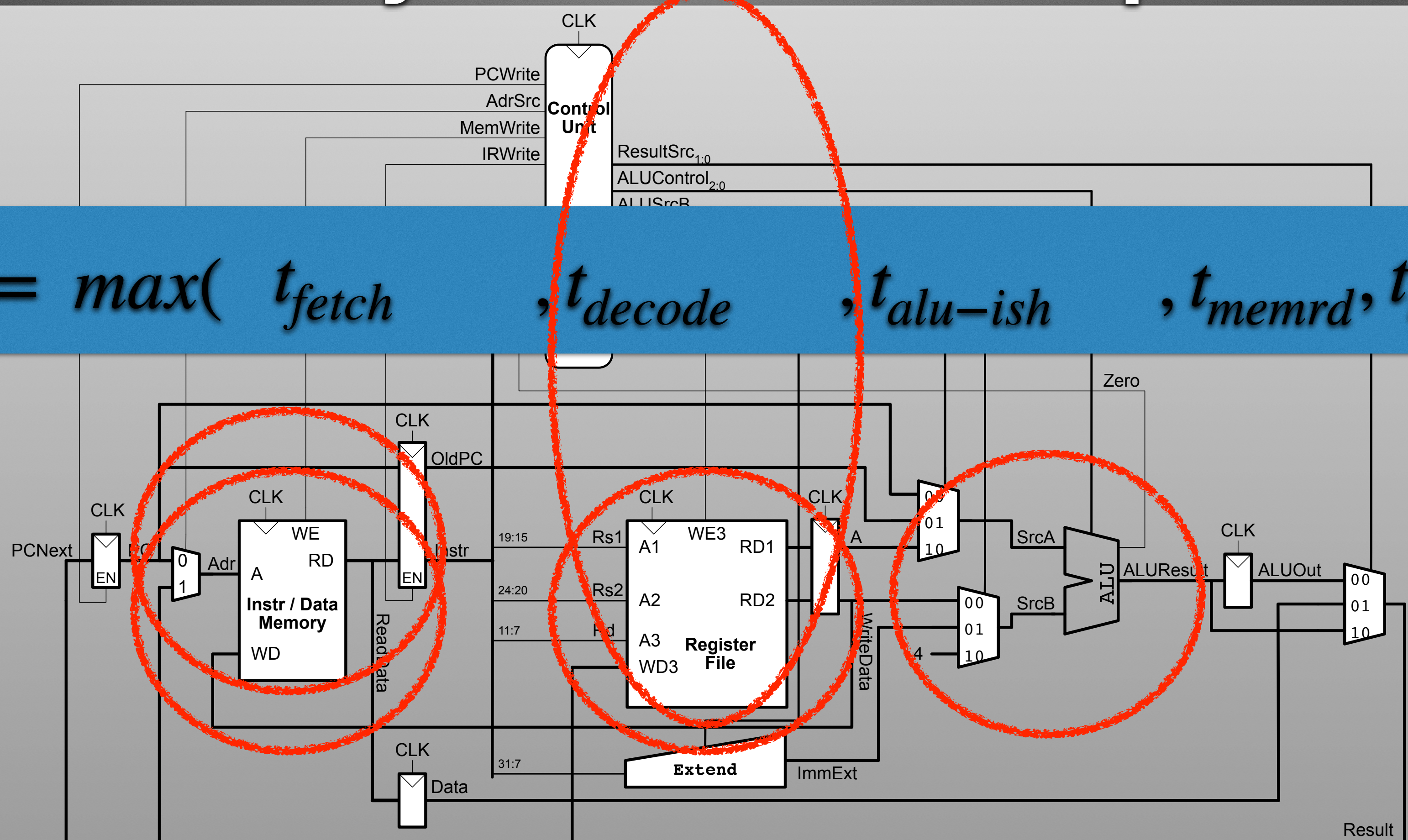




Ex: `add t0,t1,t2`



Multi-Cycle RISC-V Computer



$$t_{clock} = max(\quad t_{fetch} \quad , t_{decode} \quad , t_{alu-ish} \quad , t_{memrd}, t_{regwr})$$

