Combinational Logic
Sequential Logic
CPU Datapath

CMPT 295 Week 10

Combinational Logic

Hardware Design

- This week's goals:

1. Understand how processors work
   - Requires digital systems knowledge

2. Understand how code is actually executed on a computer to analyze reliability, performance & security

- Layered abstractions
  - Transistors → Gates → Combinational Logic → Sequential Logic → Processors → Machine Language → Assembly → High-level Programming Languages → Application programs
  - At each step we can "abstract away" the lower layers

Synchronous Digital Systems (SDS)

Hardware of a processor (e.g., RISC-V) is an example of a Synchronous Digital System

Synchronous:

- All operations coordinated by a central clock
  - "Heartbeat" of the system (processor frequency)

Digital:

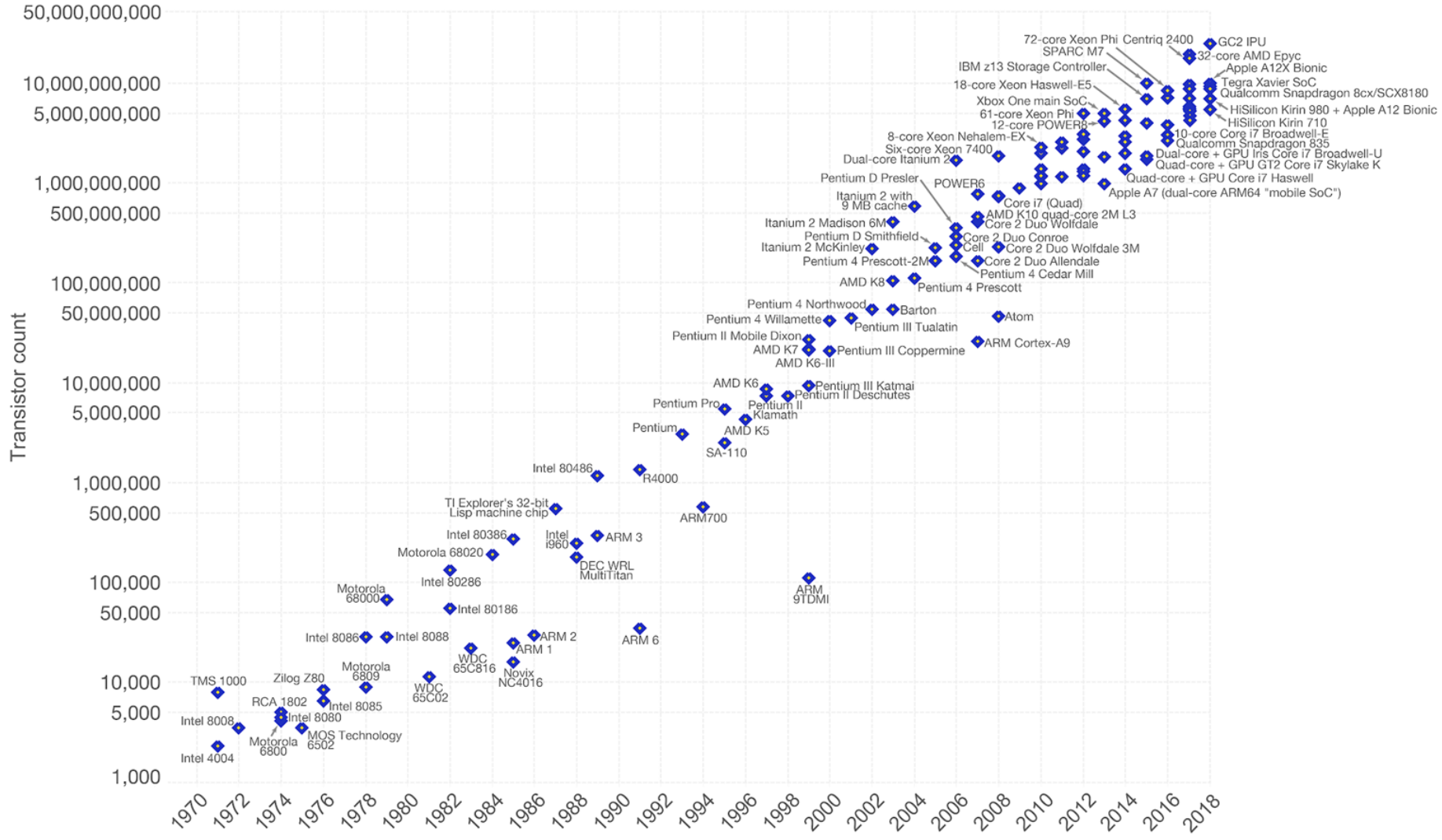
- Represent all values with two discrete values
- Electrical signals are treated as 1's and 0's
  - High/Low voltage represent True/False, 1/0

Moore's Law

- **Original Version (1965):** Since the integrated circuit was invented, the number of transistors in an integrated circuit has roughly doubled every year; this trend would continue for the foreseeable future
- 1975: Revised - circuit complexity doubles every two years

Moore's Law – The number of transistors on integrated circuit chips (1971-2018)

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are linked to Moore's law.



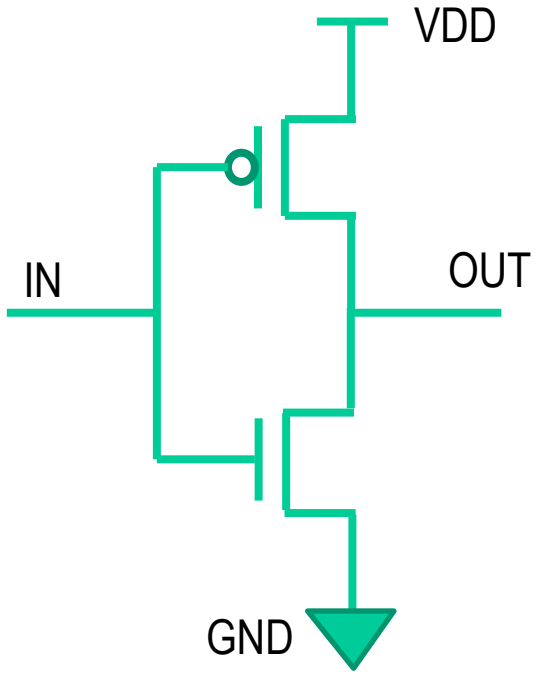
Data source: Wikipedia (https://en.wikipedia.org/wiki/Transistor_count)
The data visualization is available at OurWorldinData.org. There you find more visualizations and research on this topic.

Moore's Law

- **Original Version (1965):** Since the integrated circuit was invented, the number of transistors in an integrated circuit has roughly doubled every year; this trend would continue for the foreseeable future
- 1975: Revised - circuit complexity doubles every two years
- **Hardware Trend:** Hardware gets more powerful every year (due to technology advancement and the hard work of many engineers)
- **Software Trend:** Software gets faster and uses more resources (And has to keep up with ever-changing hardware)
- Digital circuits are used to build hardware

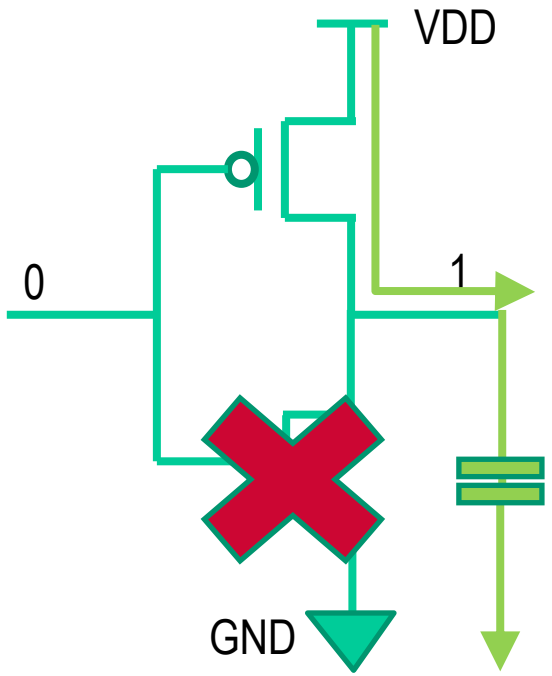
Transistors to Gates Example: Inverter

- CMOS technology
- Two transistors:
  - NMOS (top): turns on when input is 0 (low V)
  - PMOS (bottom): turns on when input is 1 (high V)



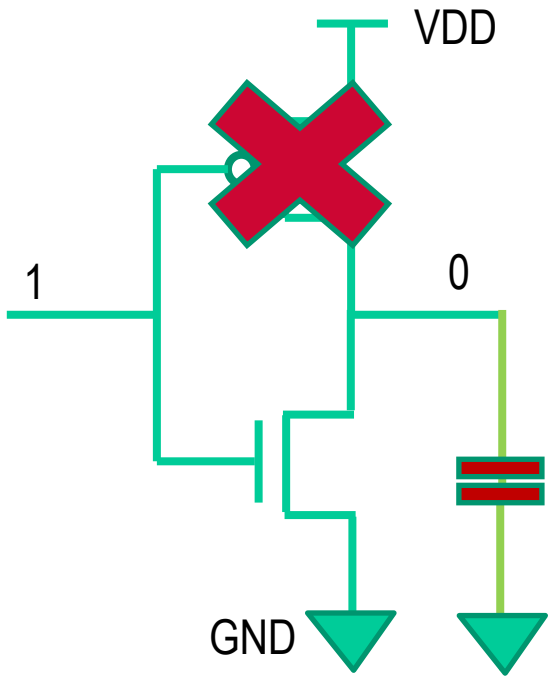
Transistors to Gates Example: Inverter

- Input = 0
- Top transistor turned on, bottom transistor turned off -> Output connected to VDD, capacitor charged
- Output = 1



Transistors to Gates Example: Inverter

- Input = 1
- Top transistor turned off, bottom transistor turned on -> Output connected to GND, capacitor discharged
- Output = 0



Inverter is commonly called “NOT gate”

Combinational vs. Sequential Logic

- *Digital Systems* consist of two basic types of circuits:
  - Combinational Logic (CL)
    - Output is a function of the inputs only, not the history of its execution
    - Example: add A, B (ALUs)
  - Sequential Logic (SL)
    - Circuits that "remember" or store information
    - Also called "State Elements"
    - Example: Memory and registers

Simple Logic Gates

- Special names and symbols:

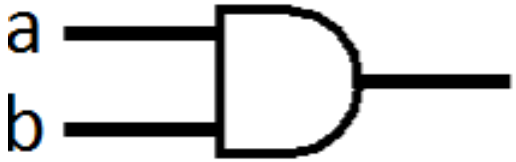
Truth Table

NOT

a	NOT a
0	1
1	0

True if input is false

AND



a	b	a AND b
0	0	0
0	1	0
1	0	0
1	1	1

True if both inputs are true

OR

a	b	A OR b
0	0	0
0	1	1
1	0	1
1	1	1

True if at least one input is true

12

More Simple Logic Gates

Inverted versions are easier to implement in CMOS

NAND

a	b	a NAND b
0	0	1
0	1	1
1	0	1
1	1	0

True if at least one input is false

NOR

a	b	a NOR b
0	0	1
0	1	0
1	0	0
1	1	0

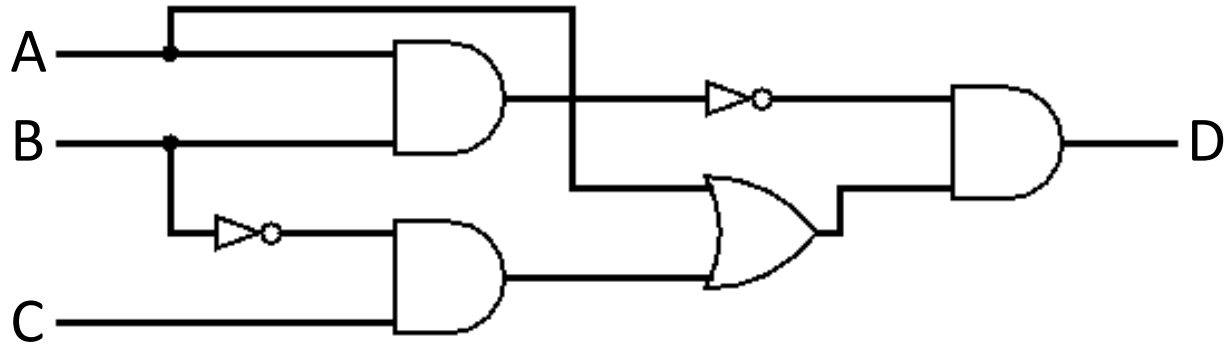
True if both inputs are false

XOR

a	b	a XOR b
0	0	0
0	1	1
1	0	1
1	1	0

True if exactly one input is true
(or if odd number of inputs are true for > 2 inputs)

Combining Multiple Logic Gates



D = (NOT(**A** AND **B**)) AND (**A** OR (NOT **B** AND **C**))

How to Represent Combinational Logic?

✓ Text Description

✓ Circuit Diagram

- Transistors and wires
- Logic Gates

✓ Truth Table

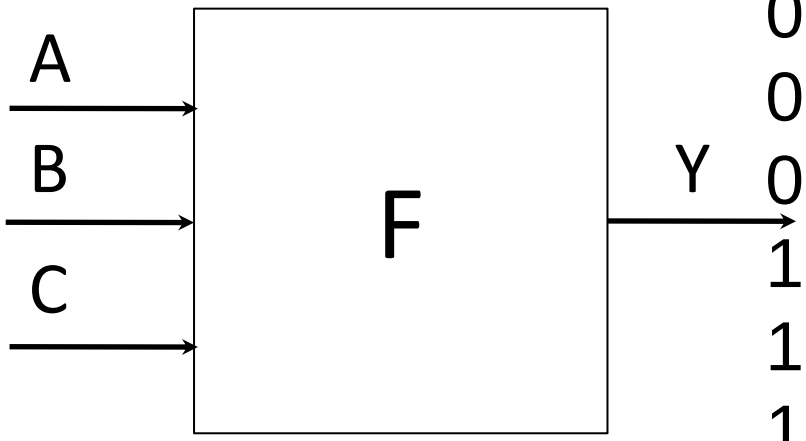
✓ Boolean Expression

✓ *All are equivalent*

Truth Tables

- Table that relates the inputs to a combinational logic circuit to its output
  - Output *only* depends on current inputs
  - Use abstraction of 0/1 (F/T) instead of high/low Voltage
  - Shows output for *every* possible combination of inputs
- How big is a truth table with N inputs?
  - 0 or 1 for each of N inputs, so 2^N rows

N-input Truth Tables

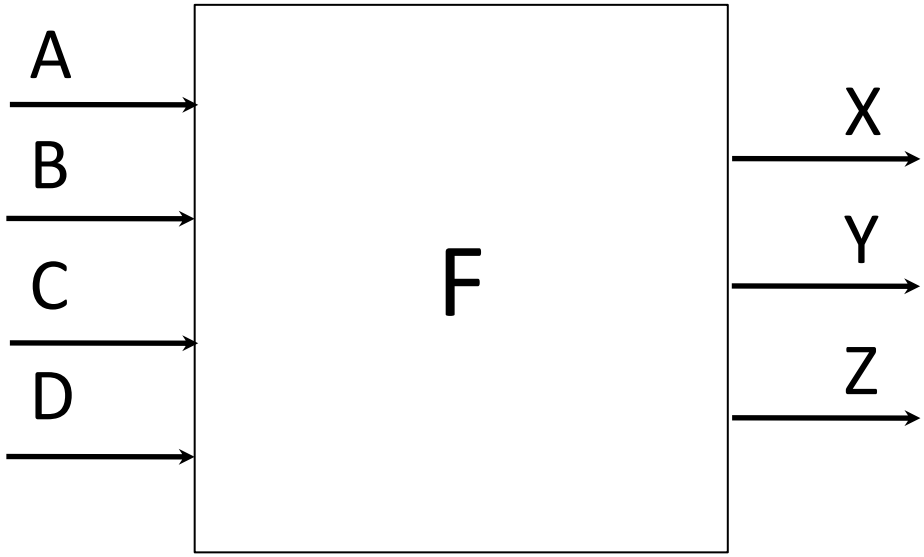


A	B	C		Y
0	0	0	F(0,0,0)	0
0	0	1	F(0,0,1)	1
0	1	0	F(0,1,0)	0
0	1	1	F(0,1,1)	0
1	0	0	F(1,0,0)	0
1	0	1	F(1,0,1)	1
1	1	0	F(1,1,0)	1
1	1	1	F(1,1,1)	0

For N inputs, how many distinct functions F do we have?

Function maps each row to 0 or 1, so 2^{2^N} possible functions

Truth Tables with Multiple Outputs



- For 3 outputs, just three indep. functions: X(A,B,C,D), Y(A,B,C,D), and Z(A,B,C,D)
  - Can show functions in separate columns (no additional rows)

Question: Which of the columns A-D is the correct output of the Truth Table for: (X XOR Y) OR (NOT Z)

X	Y	Z	(A)	(B)	(C)	(D)
0	0	0	1	1	1	1
0	0	1	0	0	0	0
0	1	0	1	1	1	1
0	1	1	1	1	1	1
1	0	0	0	1	1	1
1	0	1	1	1	0	1
1	1	0	1	1	1	0
1	1	1	1	0	1	1

Question: Which of the columns A-D is the correct output of the Truth Table for: (X XOR Y) OR (NOT Z)

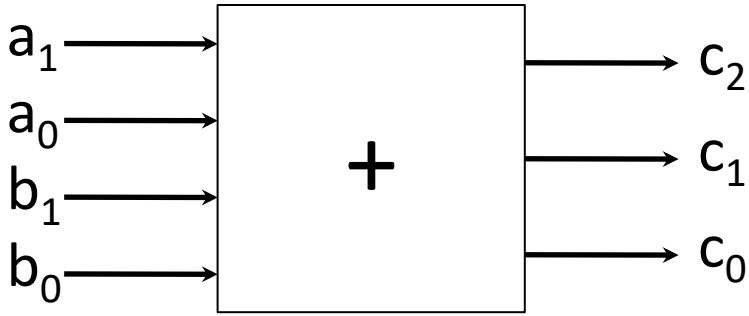
X	Y	Z	(A)	(B)	(C)	(D)
0	0	0	1	1	1	1
0	0	1	0	0	0	0
0	1	0	1	1	1	1
0	1	1	1	1	1	1
1	0	0	0	1	1	1
1	0	1	1	1	0	1
1	1	0	1	1	1	0
1	1	1	1	0	1	1

More Complex Truth Tables

3-Input Majority

a	b	c	y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

2-bit Adder



How many rows?

A		B		C		
a_1	a_0	b_1	b_0	c_2	c_1	c_0
			.			
			.			
			.			

Truth Tables Don't Scale

- Truth tables are huge
  - Write out EVERY combination of inputs and outputs (thorough, but inefficient)
  - Finding a particular combination of inputs involves scanning a large portion of the table
- Boolean Algebra is a shorter way to represent combinational logic

Boolean Algebra

- Represent inputs and outputs as variables
  - Each variable can only take on the value 0 or 1
- Overbar, ¬, ’ mean NOT: “logical complement”
  - e.g., if A = 0, $\overline{A}$ = 1. If A = 1, then $\overline{A}$ = ¬A = A’ = 0
- Plus (+) is 2-input OR: “logical sum”
- Product (·) is 2-input AND: “logical product”
  - Sometimes omitted
- All other gates and logical expressions can be built from combinations of these

$\overline{\mathbf{AB}} + \mathbf{A}\overline{\mathbf{B}}$ == (NOT(**A** AND **B**)) OR (**A** AND (NOT **B**))

Laws of Boolean Algebra

These laws allow us to simplify Boolean expressions:

$x \cdot \overline{x} = 0$	$x + \overline{x} = 1$	complementarity
$x \cdot 0 = 0$	$x + 1 = 1$	laws of 0's and 1's
$x \cdot 1 = x$	$x + 0 = x$	identities
$x \cdot x = x$	$x + x = x$	idempotent law
$x \cdot y = y \cdot x$	$x + y = y + x$	commutativity
$(xy)z = x(yz)$	$(x + y) + z = x + (y + z)$	associativity
$x(y + z) = xy + xz$	$x + yz = (x + y)(x + z)$	distribution
$xy + x = x$	$(x + y)x = x$	uniting theorem
$\overline{x}y + x = x + y$	$(\overline{x} + y)x = xy$	uniting theorem v.2
$\overline{x \cdot y} = \overline{x} + \overline{y}$	$\overline{x + y} = \overline{x} \cdot \overline{y}$	DeMorgan's Law

Converting Truth Table to Boolean Expression

- Read off of table
  - For 1, write variable name
  - For 0, write complement of variable
- *Sum of Products (SoP)*
  - Take rows with 1's in output column, sum products of inputs
  - c= $\overline{a}\,b + a\,\overline{b}$
- *Product of Sums (PoS)*
  - Take rows with 0's in output column, product the sum of the *complements of the inputs*
  - c = $(a + b) \cdot (\overline{a} + \overline{b})$

a	b	c
0	0	0
0	1	1
1	0	1
1	1	0

We can show that these are equivalent!

