

Combinational Logic (Cont.)

Sequential Logic

CPU Datapath

CMPT 295 Week 10.2

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 \leftrightarrow smaller transistor networks \leftrightarrow smaller circuit delays \leftrightarrow 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:

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

Example: Boolean Expression?

xy	0	1
0		1
1	1	1

$$x \neq y$$

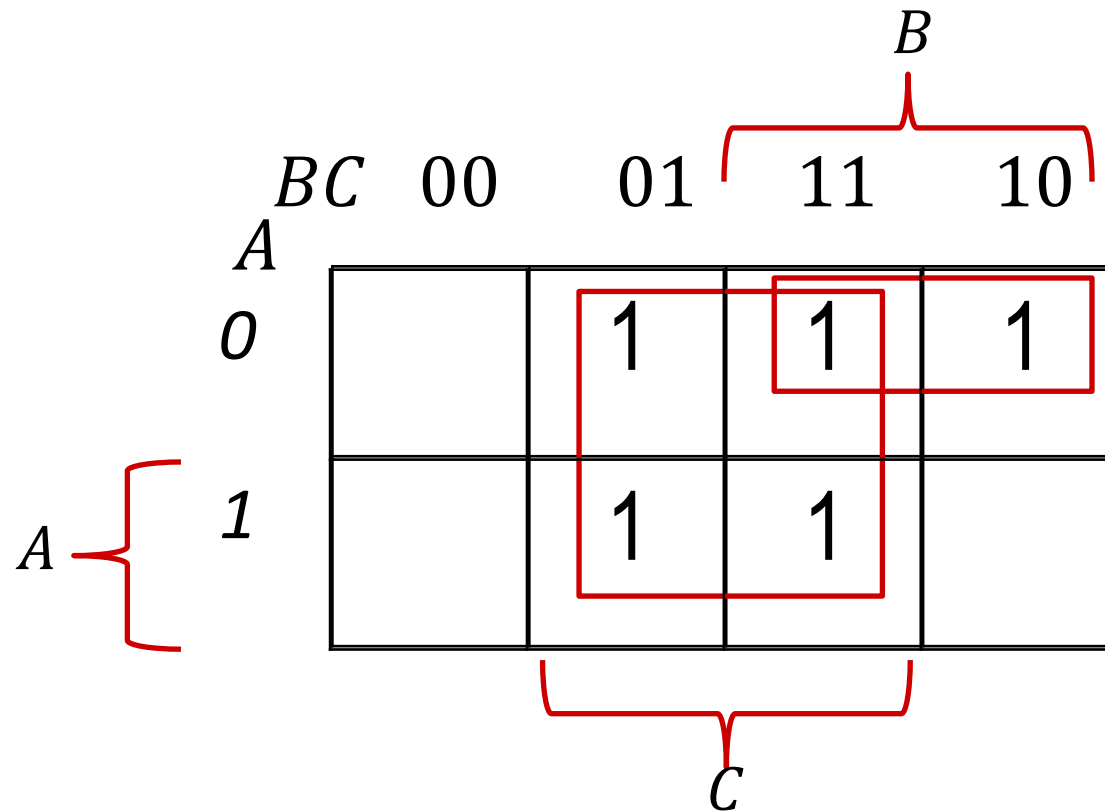
Three Variable Karnaugh Maps

		y	
		$\overbrace{11 \quad 10}^y$	
yz	00	01	
x			
0	$\bar{x}\bar{y}\bar{z}$	$\bar{x}\bar{y}z$	$\bar{x}y\bar{z}$
1	$x\bar{y}\bar{z}$	$x\bar{y}z$	xyz
		$\underbrace{\hspace{2cm}}_z$	