Pros/Cons of Multi-Cycle

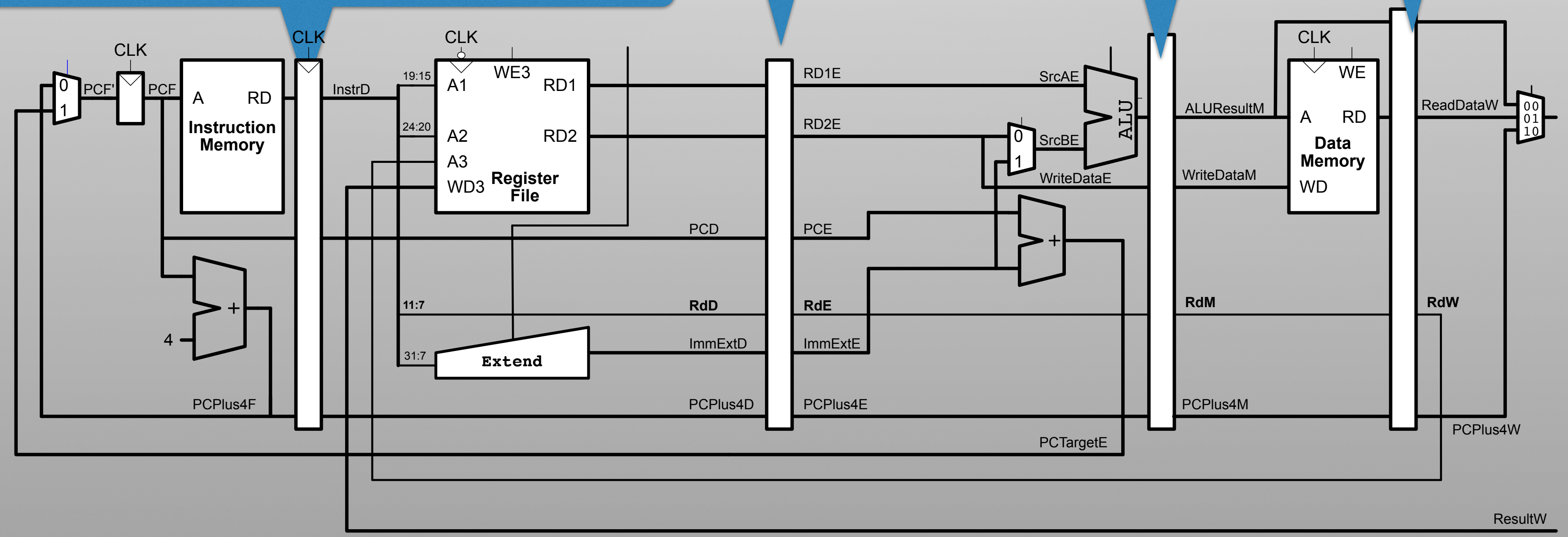
- Instructions take only required time: Not constrained by the slowest instruction!
- A little more complex

Registers:
Between stages;
Like the parts bin (hold parts for inst),
but parts move, not bins

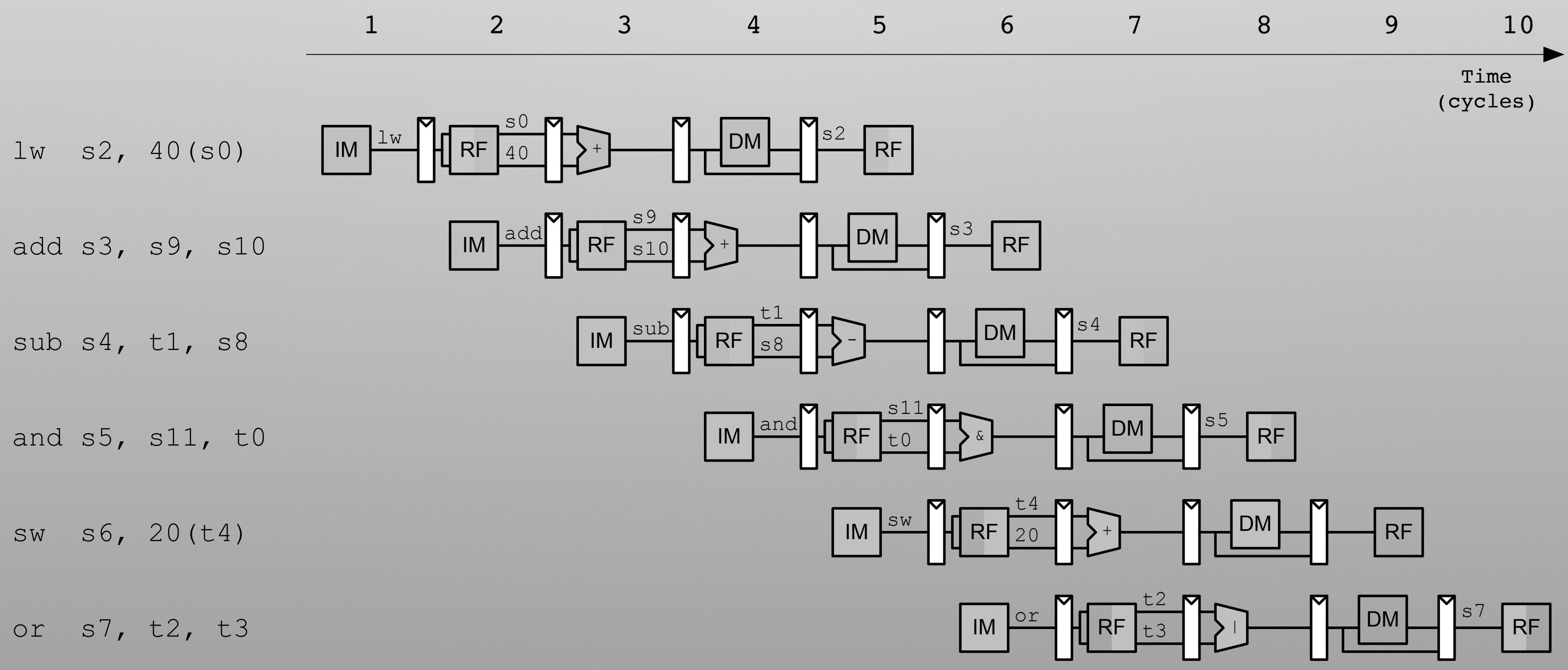
Each Stage

Each Stage

Each Stage



Pipeline CPU



Done.

Exam Q