Simplifying Boolean Expressions

- **Logic Delay:** Everything we are dealing with is just an abstraction of transistors and wires
  - Inputs propagating to the outputs are voltage signals passing through transistor networks
  - There is always some *delay* before a CL's output updates to reflect the inputs
  - Critical Path is longest delay from any input to output. Could be represented as "n gate delays"
- Simpler Boolean expressions ↔ smaller transistor networks ↔ smaller circuit delays ↔ faster hardware

Simplifying Boolean Expressions: Example

$$y = ab + a + c$$

Karnaugh Maps

- Used to simplify Boolean expressions of 2-4 variables
- Table composed of squares each representing a unique combination of all variable (1 if true, else blank)
- Two variable Map:

x \ y	0	1
0	$\bar{x}\bar{y}$	$\bar{x}y$
1	$x\bar{y}$	xy

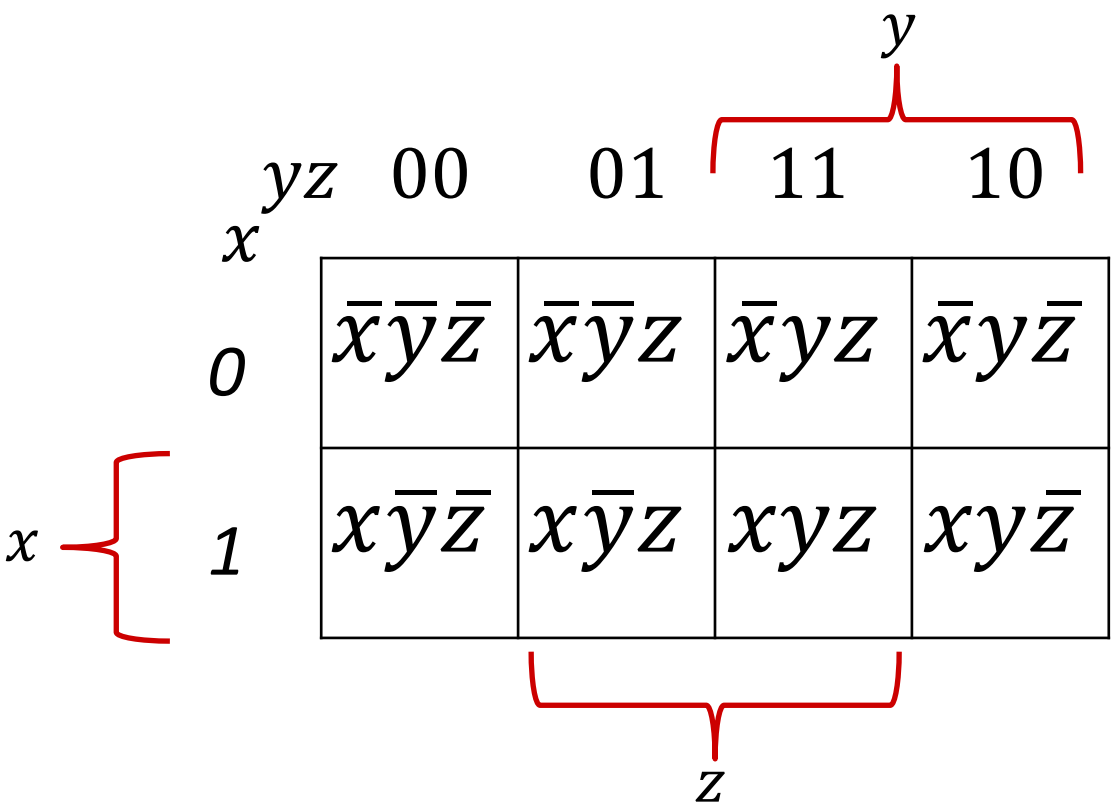
Example: Boolean Expression?

x \ y	0	1
0		1
1	1	1

x	y	?
0	0	0
0	1	1
1	0	1
1	1	1

$$x + y$$

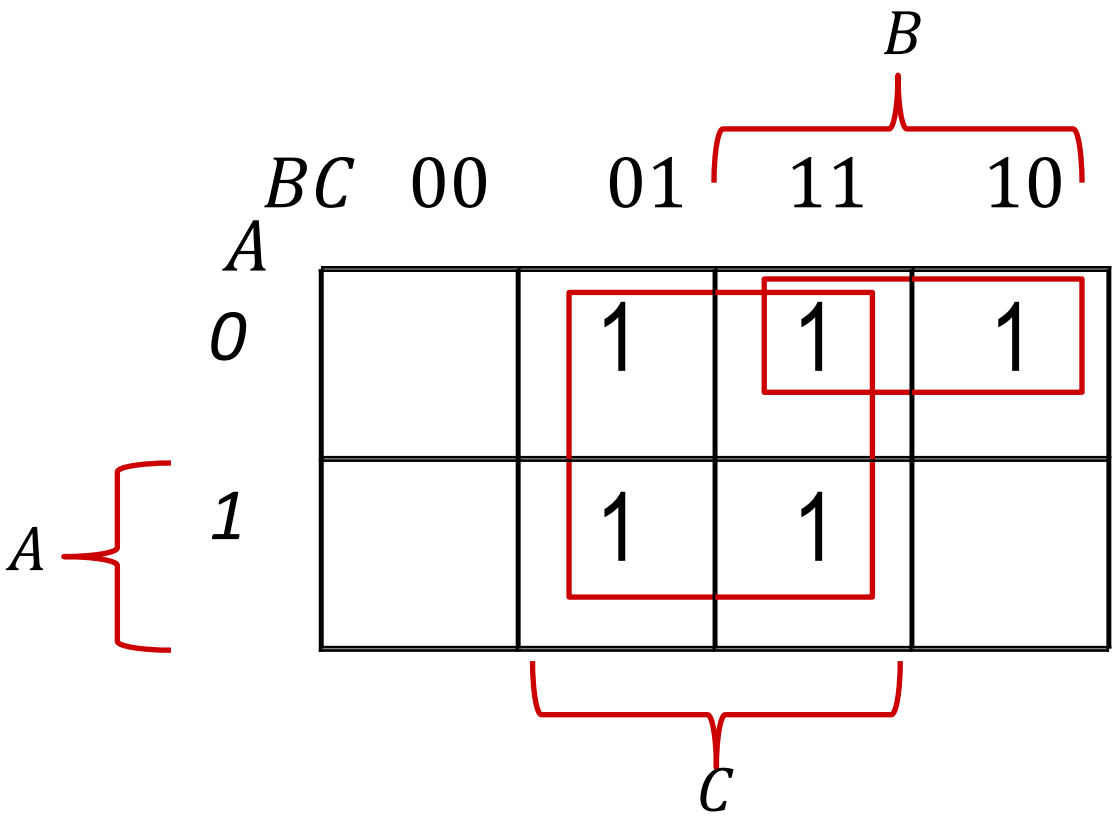
Three Variable Karnaugh Maps



Question: Simplify $\bar{A}C + \bar{A}B + A\bar{B}C + BC$

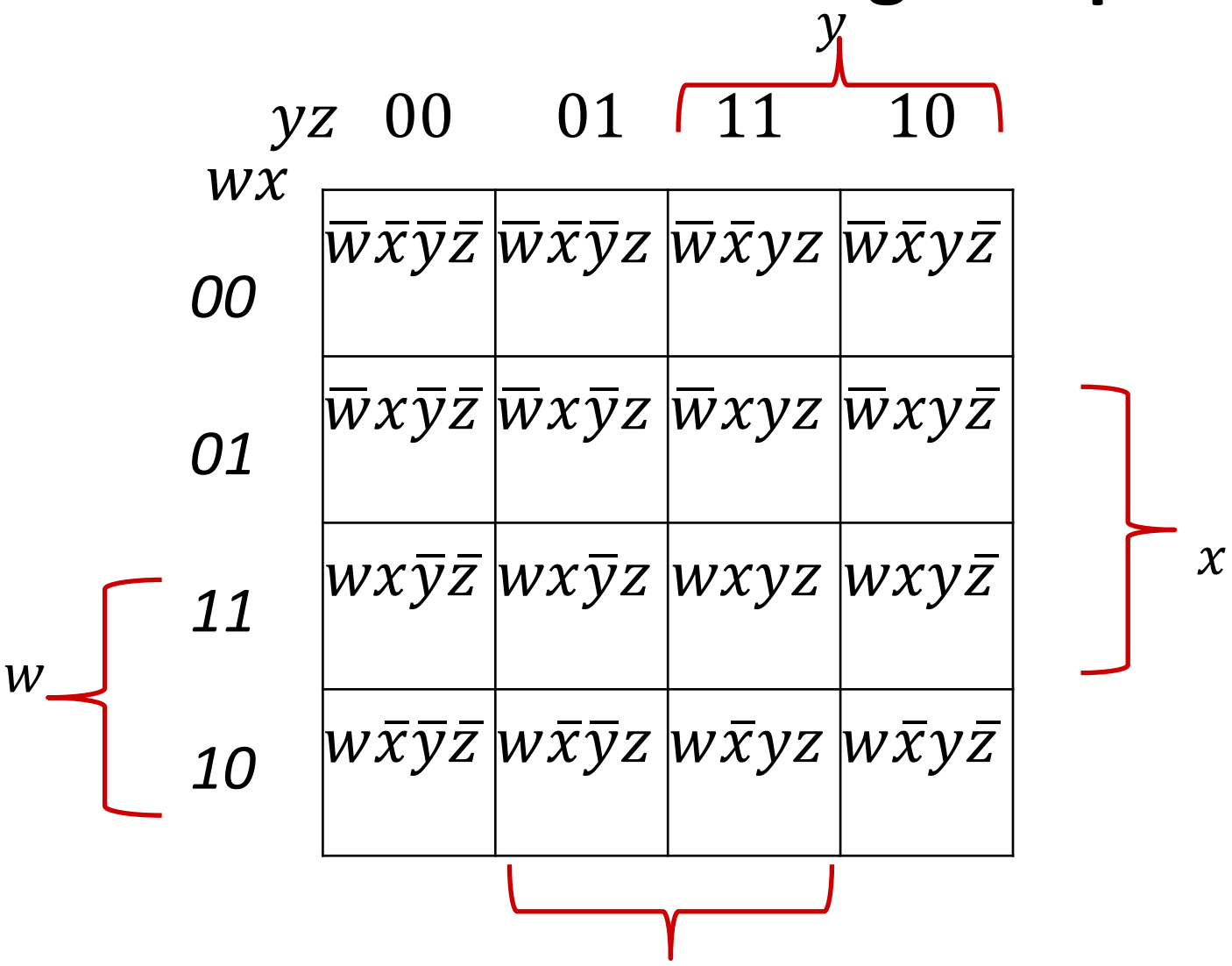
Example: Simplify 3-Variable Expression

Question: Simplify $\bar{A}C + \bar{A}B + A\bar{B}C + BC$



Answer: $C + \bar{A}B$

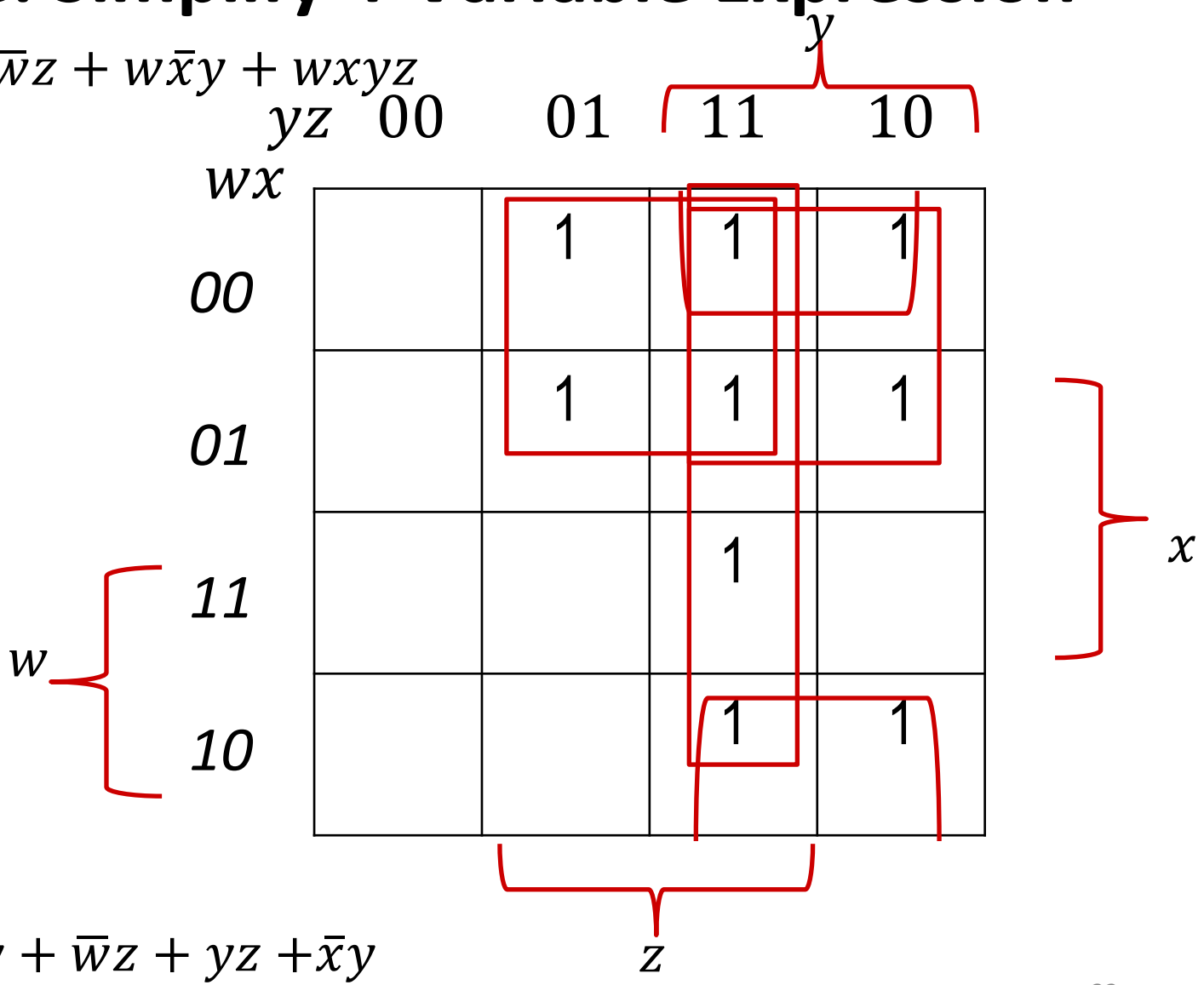
Four Variable Karnaugh Maps



Question: Simplify $\bar{w}y + \bar{w}z + w\bar{x}y + wxyz$

Example: Simplify 4-Variable Expression

Simplify $\bar{w}y + \bar{w}z + w\bar{x}y + wxyz$

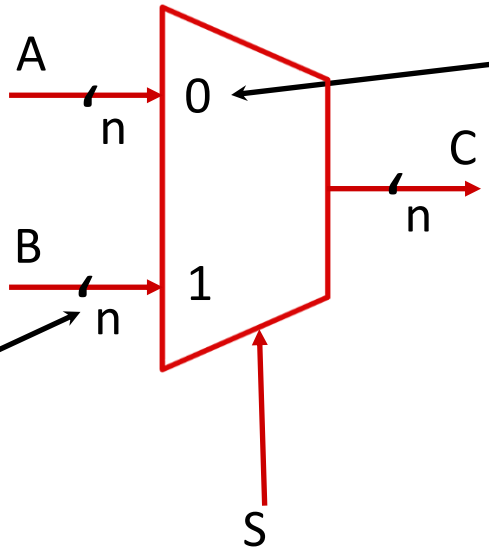


Solution: $\bar{w}y + \bar{w}z + yz + \bar{x}y$

Useful Combinational Circuits

Data Multiplexor (MUX)

- Multiplexor ("MUX") is a *selector*
  - Place one of multiple inputs onto output (N-to-1)
- Shown below is an n-bit 2-to-1 MUX
  - Input S selects between two inputs of n bits each

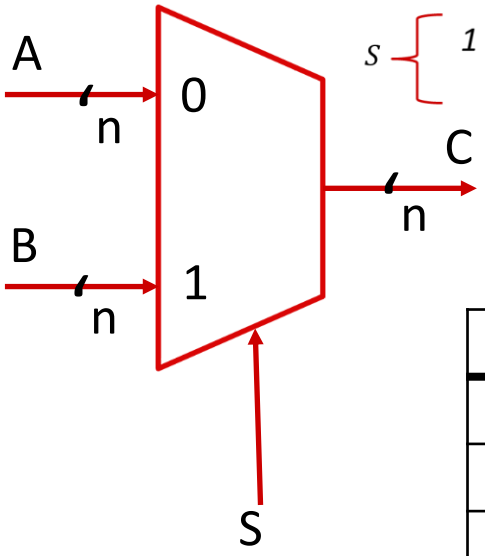


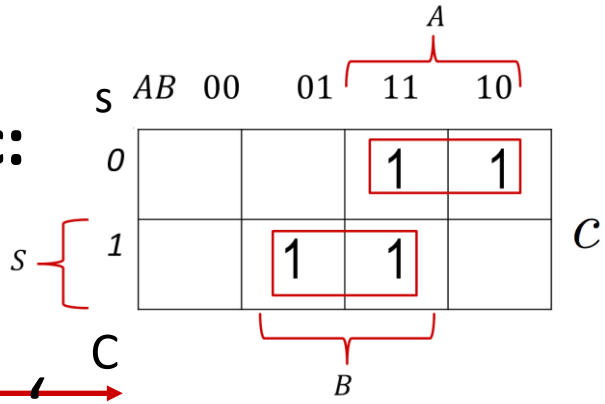
This input is passed to output if selector bits match shown value

Represents that this input has n bits

Implementing a 1-bit 2-to-1 MUX

- **Schematic:**





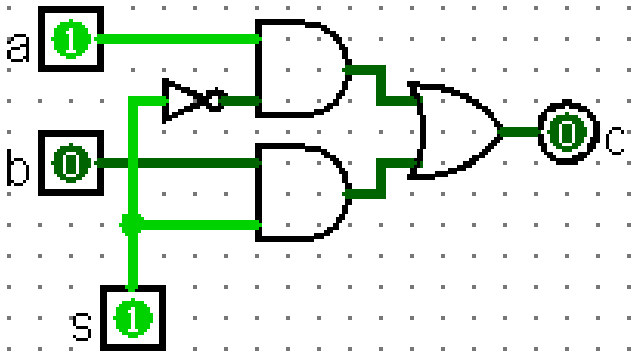
- **Boolean Algebra:**

$$
\begin{aligned}
&= \bar{s}a\bar{b} + \bar{s}ab + s\bar{a}b + sab \\
&= \bar{s}(a\bar{b} + ab) + s(\bar{a}b + ab) \\
&= \bar{s}(a(\bar{b} + b)) + s((\bar{a} + a)b) \\
&= \bar{s}(a(1) + s((1)b) \\
&= \bar{s}a + sb
\end{aligned}
$$

- **Truth Table:**

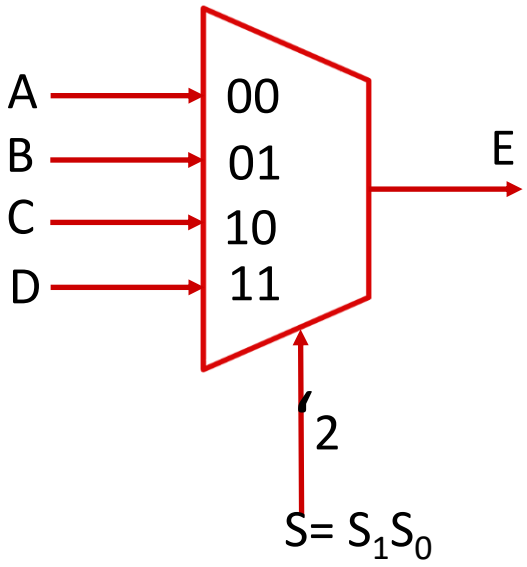
s	a	b	c
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

- **Circuit Diagram:**



1-bit 4-to-1 MUX

- **Schematic:**

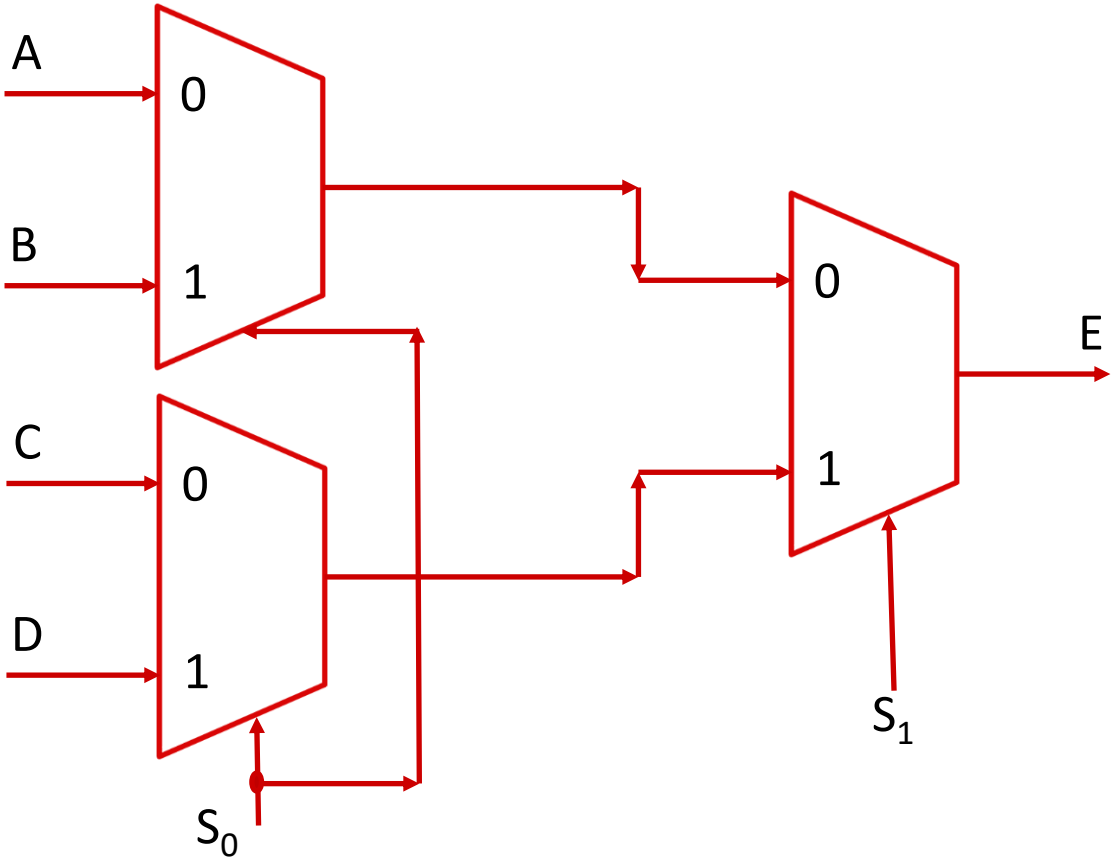


- **Truth Table:** How many rows? 2^6
- **Boolean Expression:**

$$E = \overline{S_1 S_0} A + \overline{S_1} S_0 B + S_1 \overline{S_0} C + S_1 S_0 D$$