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: $C + \bar{A}B$

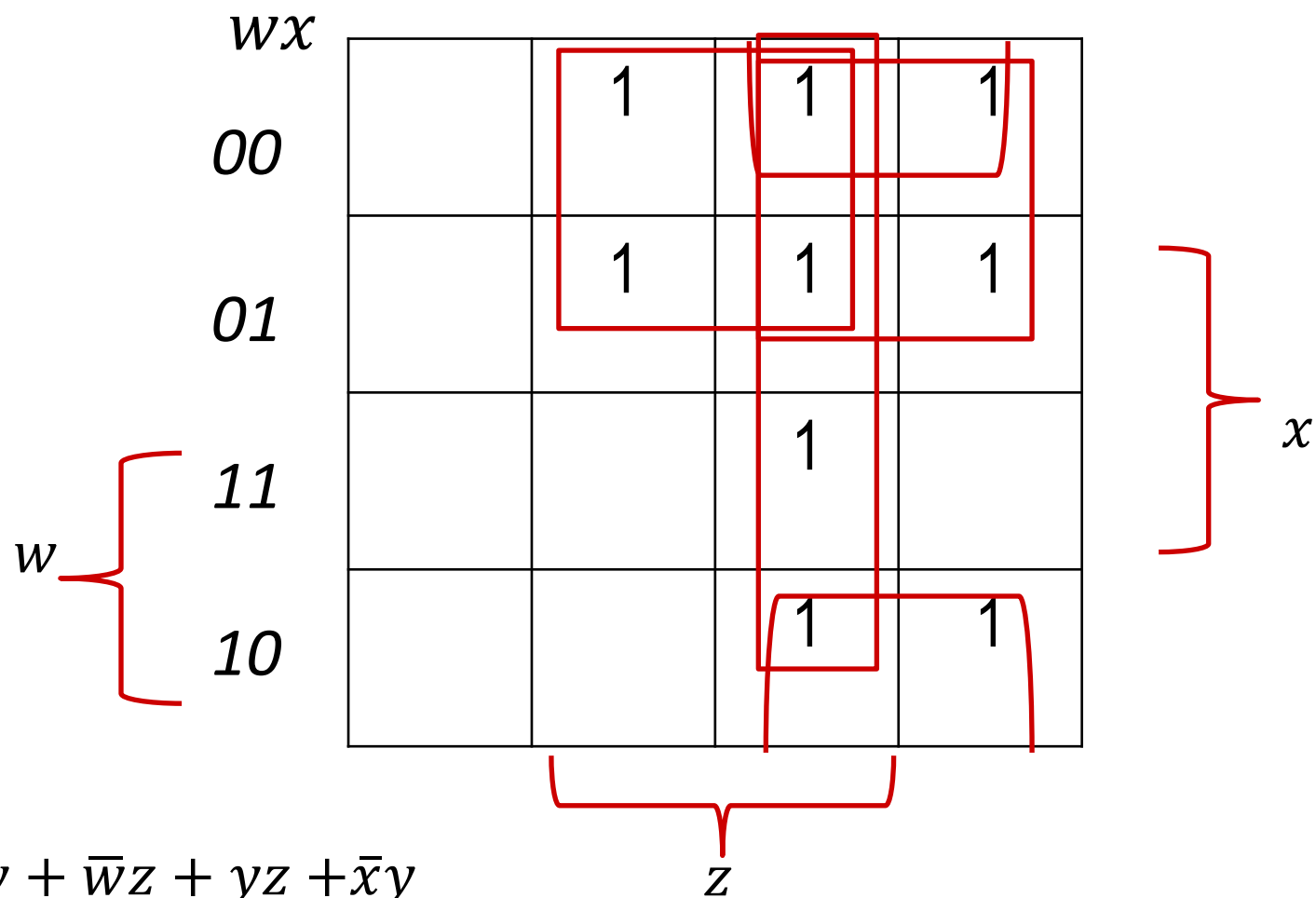
Four Variable Karnaugh Maps

		y			
		yz	00	01	$\overbrace{11 \quad 10}^y$
w	x				
w	00	$\bar{w}\bar{x}\bar{y}\bar{z}$	$\bar{w}\bar{x}\bar{y}z$	$\bar{w}\bar{x}yz$	$\bar{w}\bar{x}y\bar{z}$
	01	$\bar{w}x\bar{y}\bar{z}$	$\bar{w}x\bar{y}z$	$\bar{w}xyz$	$\bar{w}xy\bar{z}$
	11	$wx\bar{y}\bar{z}$	$wx\bar{y}z$	$wxyz$	$wxy\bar{z}$
	10	$w\bar{x}\bar{y}\bar{z}$	$w\bar{x}\bar{y}z$	$w\bar{x}yz$	$w\bar{x}y\bar{z}$
		z			

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

Example: Simplify 4-Variable Expression

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

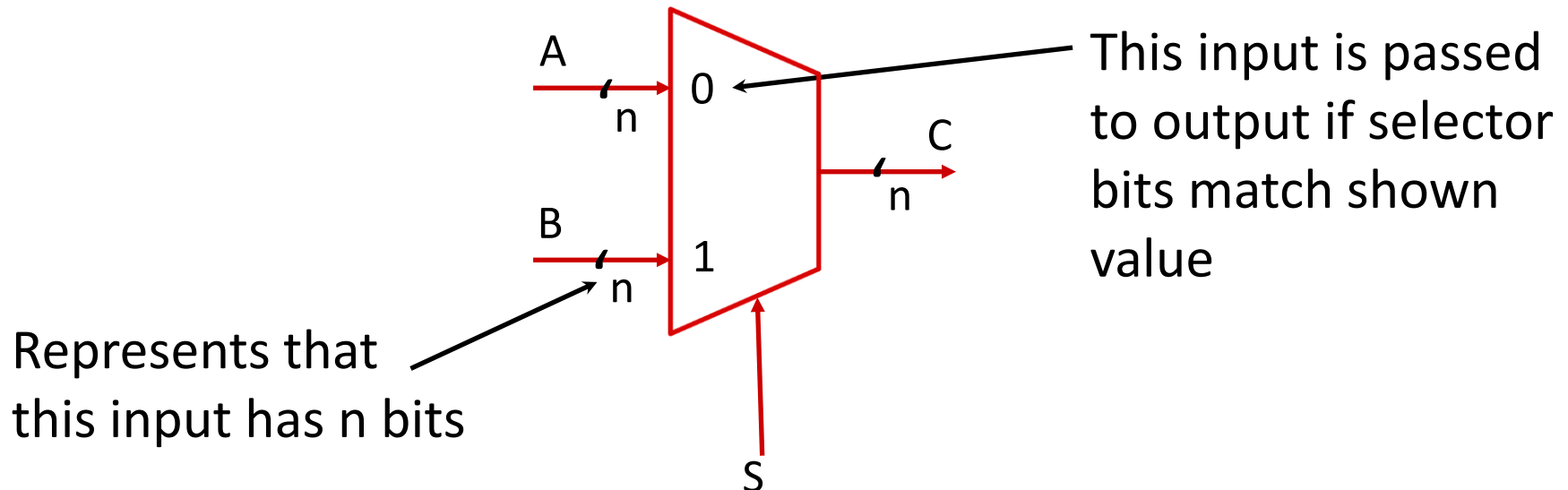


Solution: $\bar{w}y + \bar{w}z + yz + \bar{x}y$

Useful Combinational Circuits

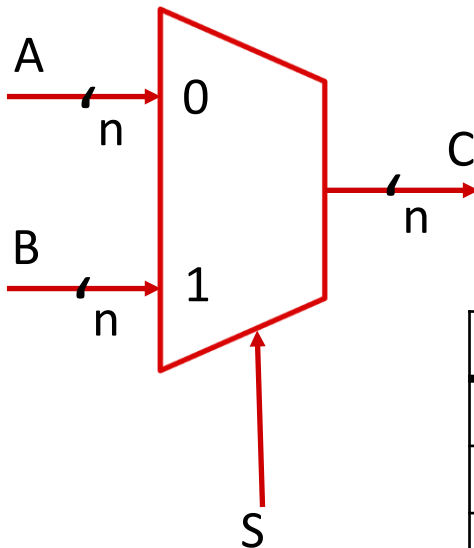
Data Multiplexor (MUX)

- 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



Implementing a 1-bit 2-to-1 MUX

- Schematic:**



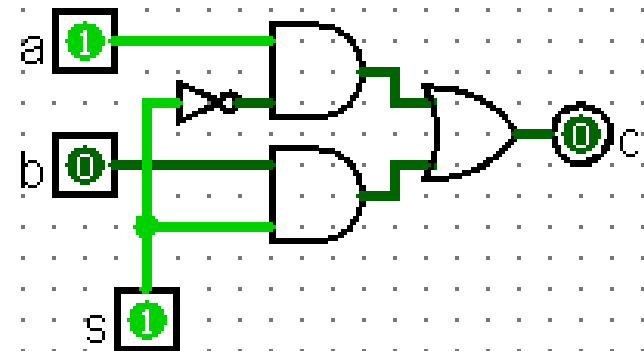
- Truth Table:**

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

- Boolean Algebra:**

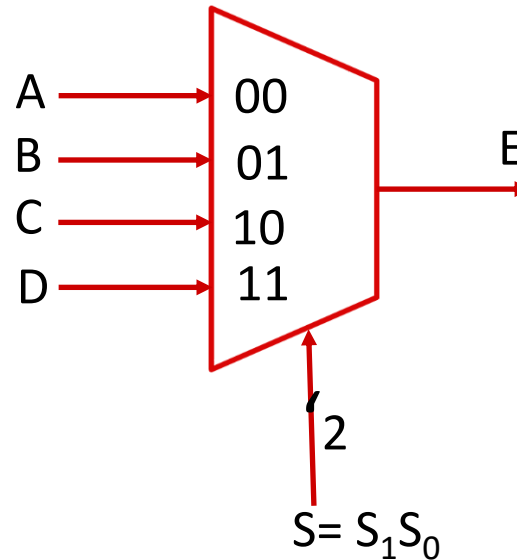
$$\begin{aligned}
 c &= \bar{s}a\bar{b} + \bar{s}ab + s\bar{a}b + sab \\
 &= \bar{s}(a\bar{b} + ab) + s(\bar{a}b + ab) \\
 &= \bar{s}(a(\bar{b} + b)) + s((\bar{a} + a)b) \\
 &= \bar{s}(a(1) + s((1)b) \\
 &= \bar{s}a + sb
 \end{aligned}$$