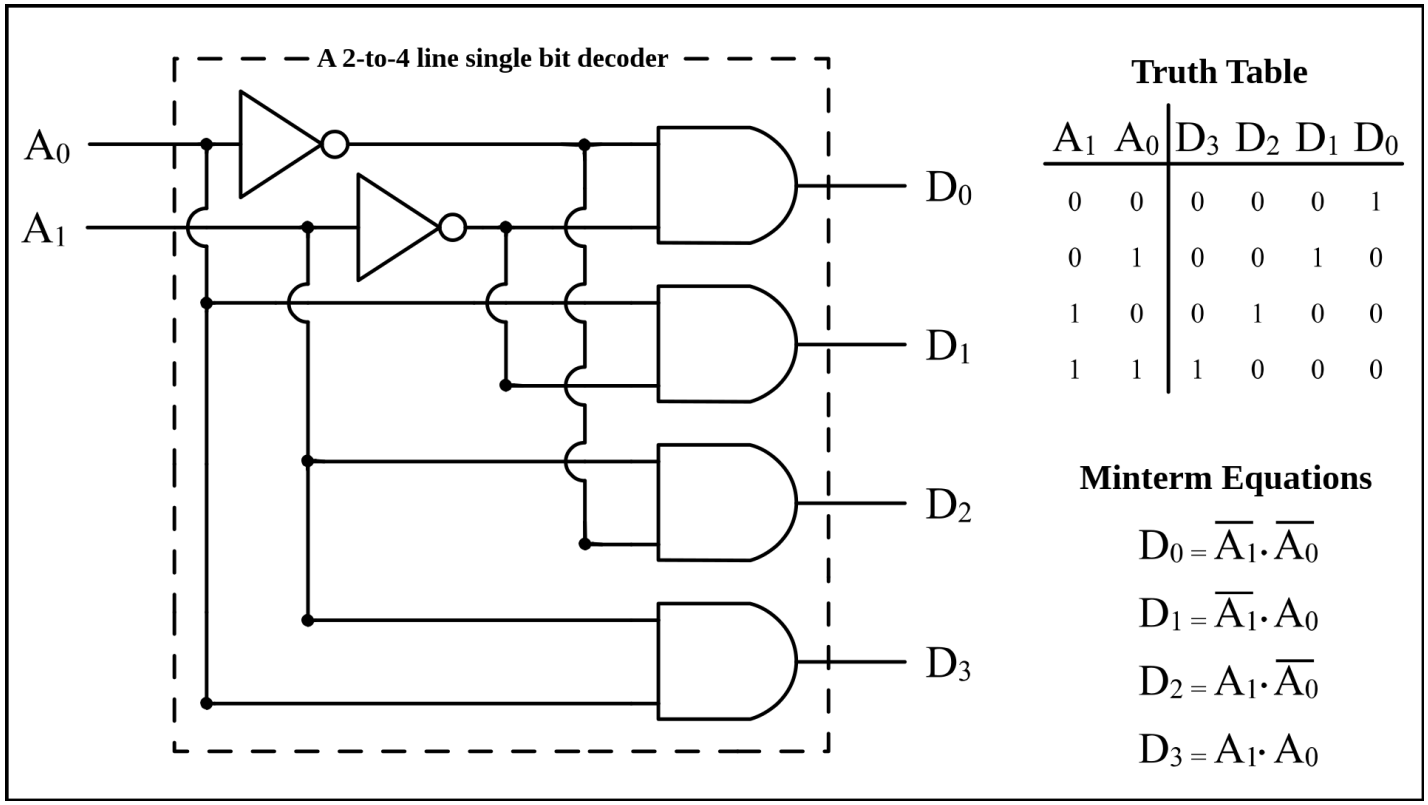
Another Design for 4-to-1 MUX

- Can we leverage what we've previously built?
  - Alternative hierarchical approach:



Decoder

- Enable one of 2^N outputs based on N input
- Example: 2-to-4 decoder

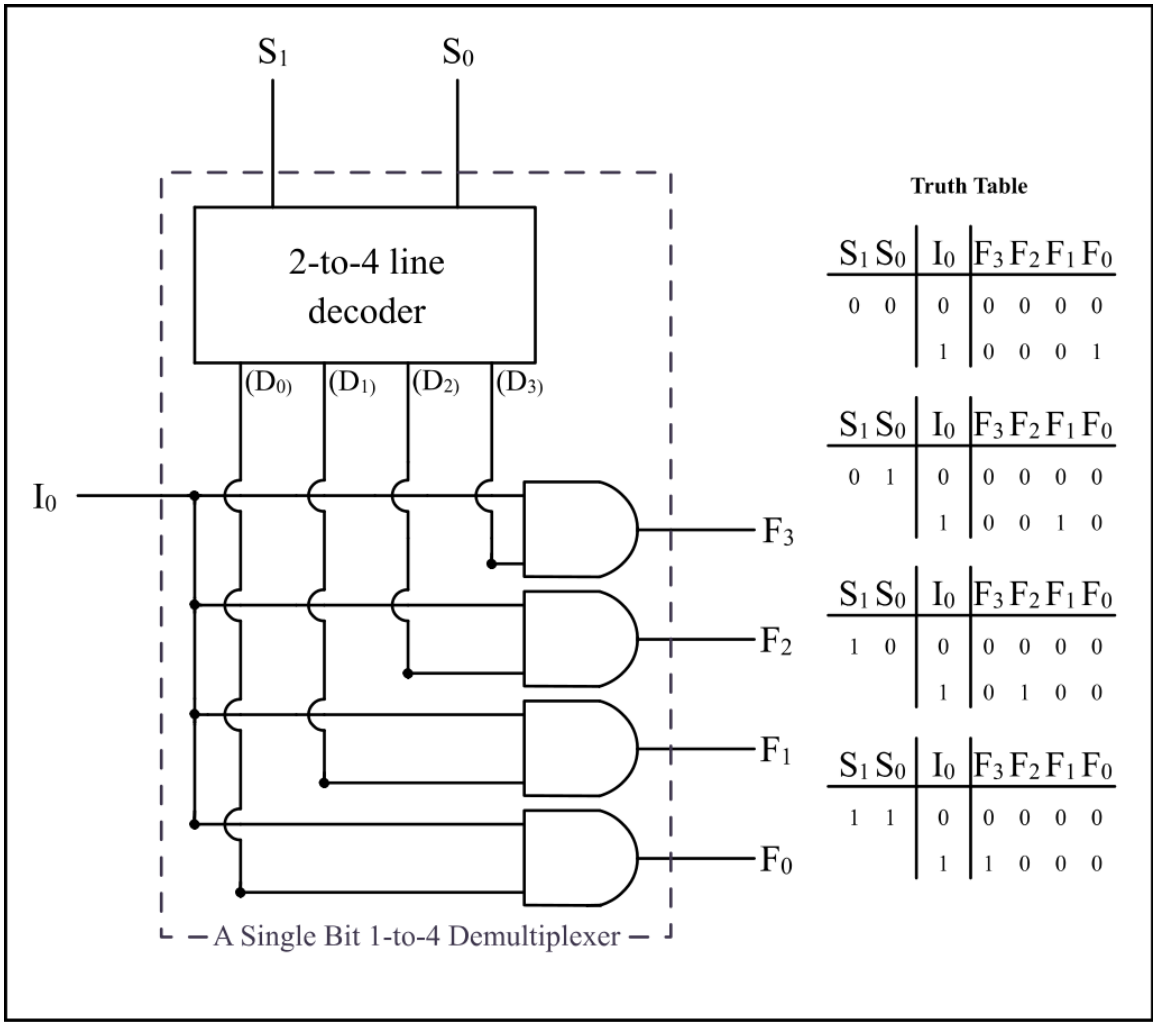


By BlueJester0101, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=3668293

- Use case: Choose ALU operation based on instruction op-code

Demultiplexer (Demux)

- Similar to decoder with an enable signal



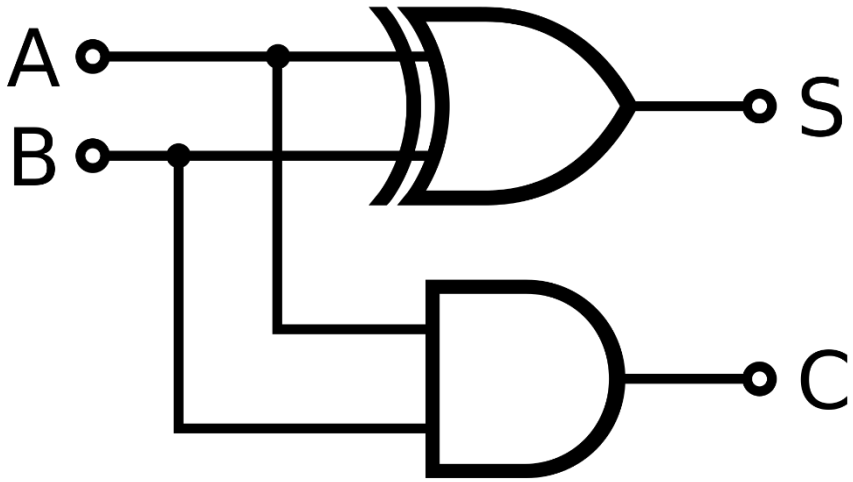
Truth Table

S_1	S_0	I_0	F_3	F_2	F_1	F_0
0	0	0	0	0	0	0
		1	0	0	0	1
0	1	0	0	0	0	0
		1	0	0	1	0
1	0	0	0	0	0	0
		1	0	1	0	0
1	1	0	0	0	0	0
		1	1	0	0	0

Single-Bit Binary Adder (Half Adder)

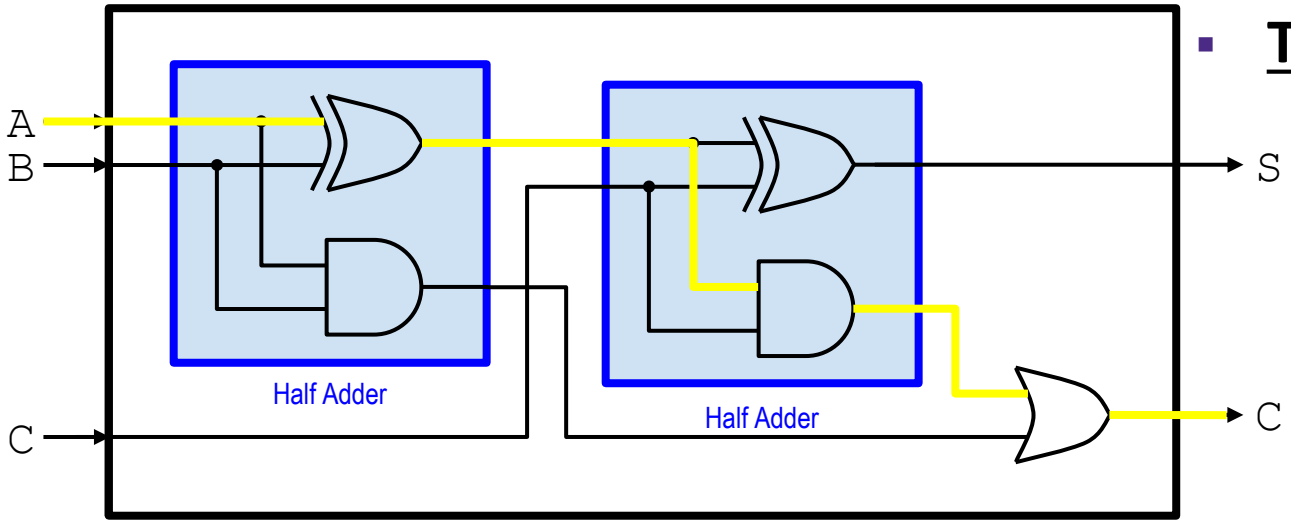
- Add A + B to get Sum (S) and Carry (C)
- Truth Table:
- Boolean Expressions:
  - S = A⊕B; C = AB
- Circuit:

A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



By inductiveload - Own work, Public Domain,
https://commons.wikimedia.org/w/index.php?curid=1023090

What is this Circuit?

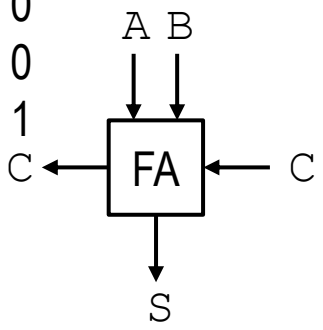


Full Adder

- **Truth table:**

A B C	C S
0 0 0	0 0
0 0 1	0 1
0 1 0	0 1
0 1 1	1 0
1 0 0	0 1
1 0 1	1 0
1 1 0	1 0
1 1 1	1 1

3-bit Addition!



Q. What's the propagation delay?

- **3 gate delays (highlighted)**

Q. What does the circuit accomplish?

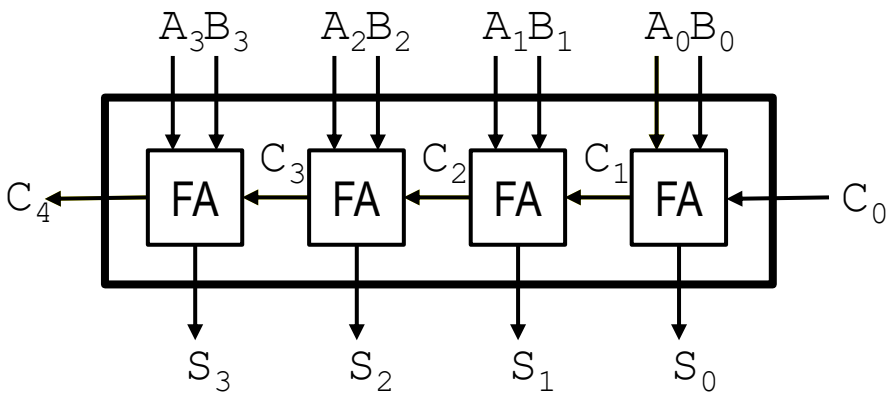
- **Algebra:**

$$S = A \oplus B \oplus C \; ; \quad C = AB + C(A \oplus B)$$

Computing with Combinational Circuits

Definition: A combinational circuit computes a pure function, i.e., its outputs react only based on its inputs. There are no feedback loops and no state information (memory) is maintained.

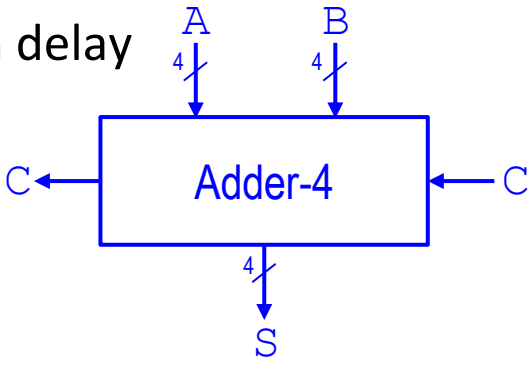
Theorem: Every Boolean function can be implemented with NAND and NOT. Circuits are modular



… a 4-bit ripple carry adder!

- Adds by columns
- Propagation delay

= 9 **(2*n* + 1)**

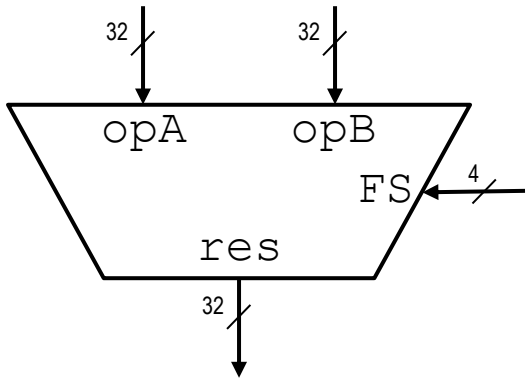


Functional Unit

Hardware circuits are fixed

- Can't adjust wires / gates while running
- Build control wires to parametrize its function

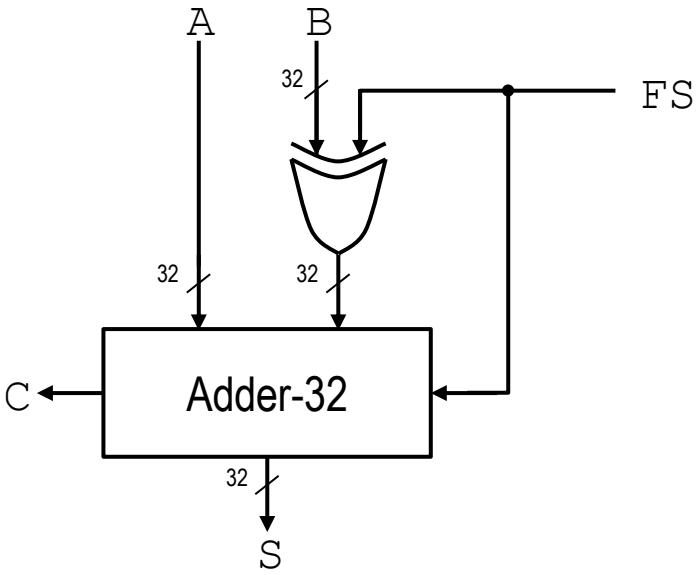
Function Unit:



Function Select:

FS	func
0001	A + B
0010	A – B
1000	A * B
0100	A ^ B
0101	A + 1
1101	B

Functional Unit: Adder-Subtractor



- if FS == 0 then
  S = A + B
- if FS == 1 then
  S = A + $\overline{B}$ + 1
  = A − B

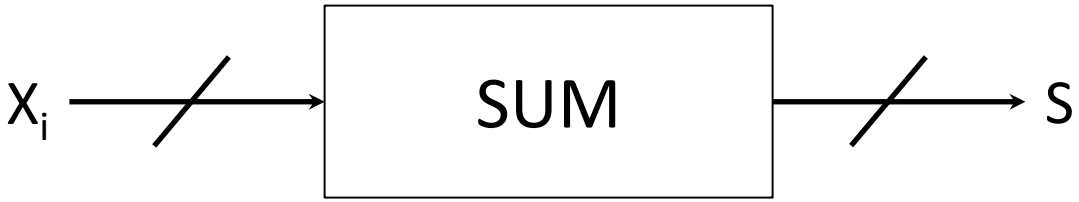
Combinational vs. Sequential Logic

- *Digital Systems* consist of two basic types of circuits:
  - Combinational Logic (CL)
    - Output is a function of the inputs only, not the history of its execution
    - Example: add A, B (ALUs)
  - Sequential Logic (SL)
    - Circuits that "remember" or store information
    - Also called "State Elements"
    - Example: Memory and registers

Sequential Logic

Accumulator Example

An example of why we would need sequential logic



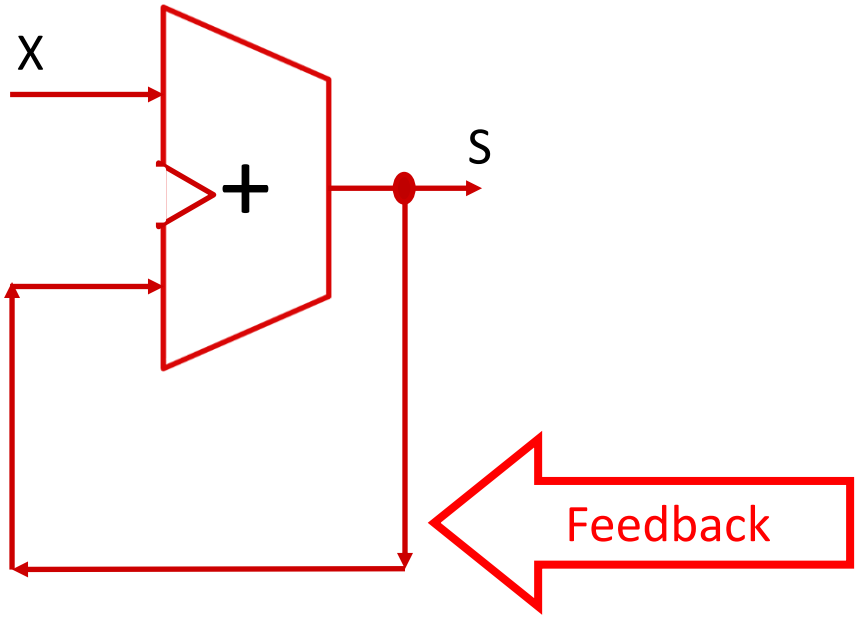
Want:

```
S=0;
for X1,X2,X3 over time...
        S = S + Xi
```

Assume:

- Each `X` value is applied in succession, one per cycle
- The sum since time 1 (cycle) is present on `S`

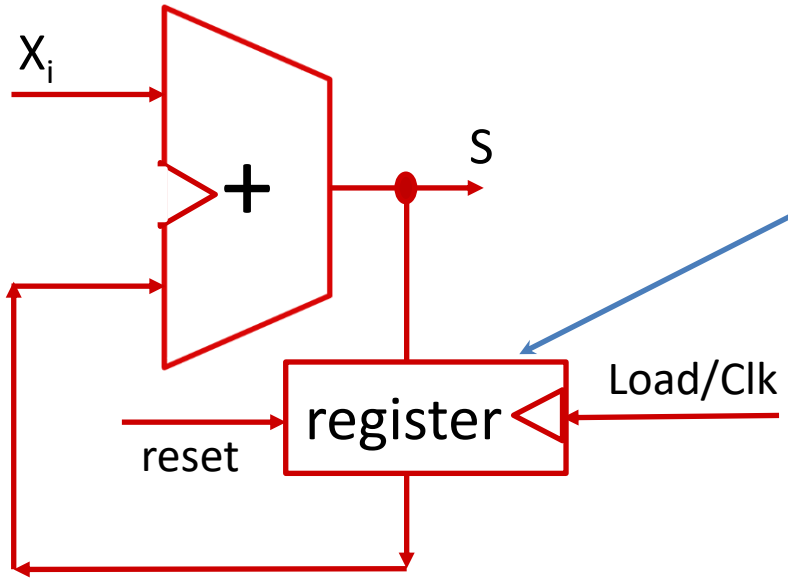
First Try: Does this work?