- Circuit Diagram:**



1-bit 4-to-1 MUX

- Schematic:**



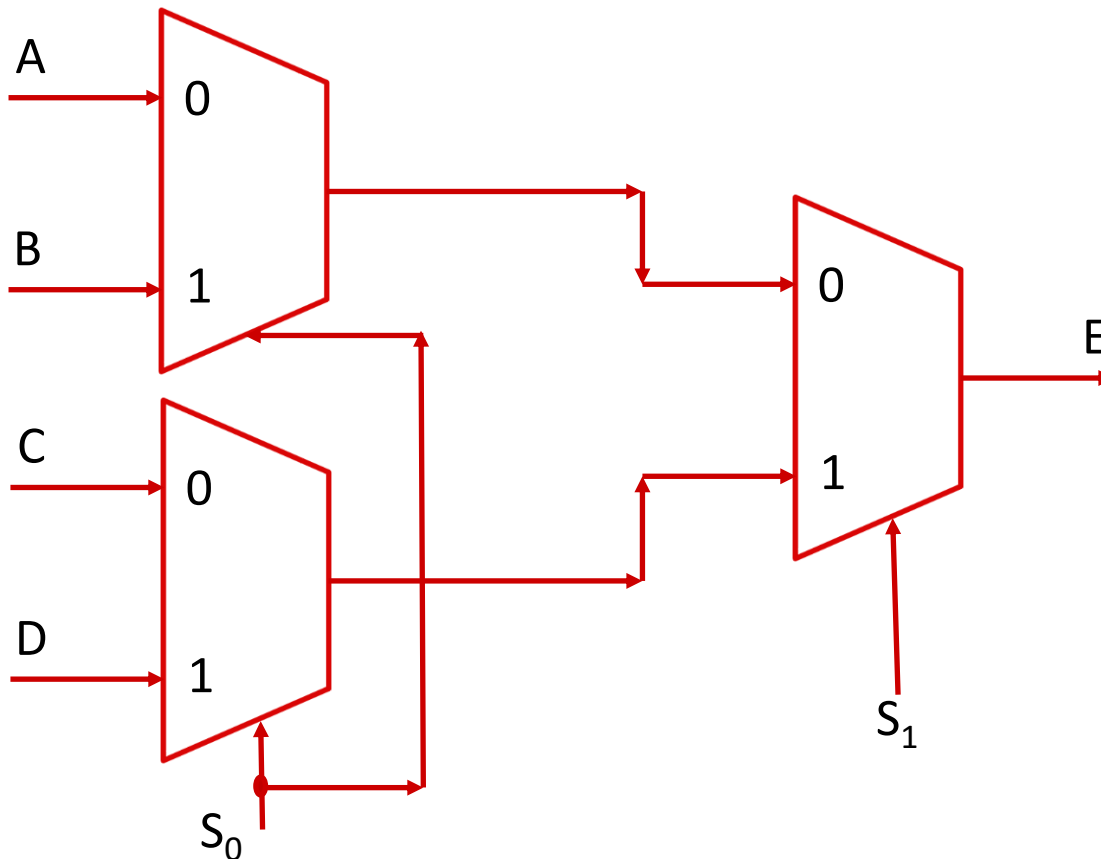
- Truth Table:** How many rows? 2^6

- Boolean Expression:**

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

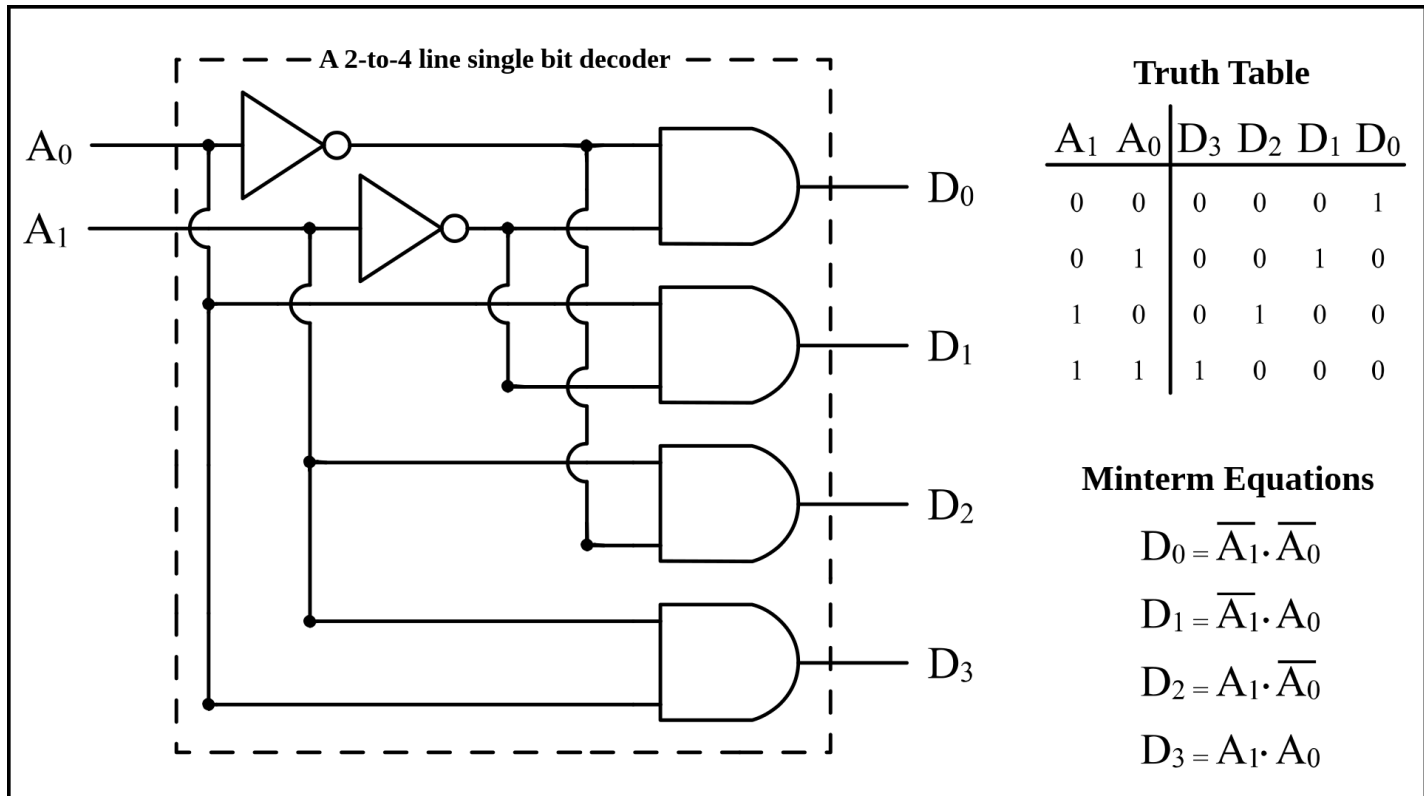
Another Design for 4-to-1 MUX

- 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

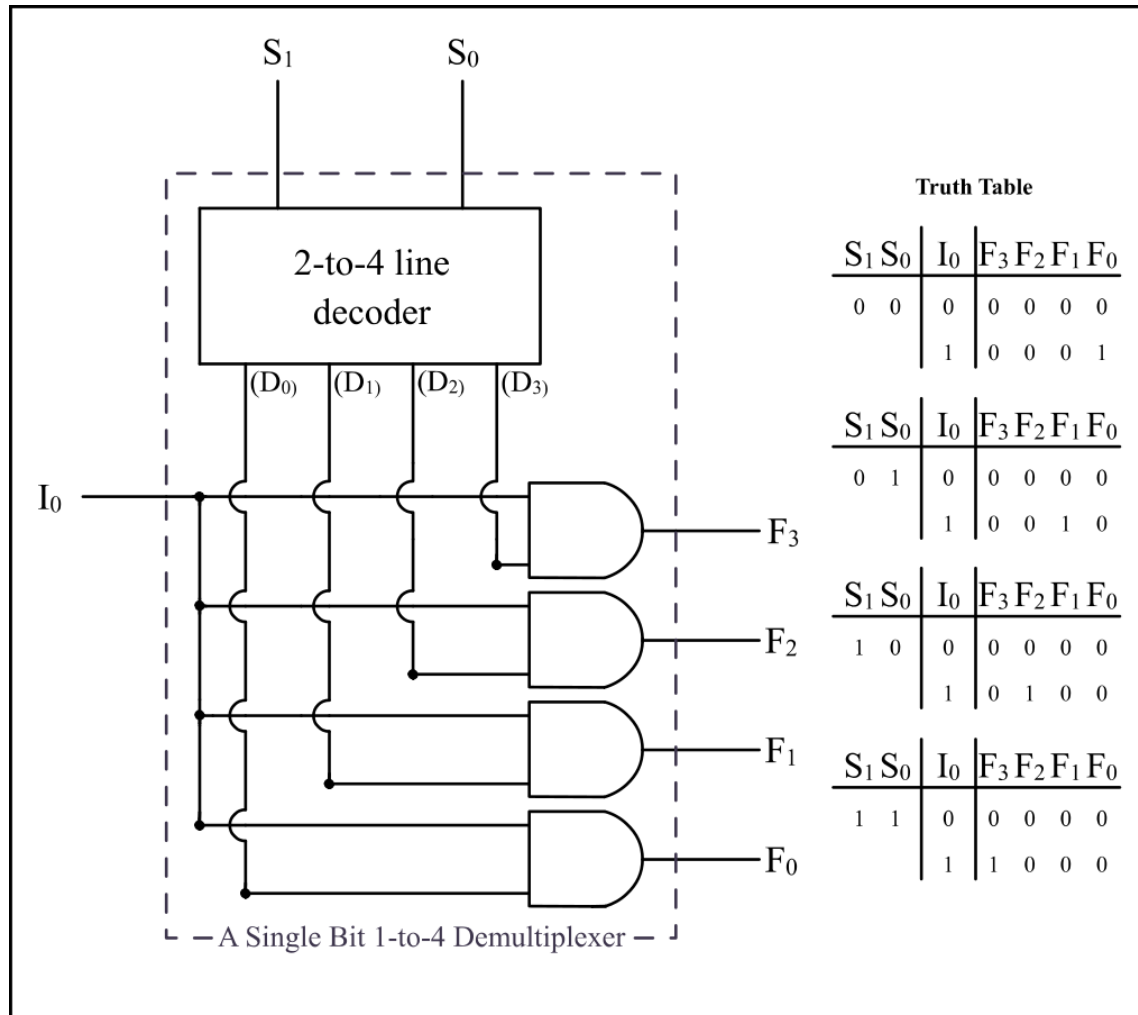