No!

1) How to control the next iteration of the ‘for’ loop?
2) How do we say: ‘S=0’?

Second Try: How About This?



A *Register* is the state element that is used here to hold up the transfer of data to the adder

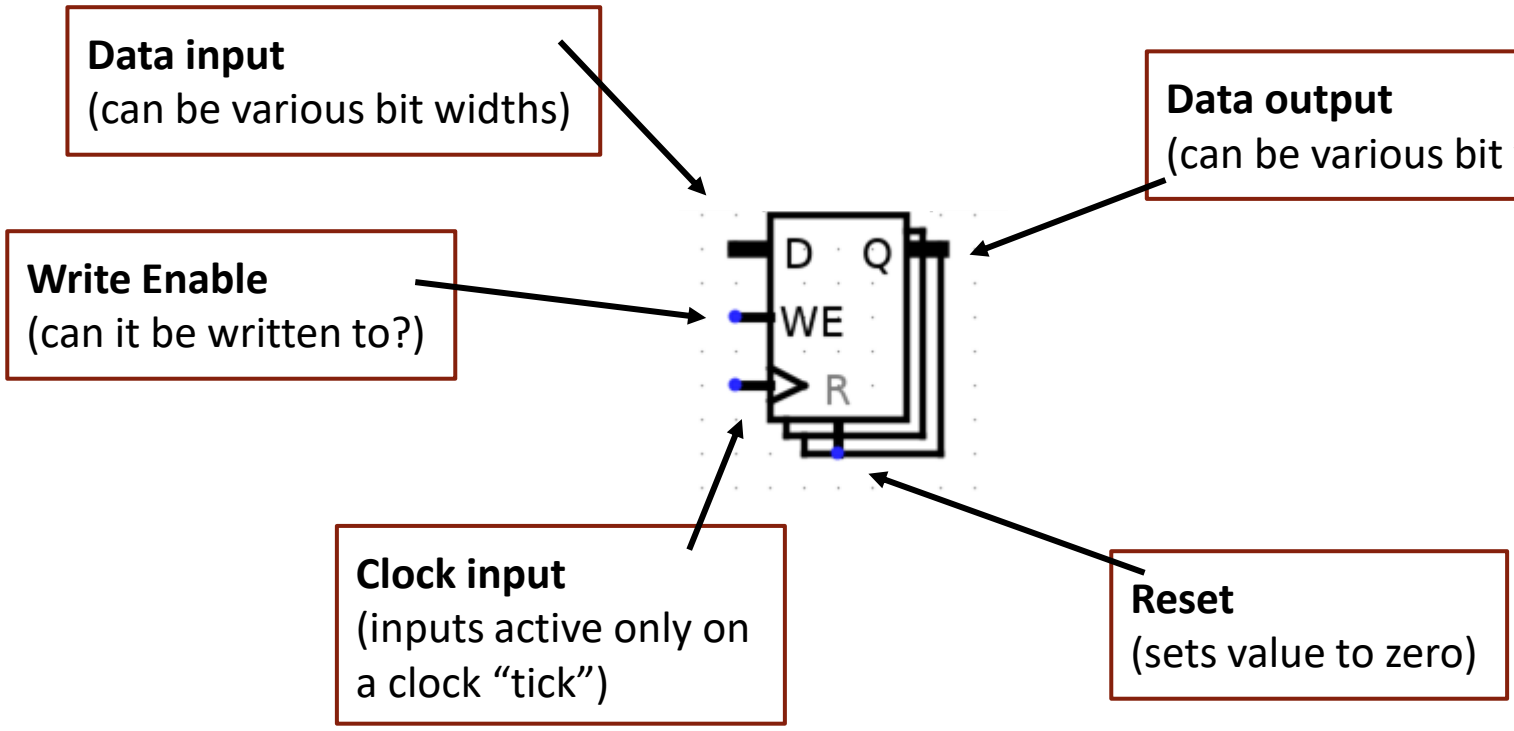
Uses for State Elements

- Place to store values for some amount of time:
  - Register files (like in RISCV)
  - Memory (caches and main memory)
- *Help control flow of information between combinational logic blocks*
  - State elements are used to hold up the movement of information at the inputs to combinational logic blocks and allow for orderly passage

Registers

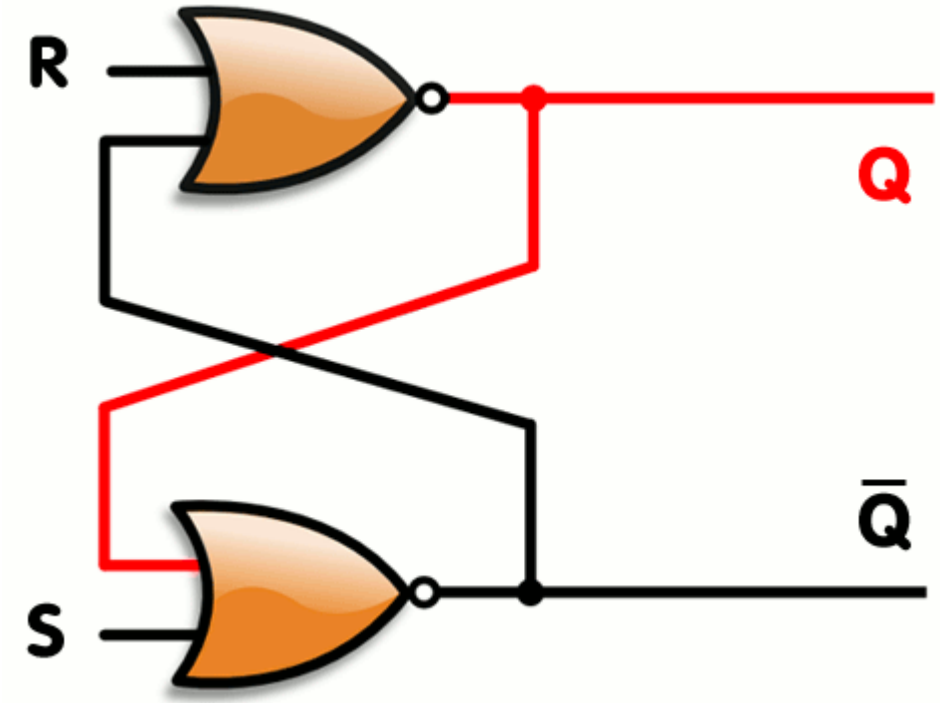
Same as registers in assembly:

- Small memory storage locations



First State Element: RS Latch

When R = 1 and S = 0 → Q is 0
When S = 1 and R = 0 → Q is 1
When both S and R are 0 → Q stays the same
When both S and R are 1 → Undefined

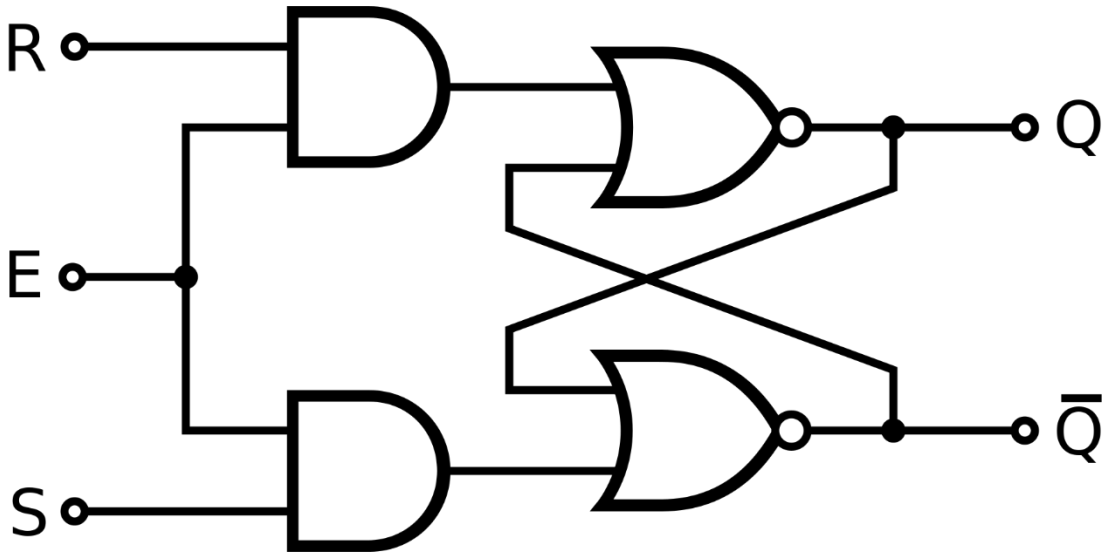


Recall:
NOR outputs 1 if both inputs are 0

By Napalm Llama - Modification of Wikimedia Commons file R-S.gif (shown below), CC BY 2.0, https://commons.wikimedia.org/w/index.php?curid=4845402

RS Latch with Enable

- Only changes state when E = 1.
- Stays the same when E = 0

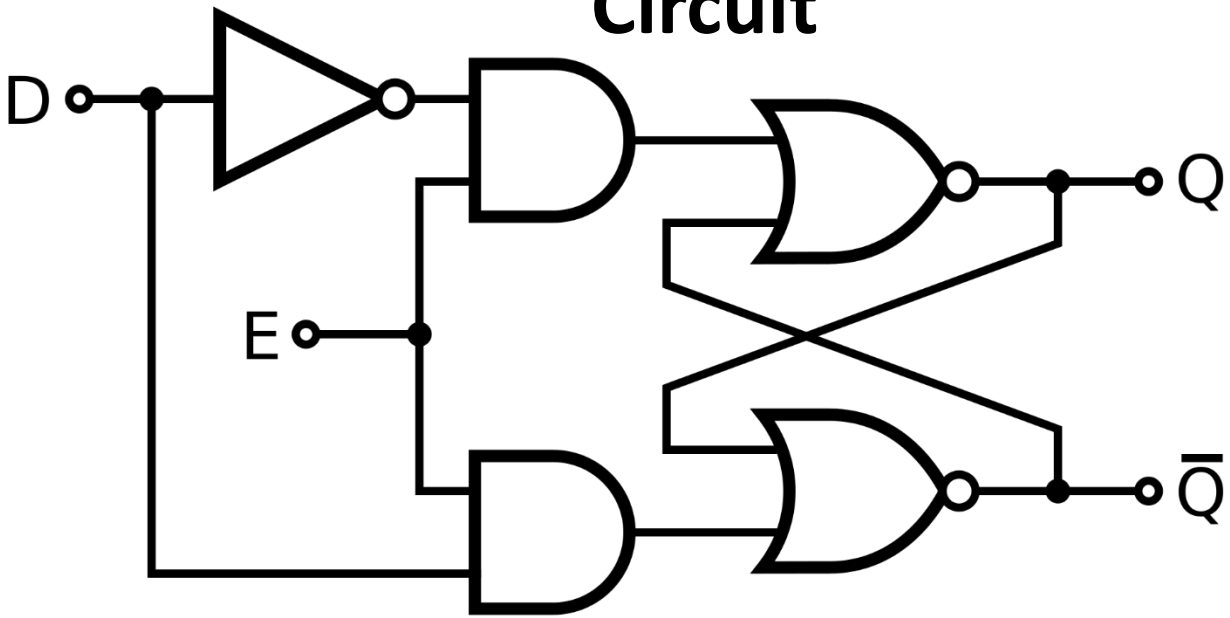


By Inductiveload - Own Drawing in Inkscape 0.43, Public Domain, https://commons.wikimedia.org/w/index.php?curid=873598

D Latch

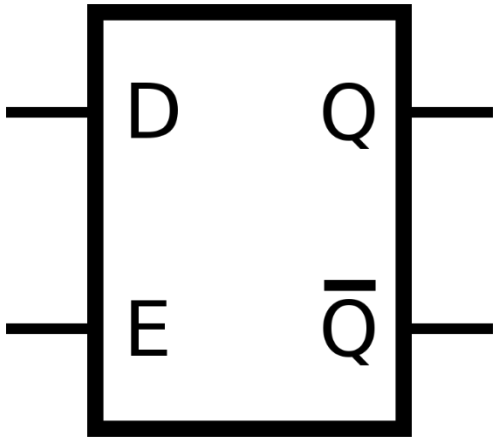
- Avoids undefined state of RS Latch when R=S=1
- Q is set to D when E = 1; Q stays the same when E = 0

Circuit



By Inductiveload - Own work, Public Domain, https://commons.wikimedia.org/w/index.php?curid=6712572

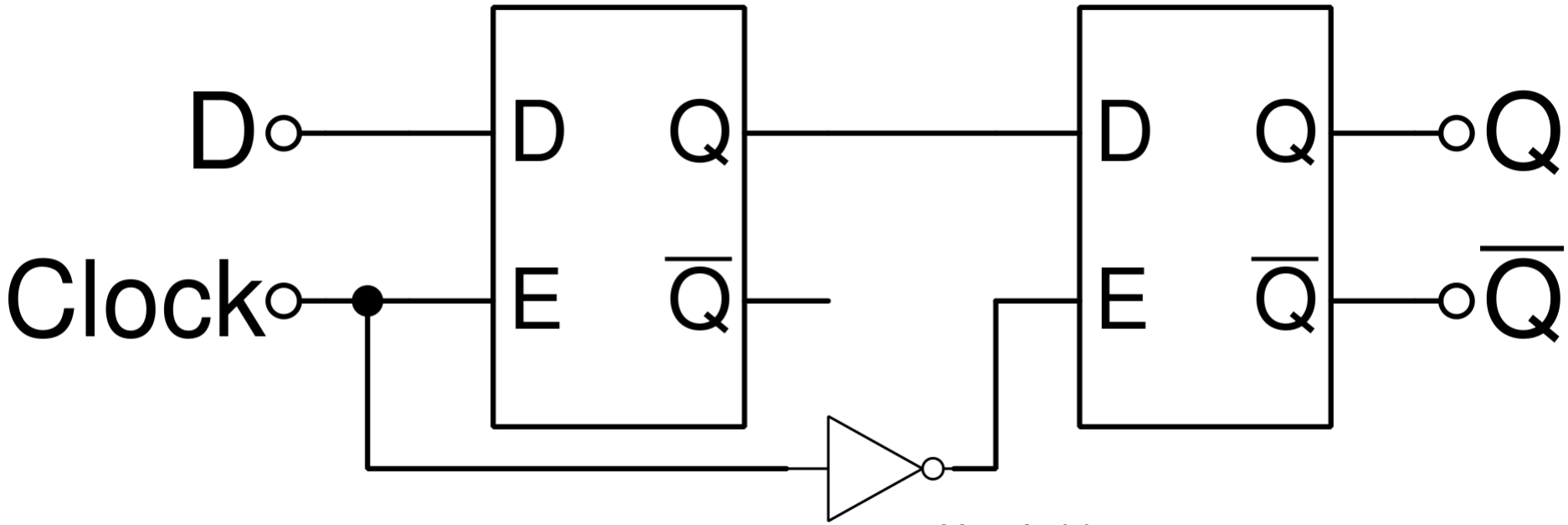
Schematic



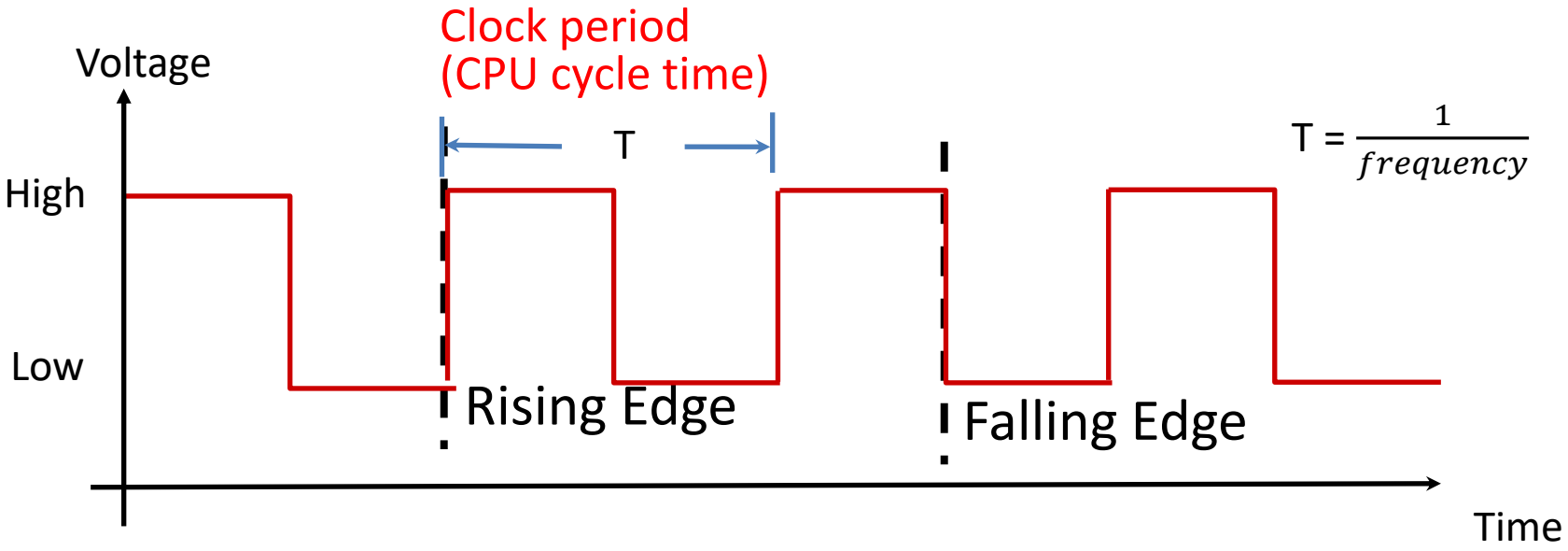
By Inductiveload - Own work, Public Domain, https://commons.wikimedia.org/w/index.php?curid=6712594

D Flip-Flop

- Changes state only on falling edge of Clock (i.e., Clock changes from 1 to 0)
- Use $\overline{\text{Clock}}$ to change on rising edge



Signals and Waveforms: Clocks

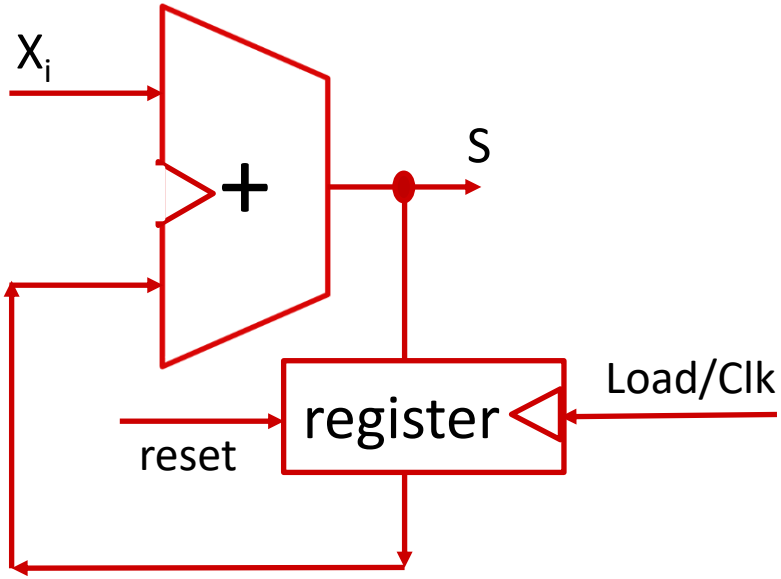


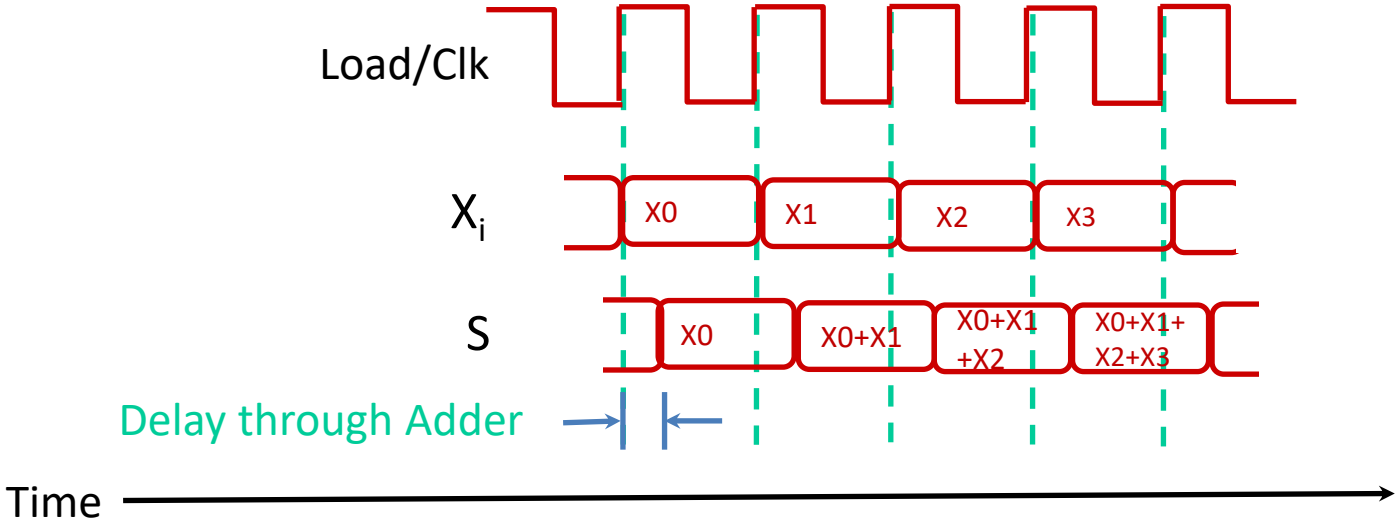
- Signals transmitted over wires continuously
- Transmission is effectively instantaneously
  - Implies that any wire only contains one value at any given time