By BlueJester0101, CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=3668293>

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

Demultiplexer (Demux)

- Similar to decoder with an enable signal

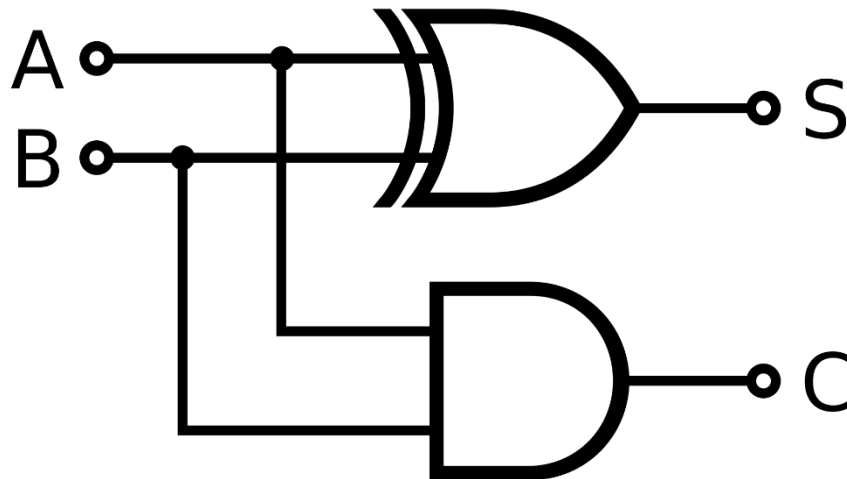


By BlueJester0101, CC BY-SA 3.0,
<https://commons.wikimedia.org/w/index.php?curid=3668293>

Single-Bit Binary Adder (Half Adder)