Dealing with Waveform Diagrams

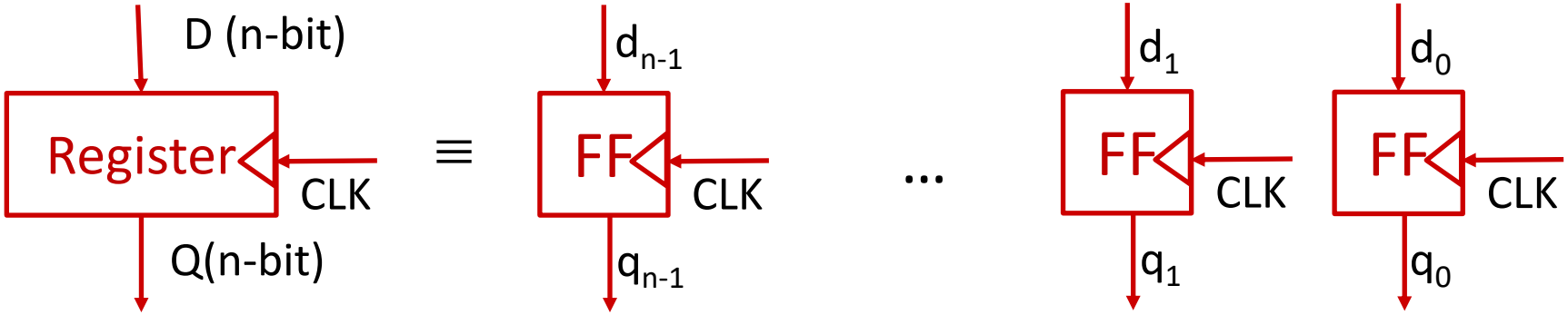
- Easiest to start with CLK on top
  - Solve signal by signal, from inputs to outputs
  - Can only draw the waveform for a signal if *all* of its input waveforms are drawn
- When does a signal update?
  - A *state element* updates based on CLK triggers
  - A *combinational element* updates ANY time ANY of its inputs changes

Accumulator 2nd Try: How About This?

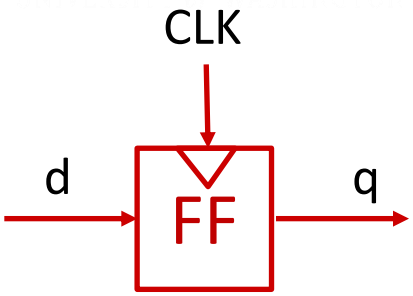




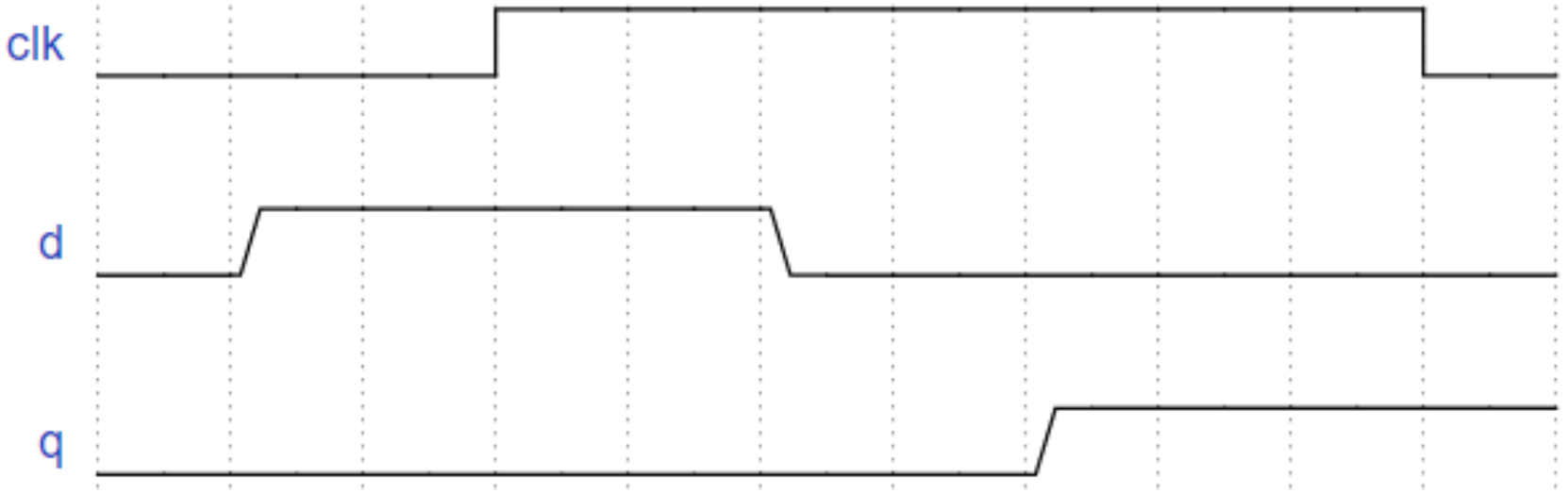
Register Internals

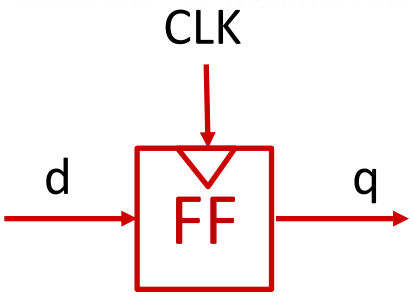


- N-bit register ≡ n instances of a *"Flip-Flop"*
  - Output flips and flops between 0 and 1
- Specifically this is a "D-type Flip-Flop"
  - D is "data input", Q is "data output"
  - A group of wires when interpreted as a bit field is called a *bus*

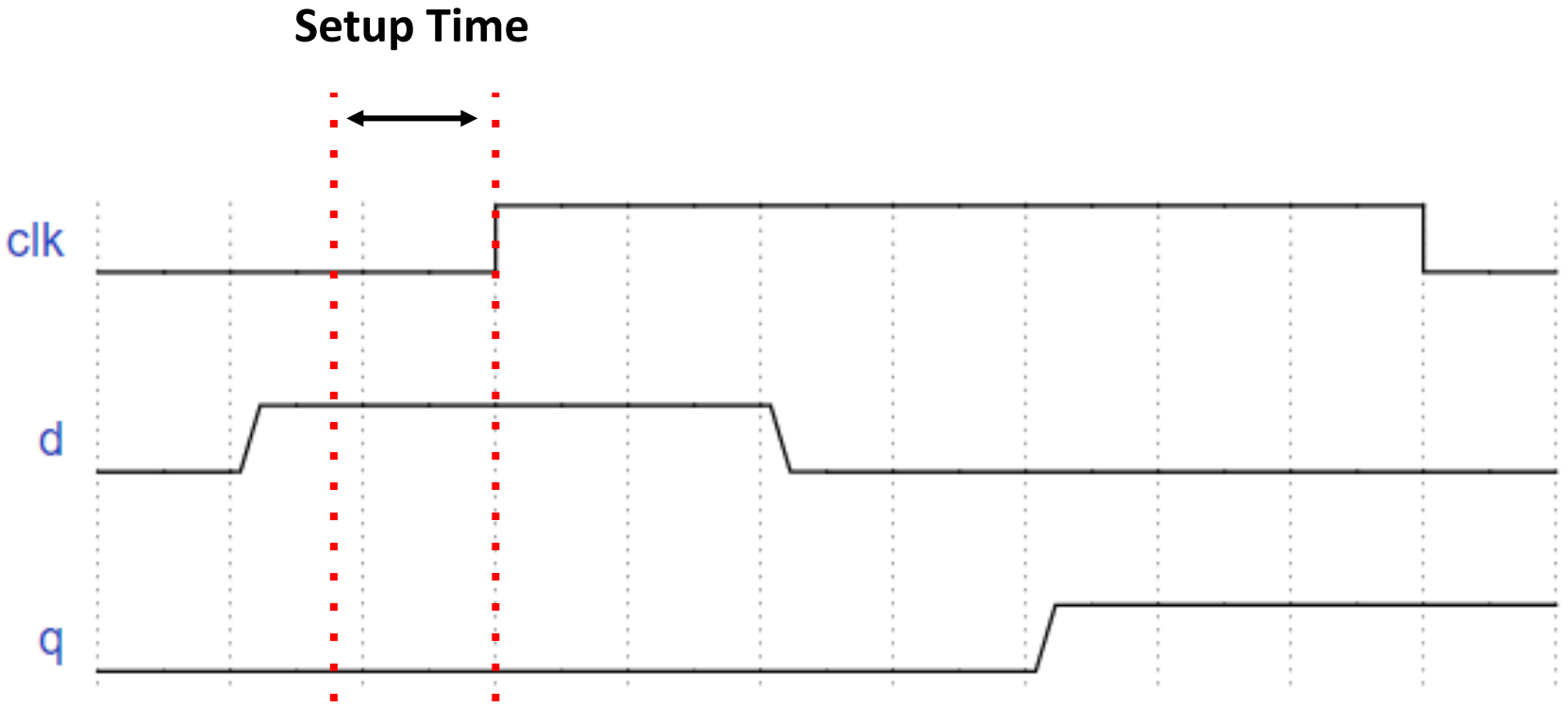


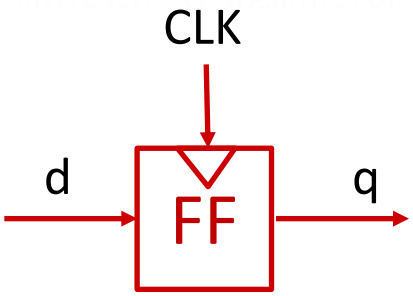
Flip-Flop Timing Behavior



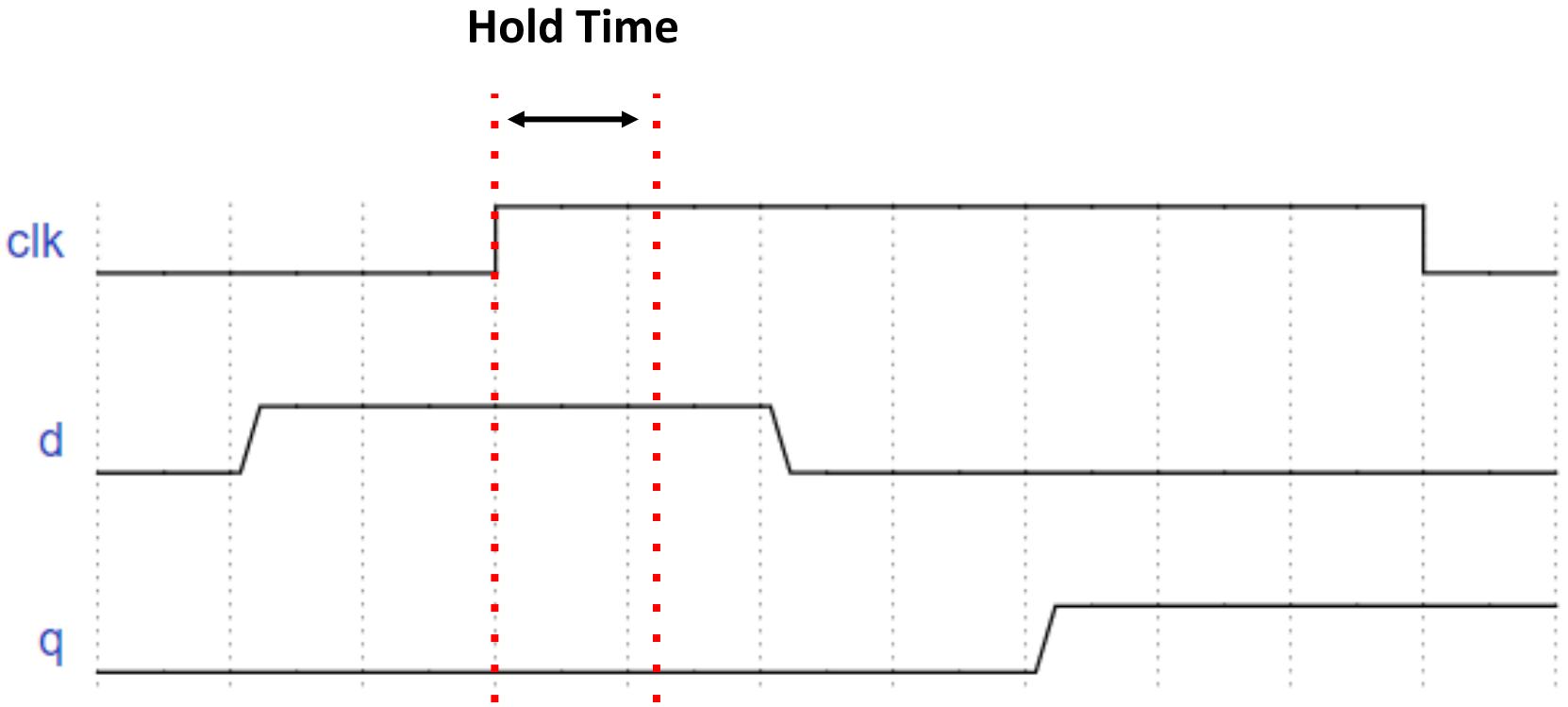


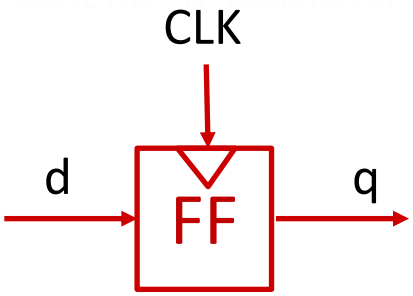
Flip-Flop Timing Behavior



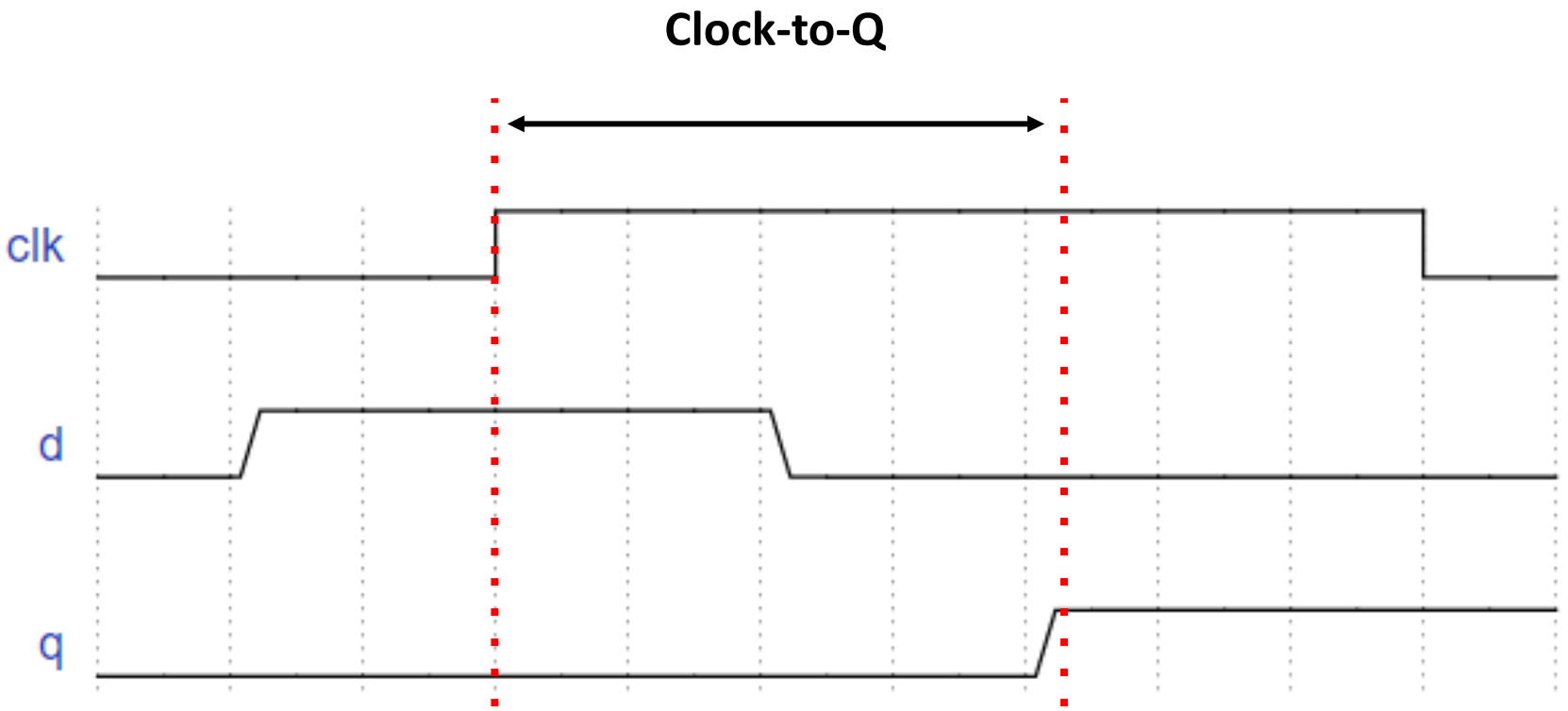


Flip-Flop Timing Behavior

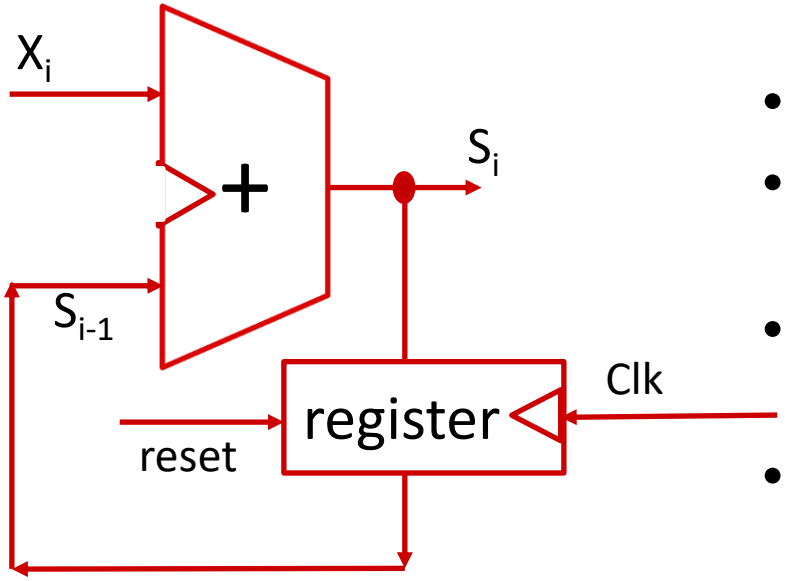




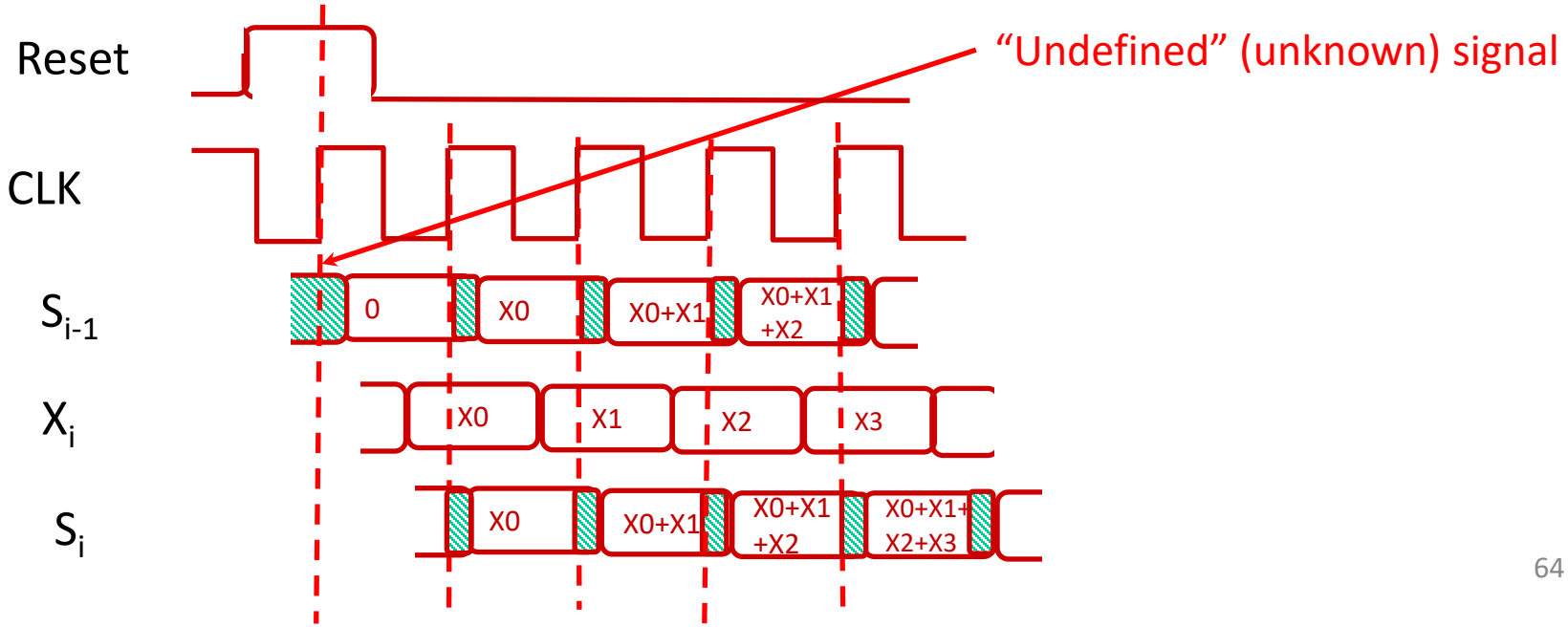
Flip-Flop Timing Behavior



Accumulator Revisited: Proper Timing



- Reset signal shown
- In practice X_i might not arrive to the adder at the same time as S_{i-1}
- S_i temporarily is wrong, but register always captures correct value
- In good circuits, instability never happens around rising edge of CLK

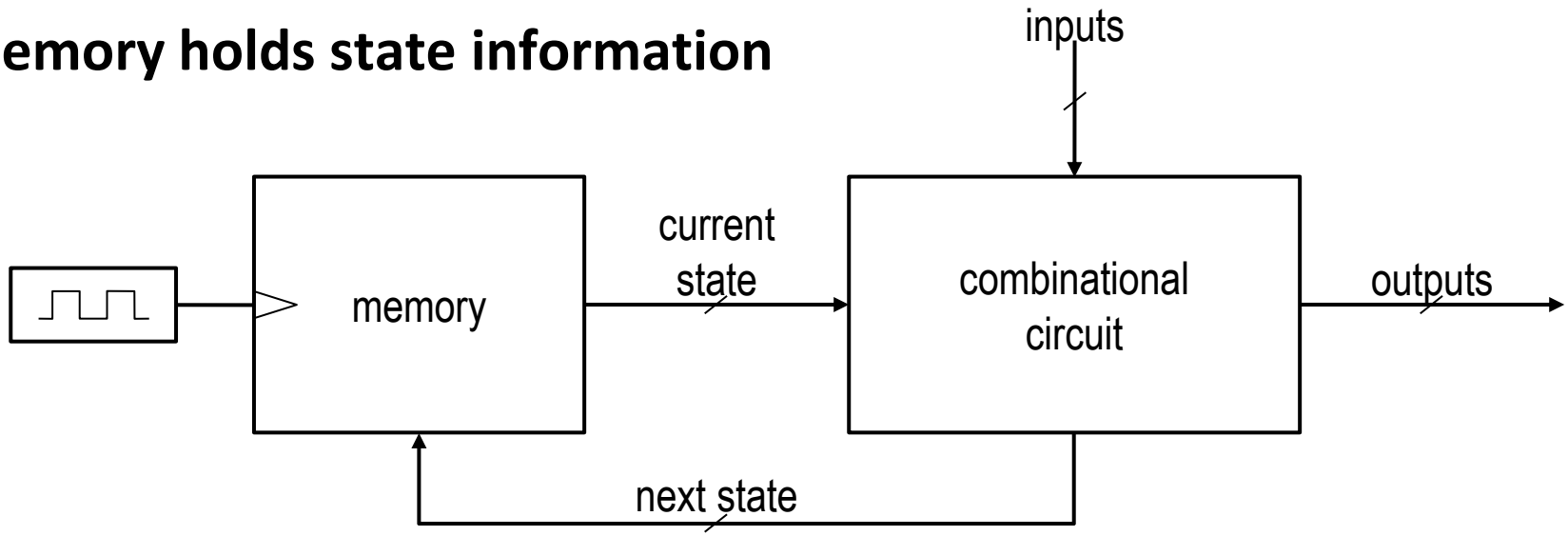


Timing Terms

- Clock: Steady square wave that synchronizes system
- Register: Several bits of state that samples on rising edge of Clock (positive edge-triggered); also has RESET
- Setup Time: When input must be stable *before* Clock trigger
- Hold Time: When input must be stable *after* Clock trigger
- Clock-to-Q Delay: How long it takes output to change from Clock trigger

Digital State Machines

Memory holds state information



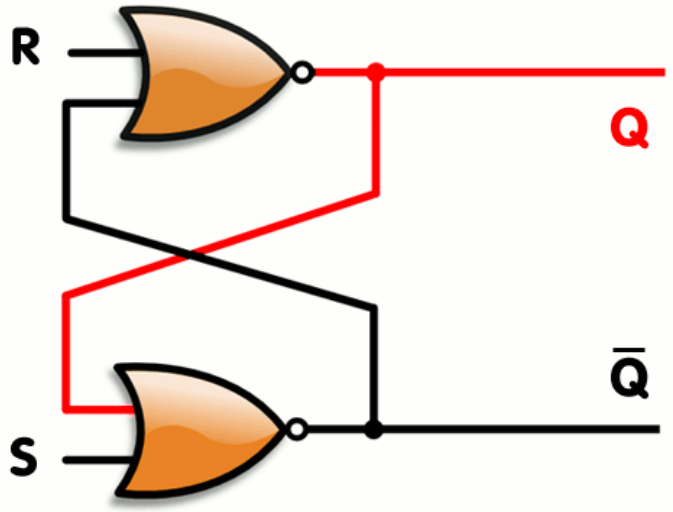
- **compute next state based on (current state, inputs)**
- **compute outputs based on (current state, inputs)**

Q. What does this imply about the clock period?

- **clock period must exceed (t_{pd} of combinational circuit + t_{pd} of registers) where t_{pd} is propagational delay**

Waveform Example: RS Latch

a	b	a NOR b
0	0	1
0	1	0
1	0	0
1	1	0



By Napalm Llama - Modification of Wikimedia Commons file R-S.gif (shown below), CC BY 2.0, https://commons.wikimedia.org/w/index.php?curid=4845402

Q

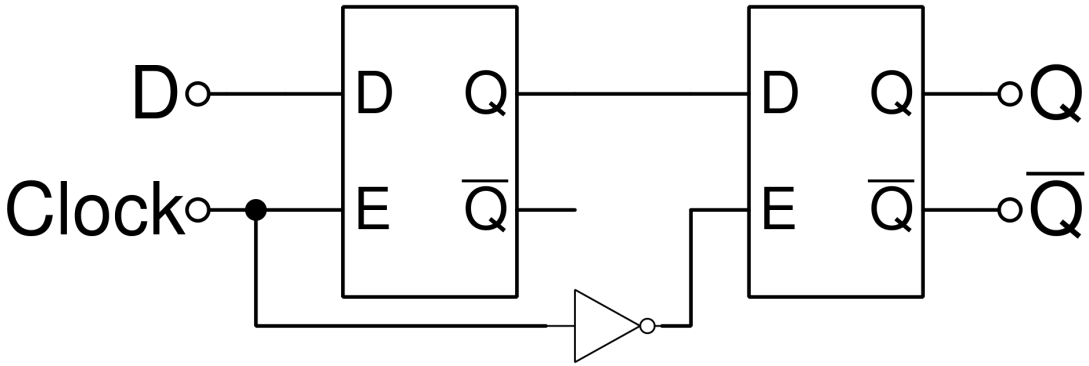
$\bar{Q}$

R

S

Time

Waveform Example: D Flip-Flop

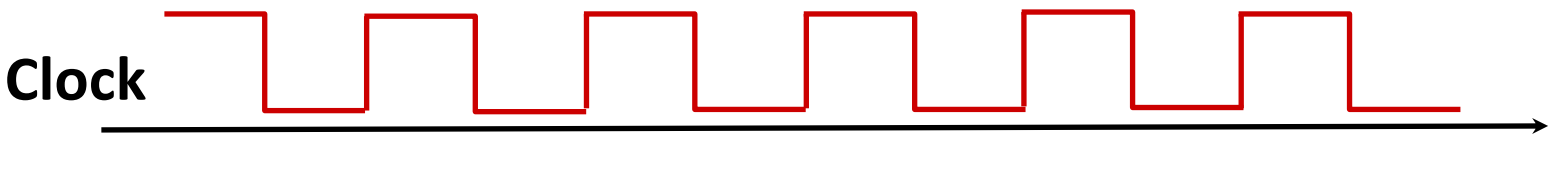


By Nolanjshettle at English Wikipedia, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=40852395

Q

$\overline{Q}$

D



CPU Hardware

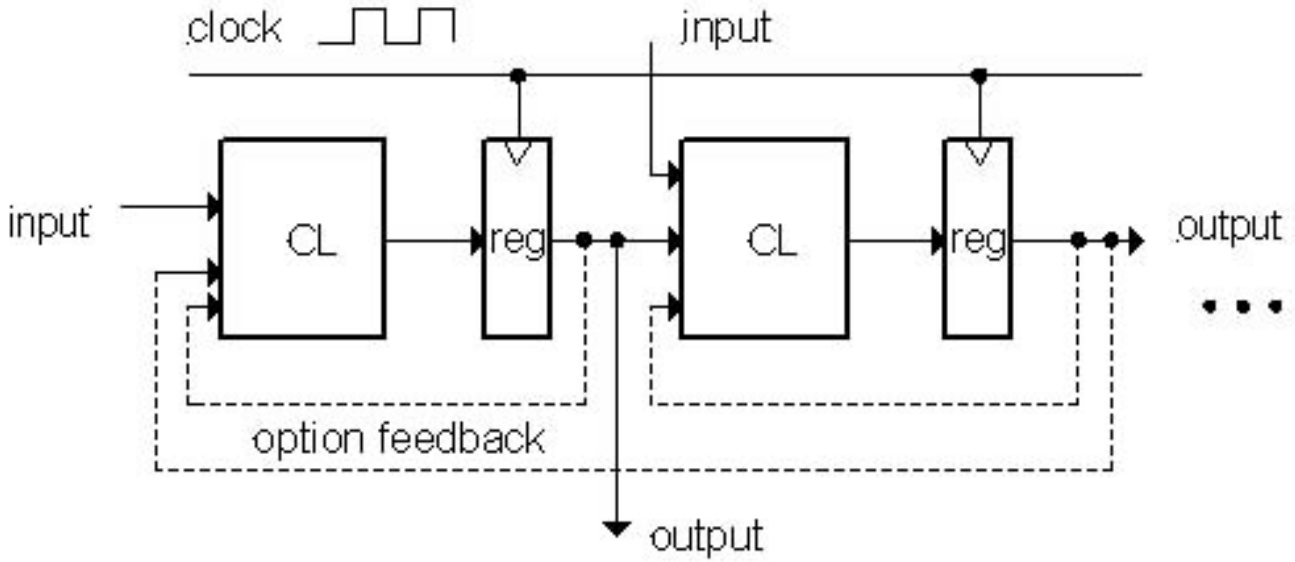
Goal: Given an instruction set architecture, construct a machine that reliably executes instructions.

Design choices will influence speed of instructions:

- **Some instructions will be faster than others**
- **Order of instructions may matter**
- **Order of memory accesses may matter**

"conflicts" or "hazards"

Model for Synchronous Systems

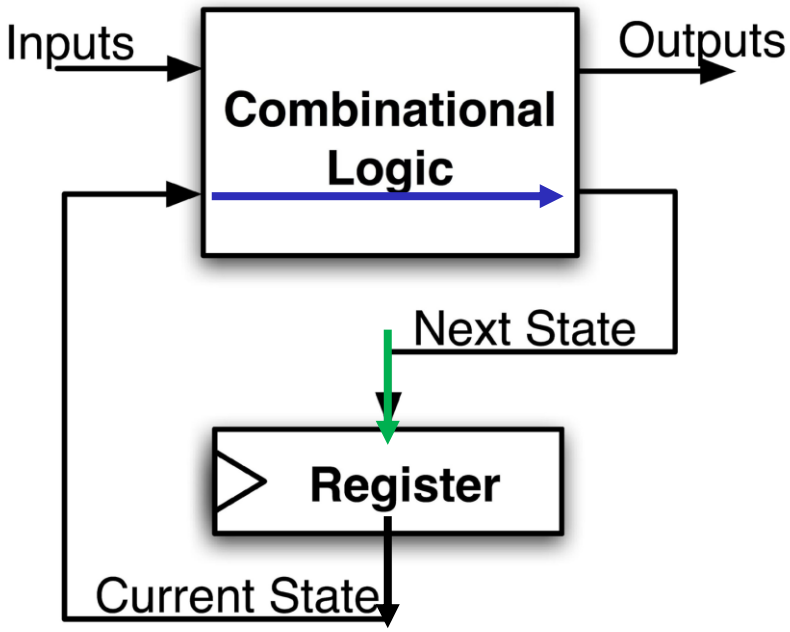


- Combinational logic blocks separated by registers
  - Clock signal connects only to sequential logic elements
  - Feedback is optional depending on application
- How do we ensure proper behavior?
  - How fast can we run our clock?

Maximum Clock Frequency

- What is the max frequency of this circuit?
  - Limited by how much time needed to get correct Next State to Register (t_{setup} constraint)

Assumes Max Delay > Hold Time



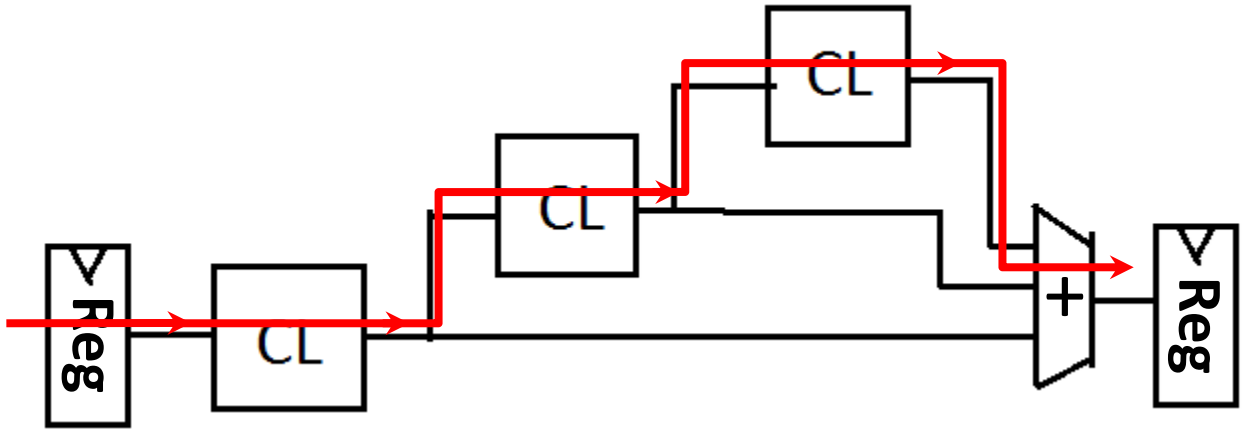
Max Delay = CLK-to-Q Delay
+ CL Delay
+ Setup Time

Min Period = Max Delay

Max Freq = 1/Min Period

The Critical Path

- The *critical path* is the longest delay between *any* two registers in a circuit
- The clock period must be *longer* than this critical path, or the signal will not propagate properly to that next register



How to get faster clock frequency?

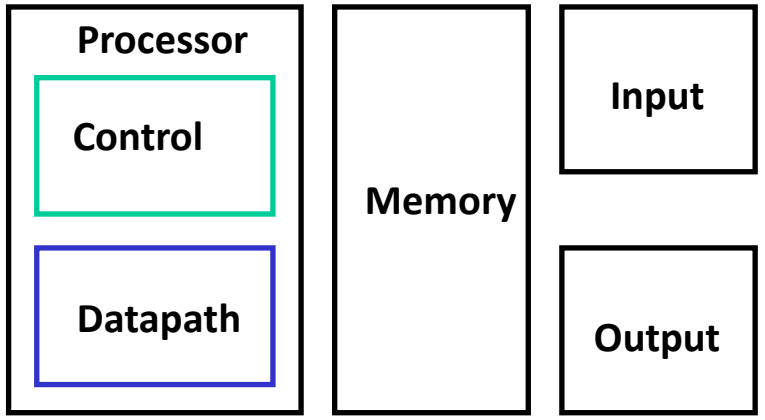
Pipelining!

- Split operation into smaller parts and add registers between each two consecutive parts (next week).

RISC-V *CPU Datapath, Control Intro*

CPU Design Principles

1) Analyze instruction set → datapath requirements
2) Select set of datapath components & establish clock methodology
3) Assemble datapath meeting the requirements



4) Analyze implementation of each instruction to determine setting of control points that effects the register transfer
5) Assemble the control logic
   - Formulate Logic Equations
   - Design Circuits

RISC-V Single-Cycle CPU

- Universal datapath
  - Capable of executing all RISC-V instructions in one cycle each
  - Not all units (hardware) used by all instructions
- 5 Phases of execution
  - IF (Instruction Fetch), ID (Instruction Decode), EX (Execute), MEM (Memory), WB (Write Back)
  - Not all instructions are active in all phases (except for loads!)
- Controller specifies how to execute instructions

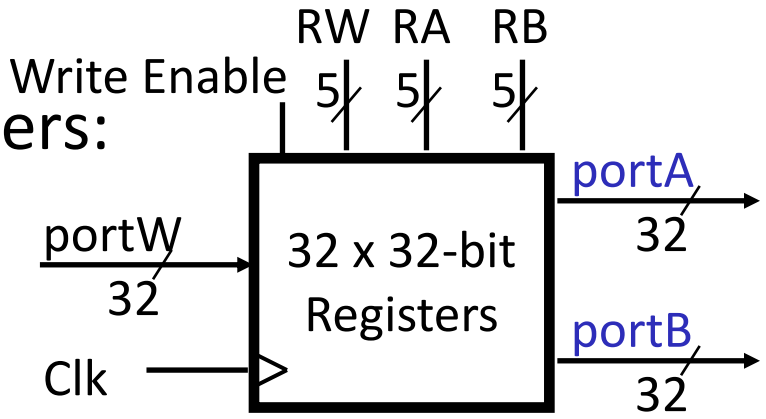
RISC-V CPU in two parts

- ***Central Processing Unit (CPU):***
  - *Datapath:* Contains the hardware necessary to perform operations required by the processor
    - Reacts to what the controller tells it. (i.e., "I was told to do an add, so I"ll feed these arguments through an adder)
  - *Control:* Decides what each piece of the datapath should do
    - What operation am I performing? Do I need to get info from memory? Should I write to a register? Which register?
    - Has to make decisions based on the input instruction only.