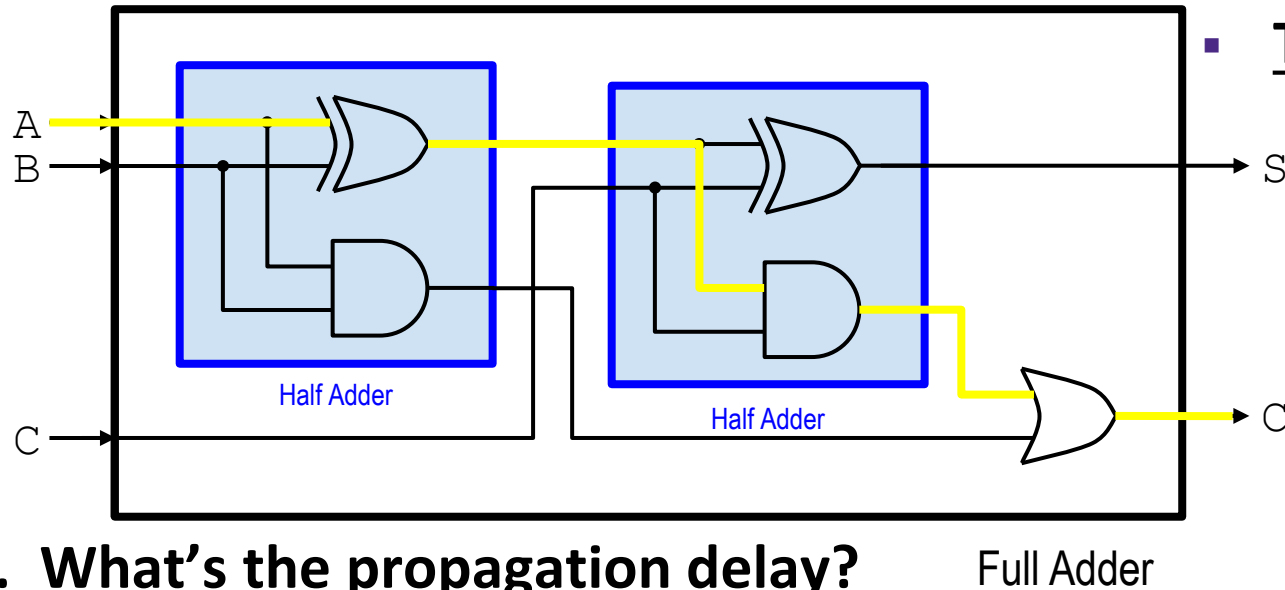
- Add $A + B$ to get Sum (S) and Carry (C)
- Truth Table:
- Boolean Expressions:
 - $S = A \oplus B$; $C = AB$
- Circuit:

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>

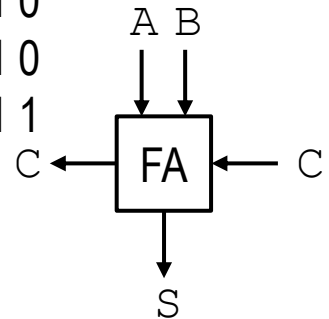
What is this Circuit?



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
<hr/>				
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?

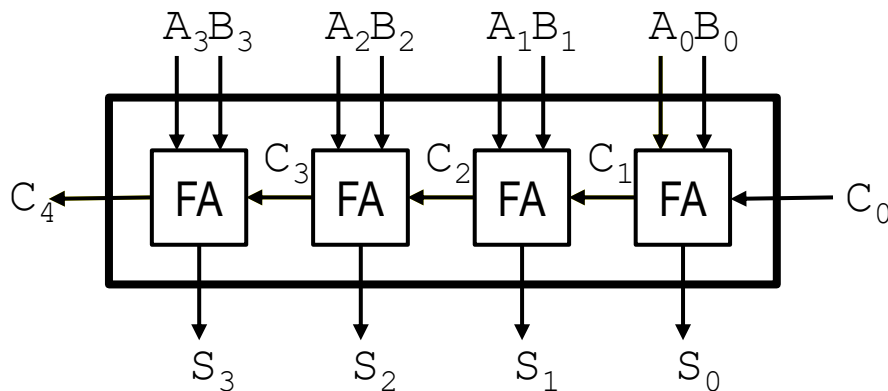
- Algebra:

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

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

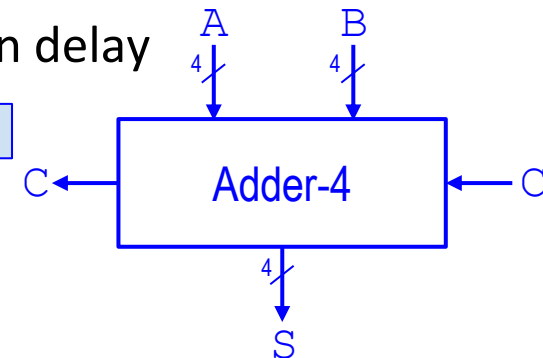


... a 4-bit ripple carry adder!