Design Principles

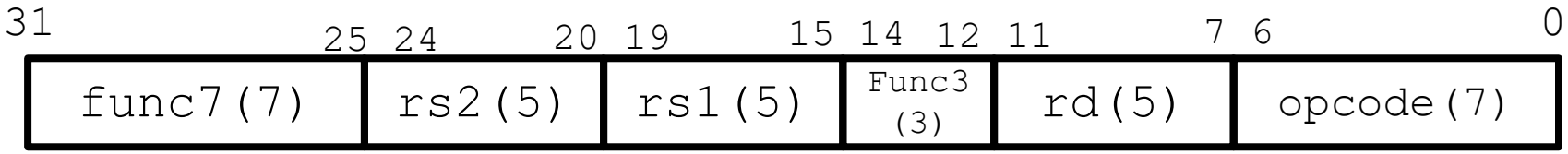
- Determining control signals
  - Any time a datapath element has an input that changes behavior, it requires a control signal (e.g. ALU operation, read/write)
  - Any time you need to pass a different input based on the instruction, add a **MUX** with a control signal as the selector (e.g. next PC, ALU input, register to write to)
- Control signals will change based on exact datapath
- Datapath will change based on ISA

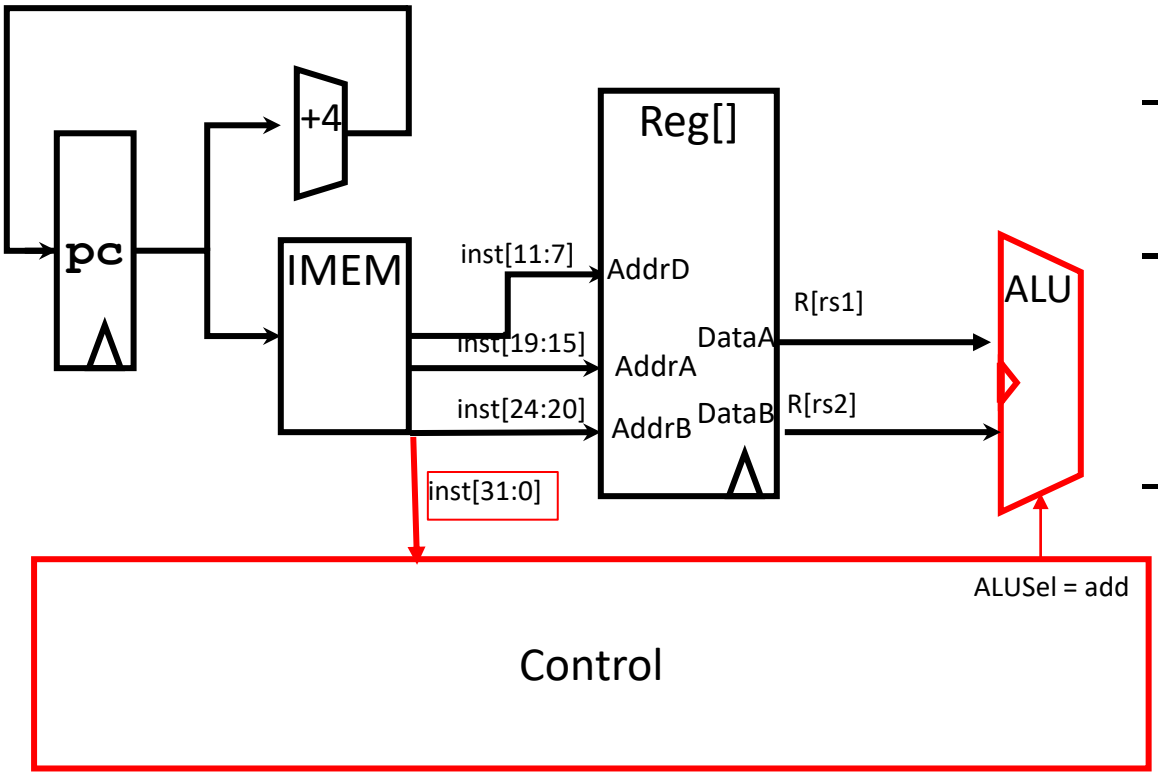
Storage Element: Register File



- *Register File* consists of 32 registers:
  - Output ports portA and portB
  - Input port portW
- Register selection
  - Place data of register RA (number) onto portA
  - Place data of register RB (number) onto portB
  - Store data on portW into register RW (number) when Write Enable is 1
- Clock input (CLK)
  - CLK is passed to all internal registers so they can be written to if they match RW and Write Enable is 1

Implementing R-Types

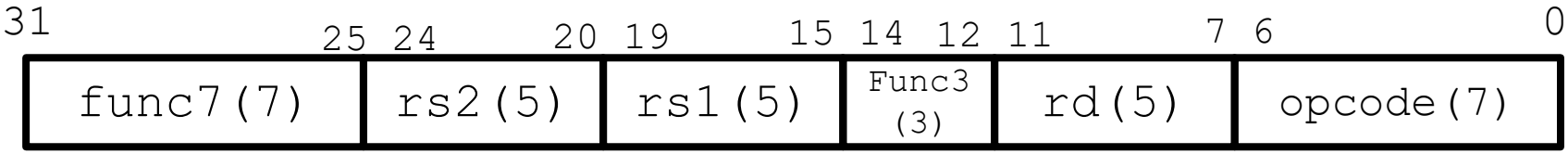


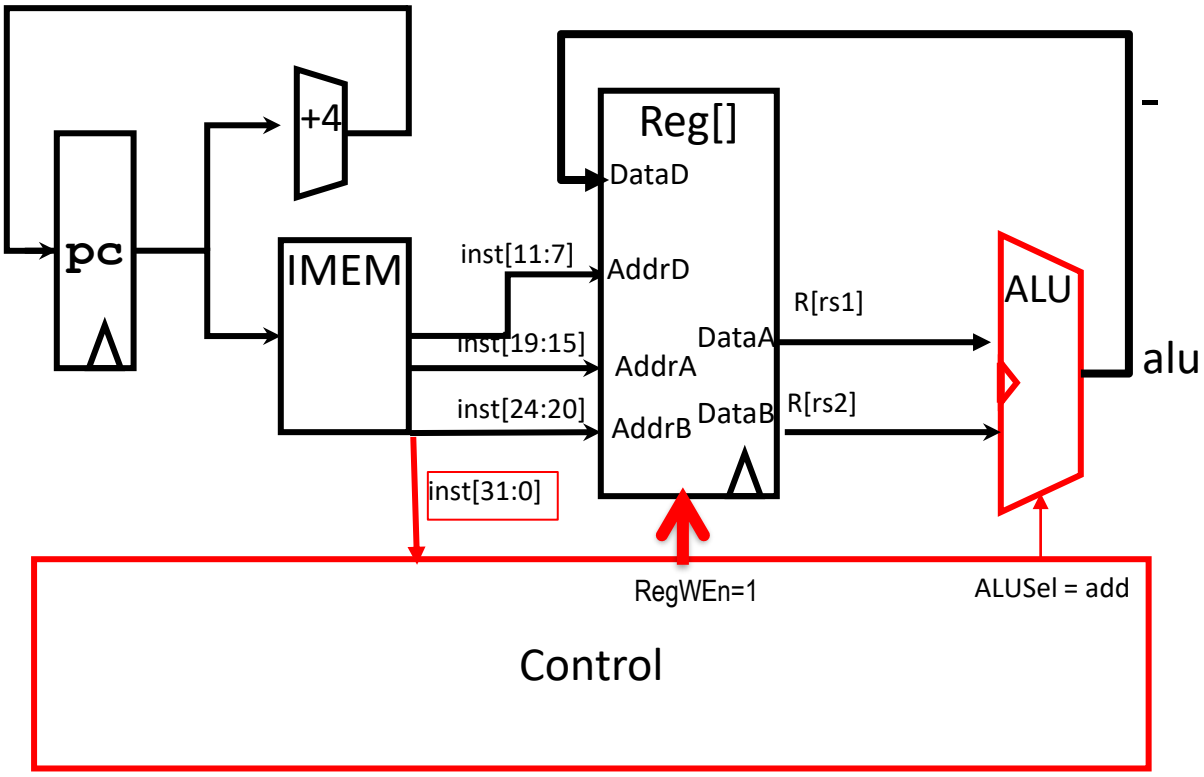


Perform operation

- New hardware: ALU (Arithmetic Logic Unit)
- Abstraction for adders, multipliers, dividers, etc.
- How do we know what operation to execute?
  - Our first control bit! ALUSel(ect)

Implementing R-Types

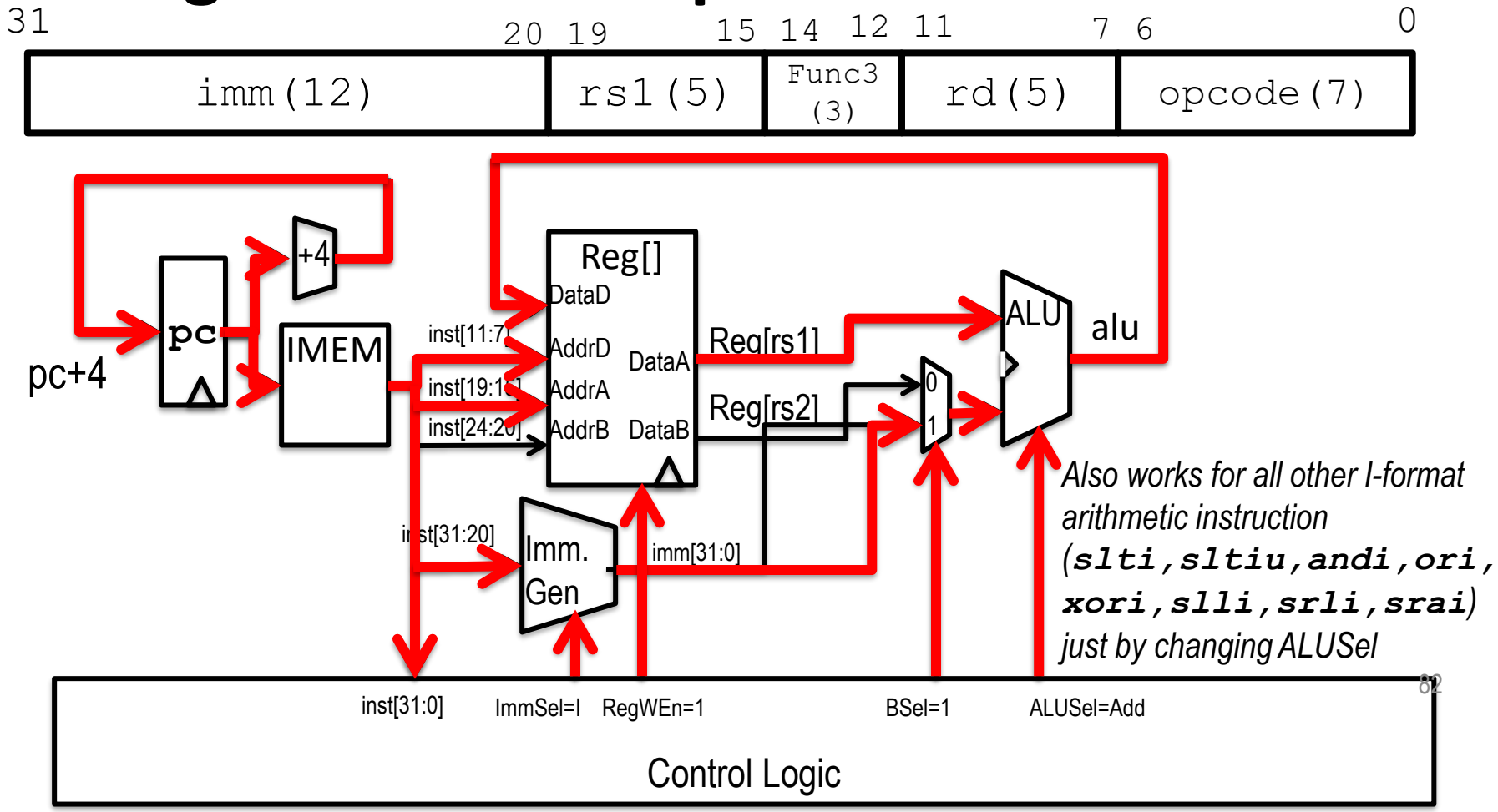




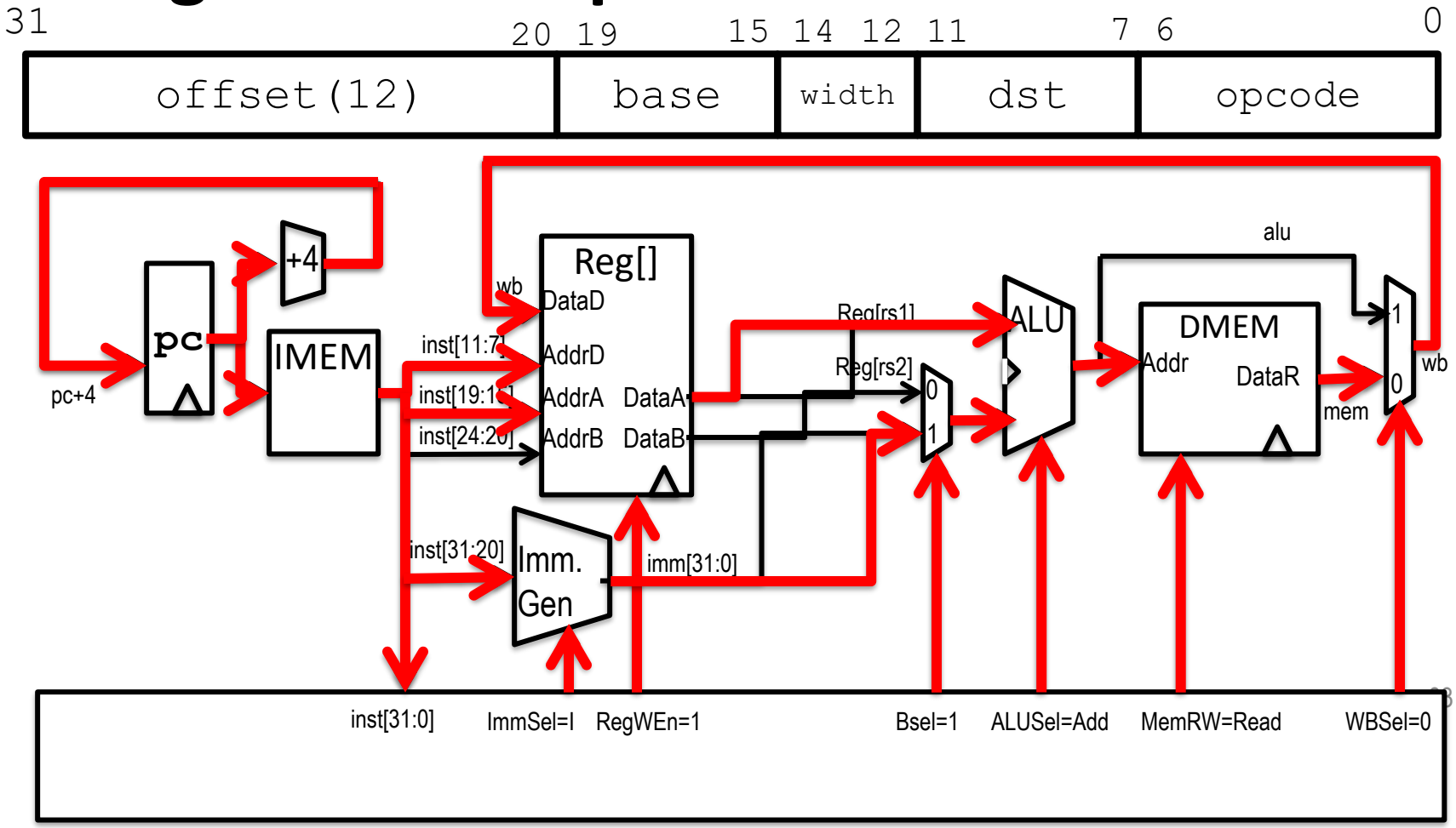
Perform operation

- Write ALU result to rd

Adding addi to datapath

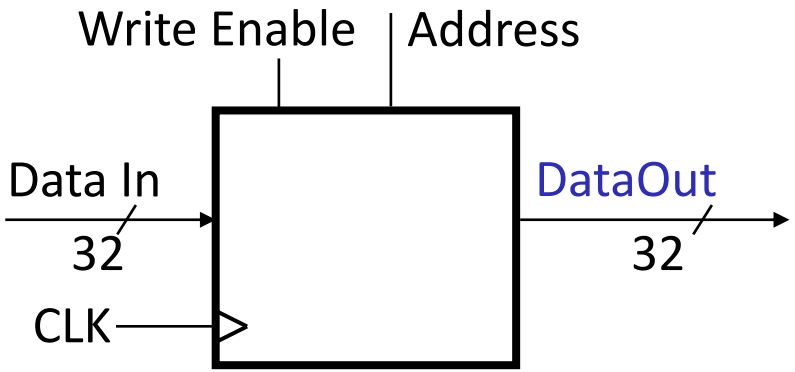


Adding `lw` to datapath

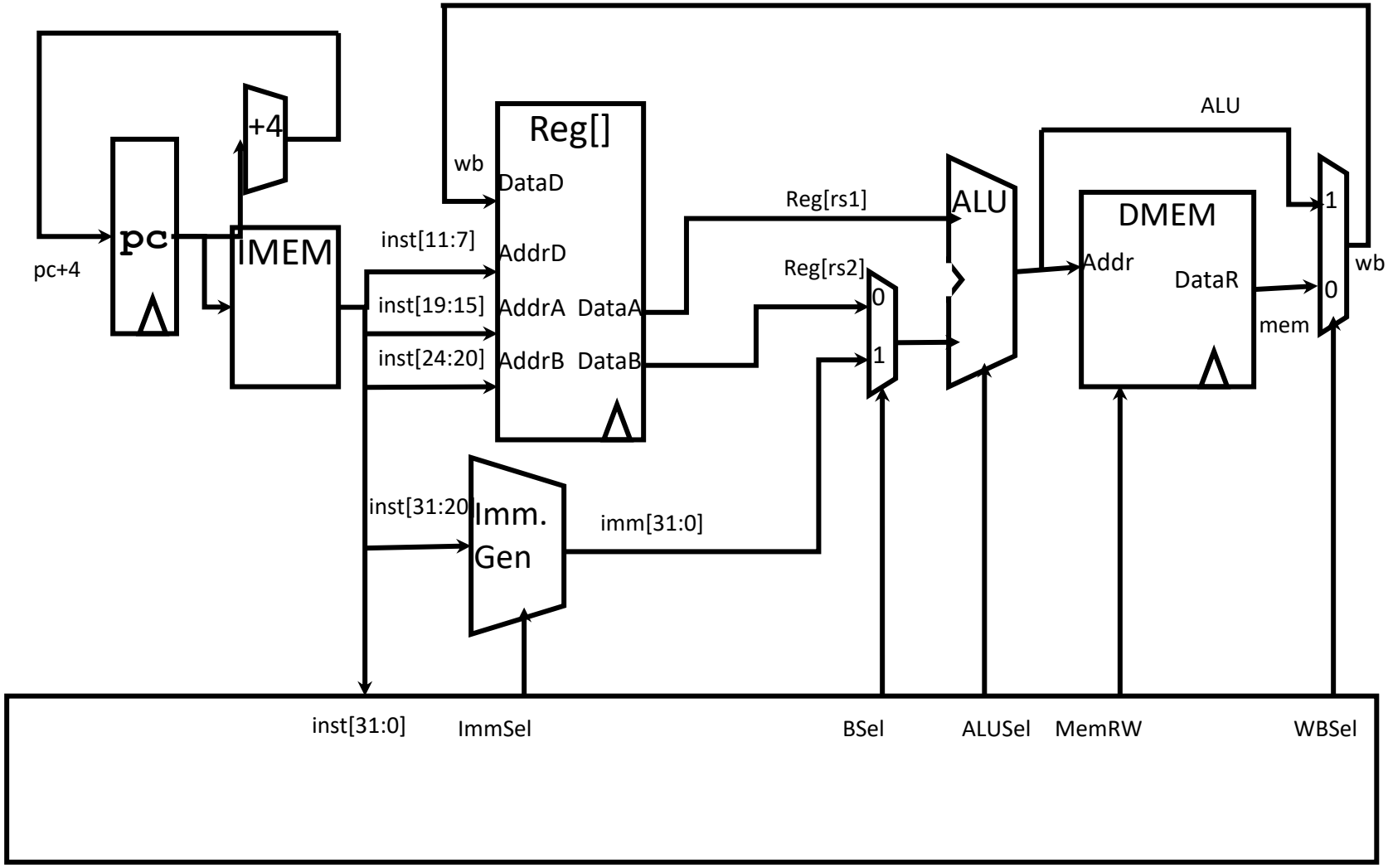


Storage Element: Idealized Memory

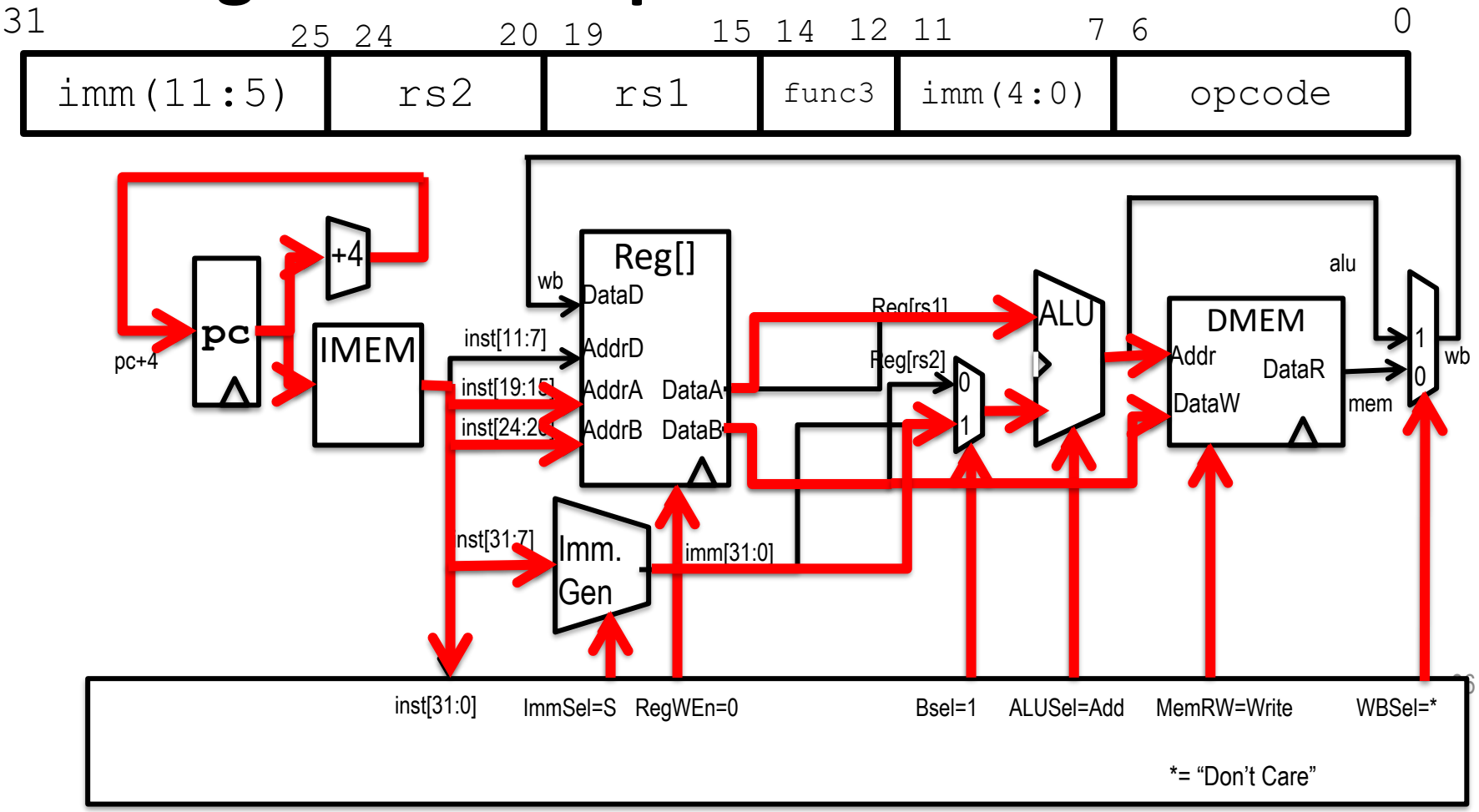
- Memory (idealized)
  - One input port: Data In
  - One output port: Data Out
- Memory access:
  - Read: Write Enable = 0, data at Address is placed on Data Out
  - Write: Write Enable = 1, Data In written to Address
- Clock input (CLK)
  - CLK input is a factor ONLY during write operation
  - During read, behaves as a combinational logic block: Address valid → Data Out valid after "access time"