- Adds by columns
- Propagation delay

= 9

$$(2n + 1)$$

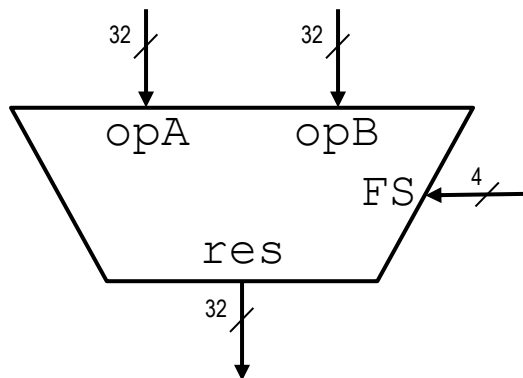


Functional Unit

Hardware circuits are fixed

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

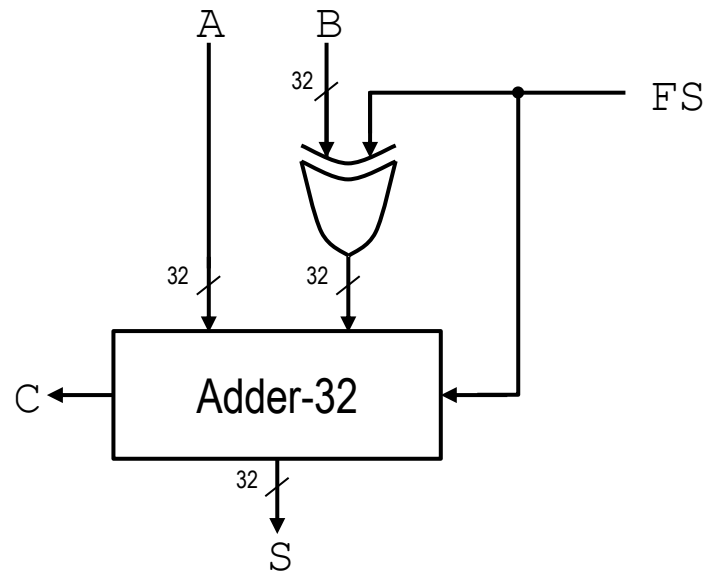
Function Unit:



Function Select:

FS	func
0001	$A + B$
0010	$A - B$
1000	$A * B$
0100	$A \wedge B$
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

Accumulator Example

An example of why we would need sequential logic

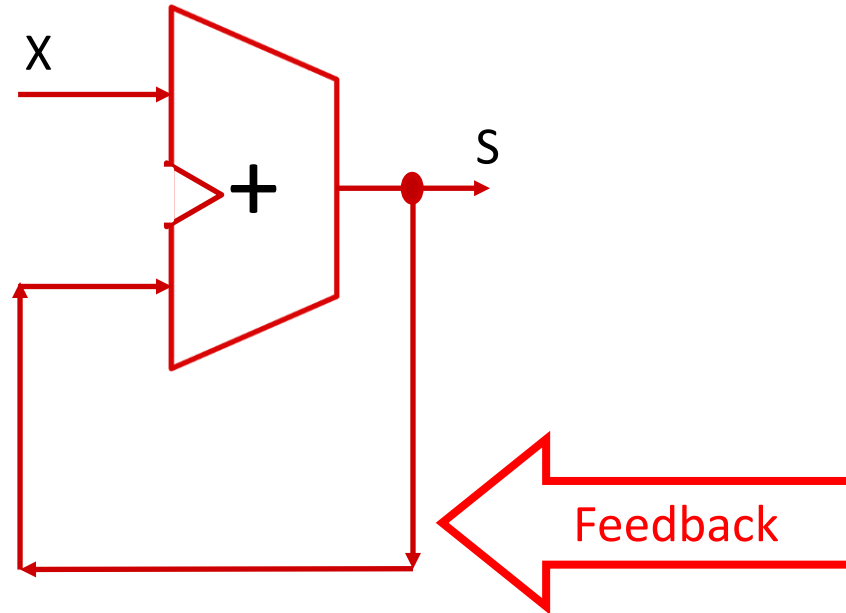


Want: $S=0;$
for X_1, X_2, X_3 over time...
 $S = S + X_i$

Assume:

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

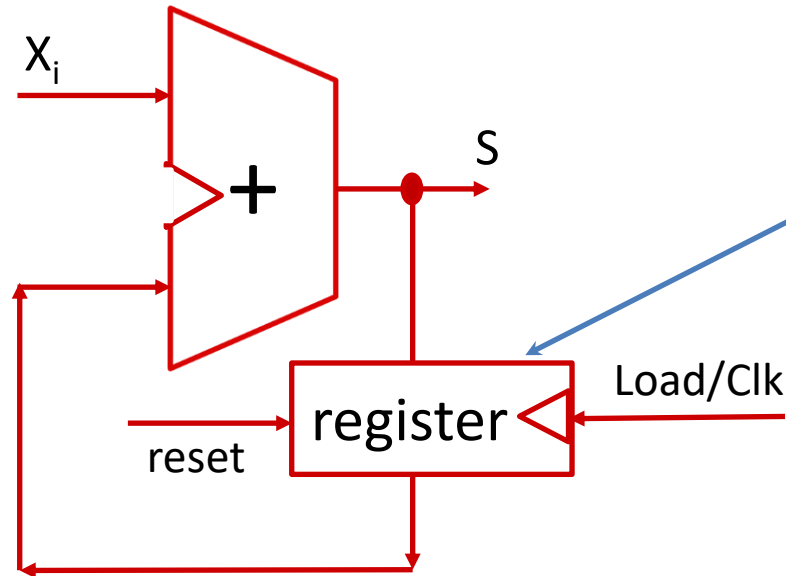
First Try: Does this work?