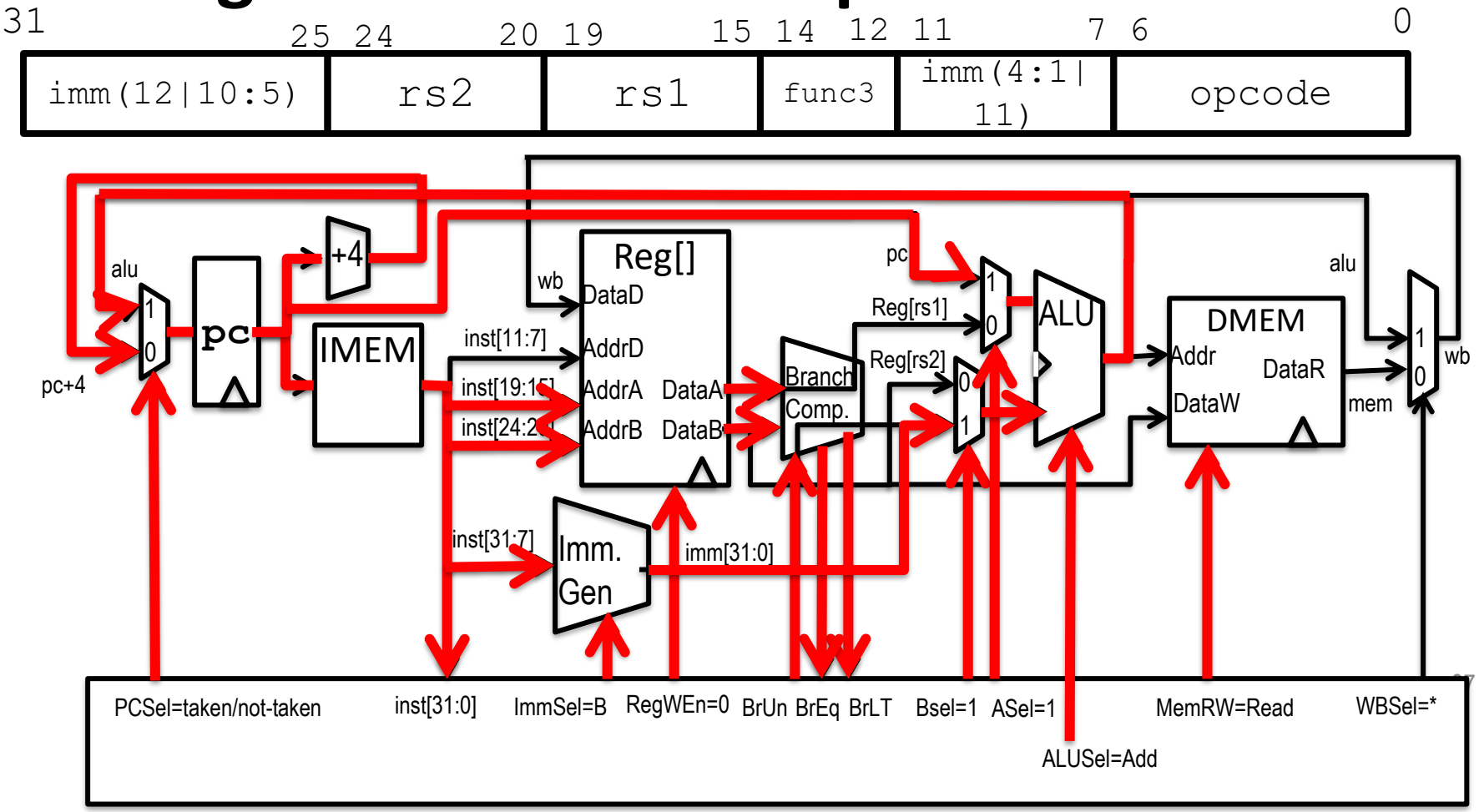
Current Datapath



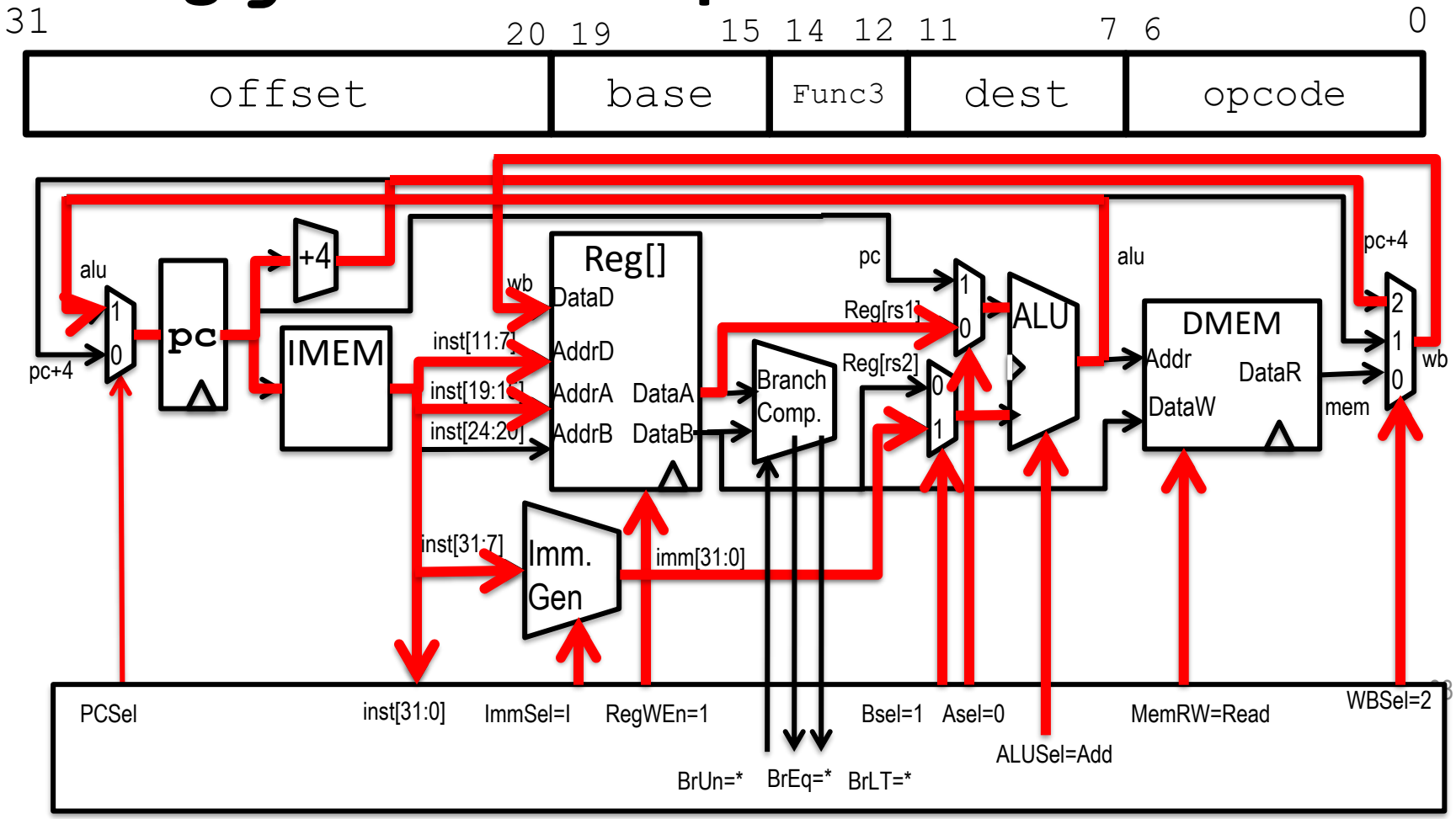
Adding sw to datapath



Adding branches to datapath

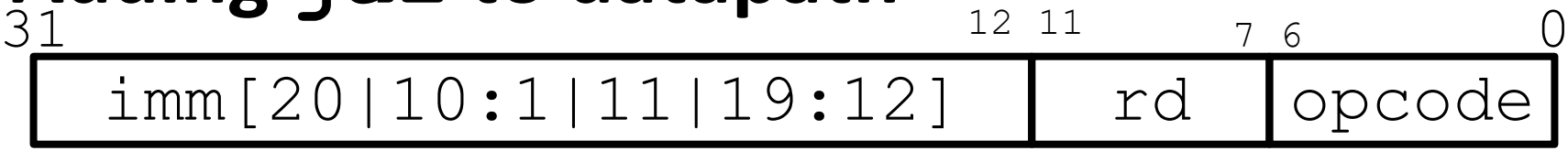


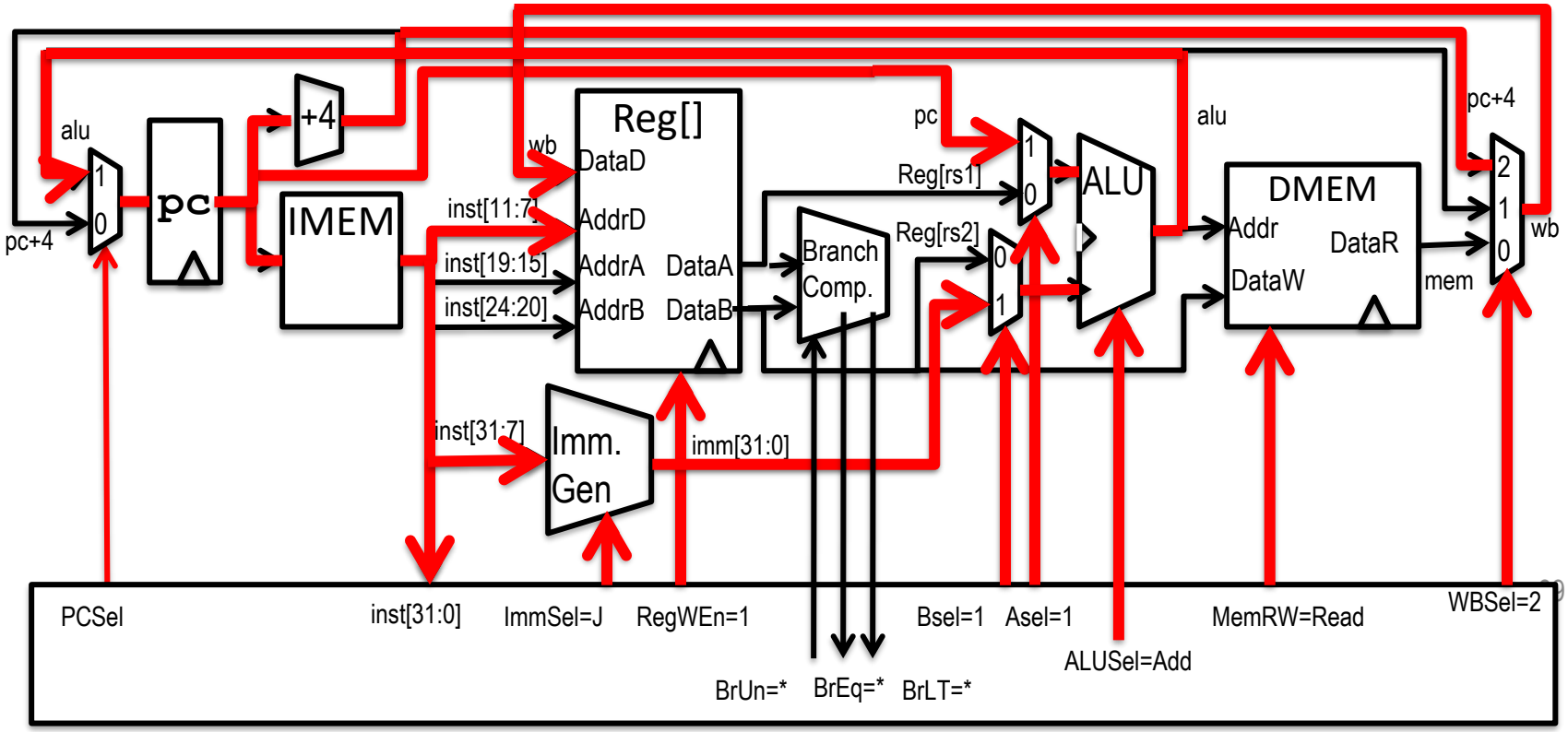
Adding `jalr` to datapath



- Writes PC+4 to `dest` (return address)
- Sets PC = `base` + `offset`

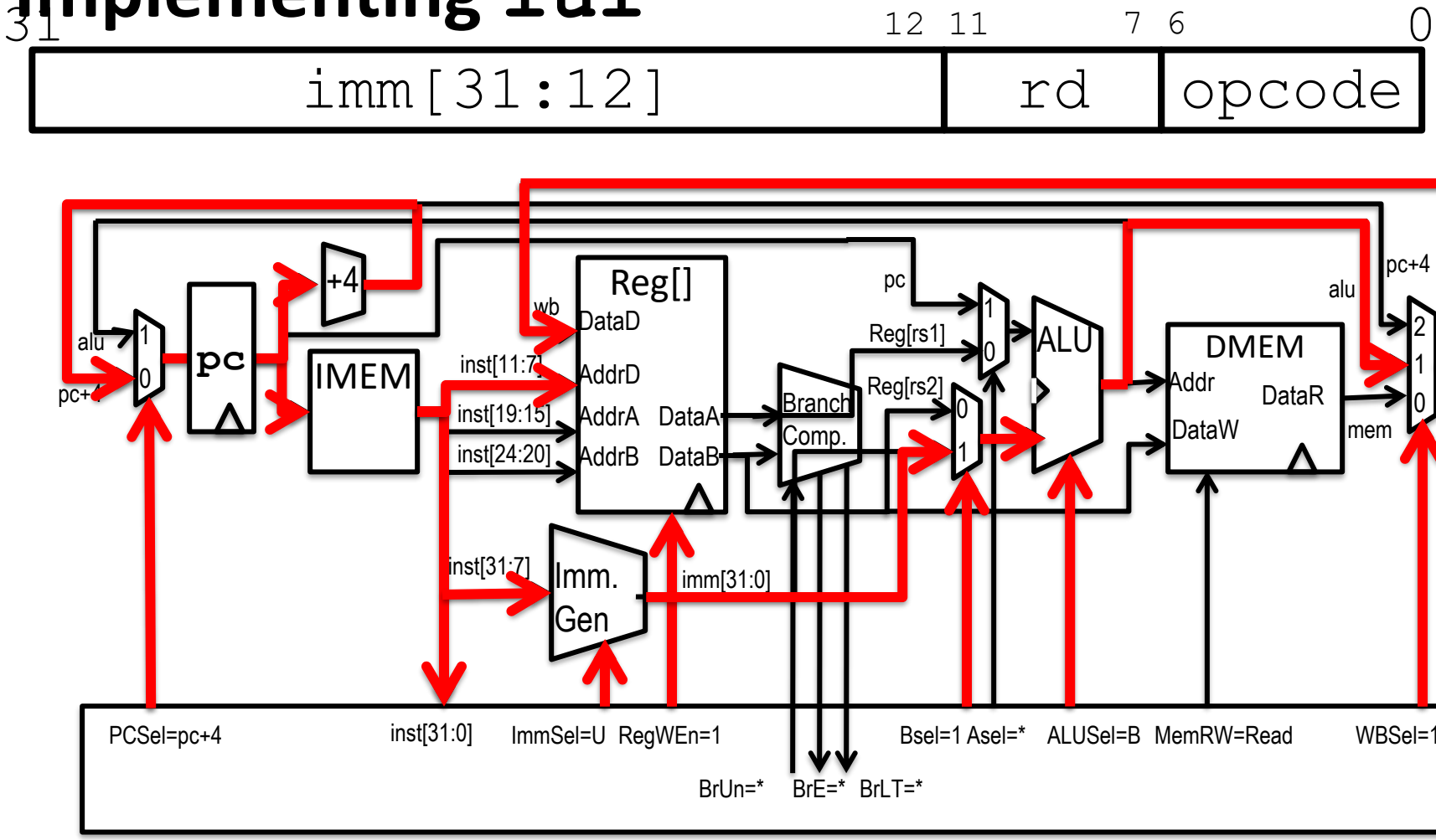
Adding `jal` to datapath





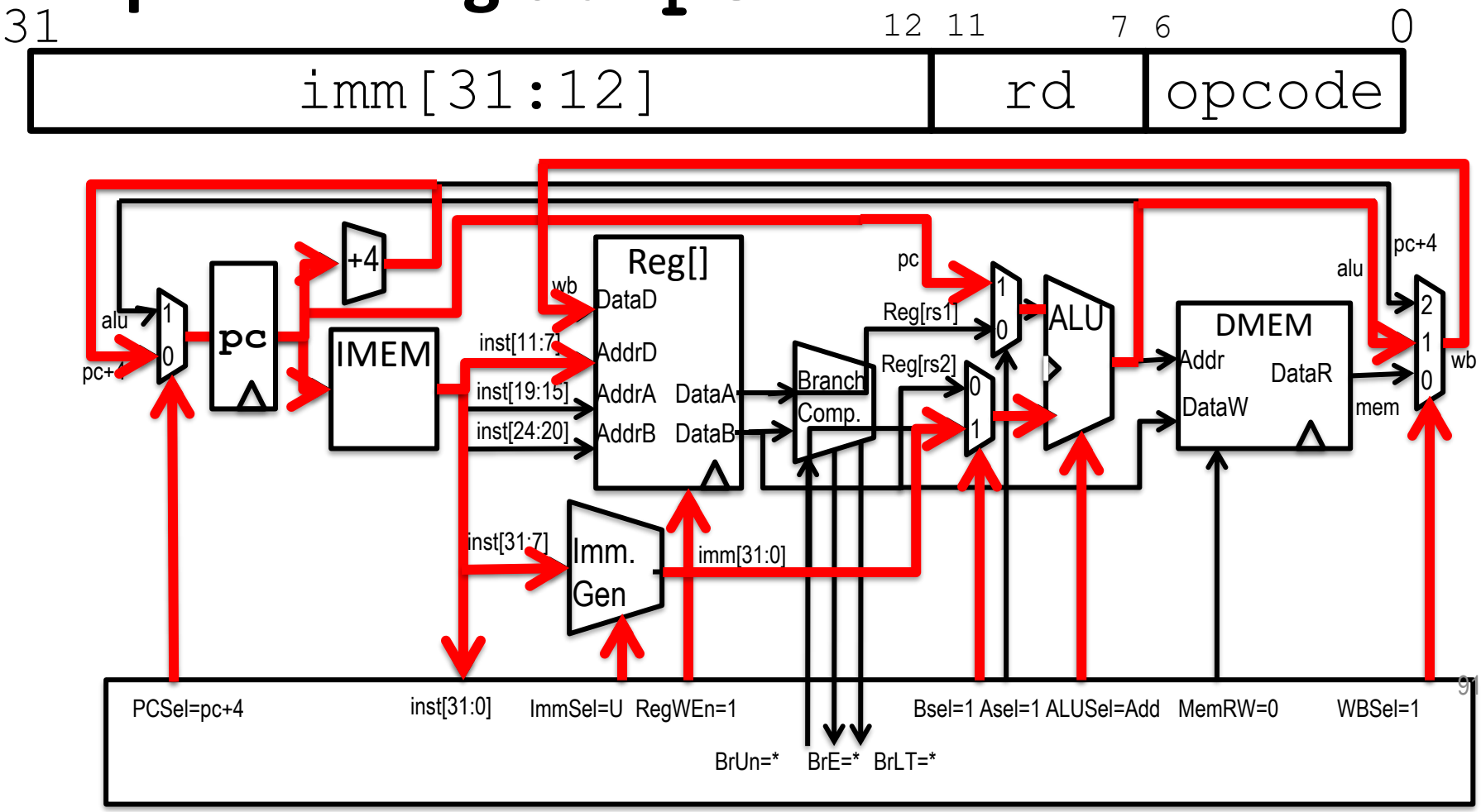
- `jal` saves PC+4 in register `rd` (the return address)
- Set PC = PC + offset (PC-relative jump)

Implementing `lui`



`lui` writes the upper 20 bits of the destination with the immediate value, and clears the lower 12 bits

Implementing `auipc`



Adds upper immediate value to PC and places result in destination register