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

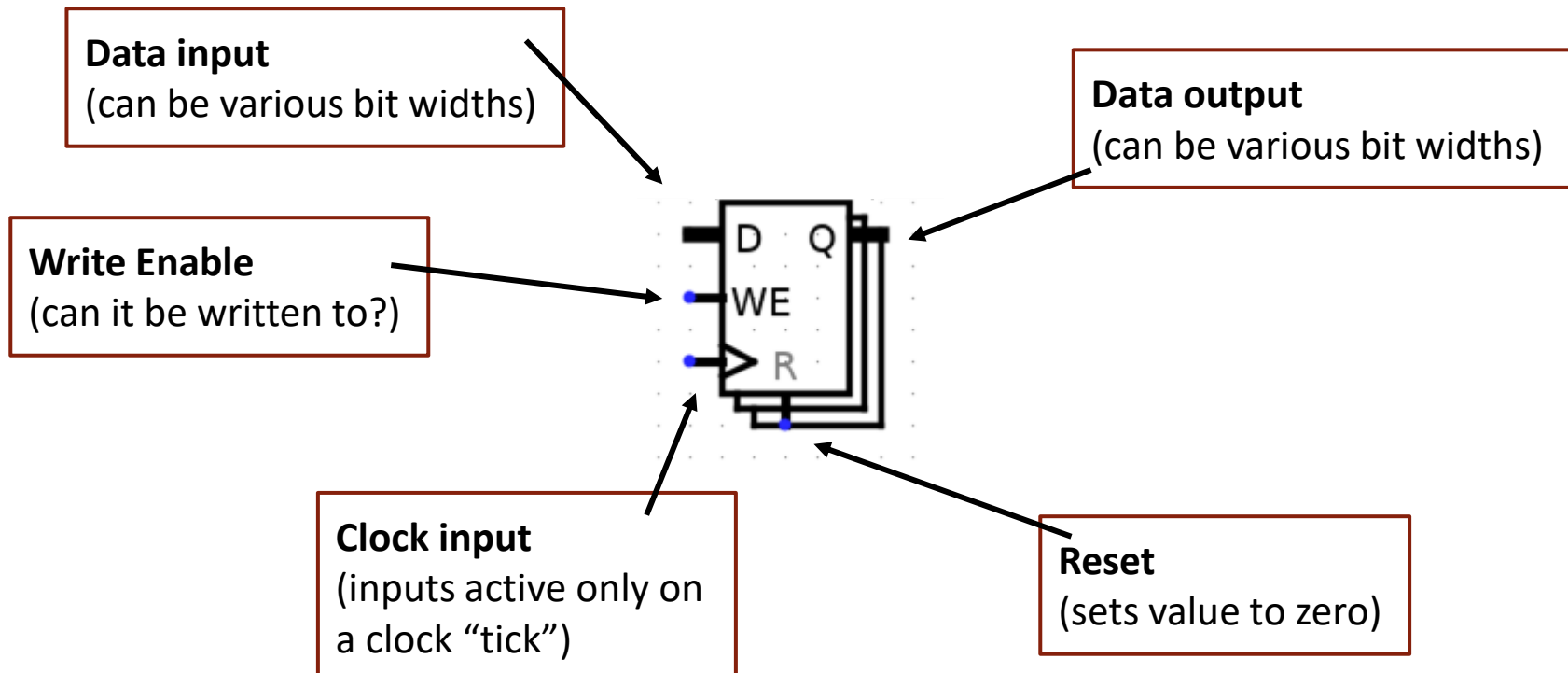
Uses for State Elements

- Place to store values for some amount of time:
 - Register files (like in RISC-V)
 - 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



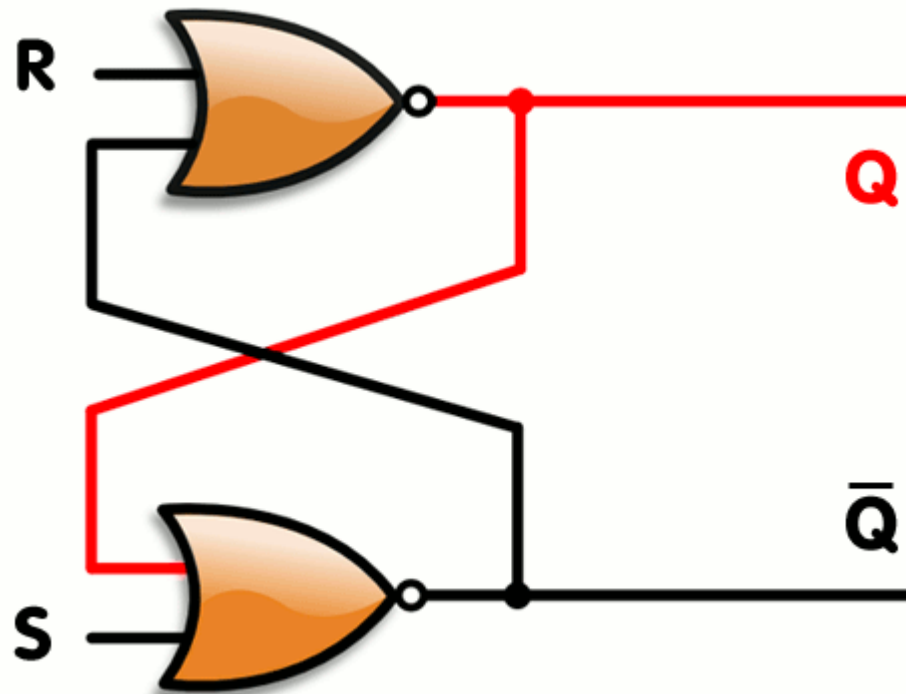
First State Element: RS Latch

When $R = 1$ and $S = 0 \rightarrow Q$ is 0

When $S = 1$ and $R = 0 \rightarrow Q$ is 1

When both S and R are 0 $\rightarrow Q$ stays the same

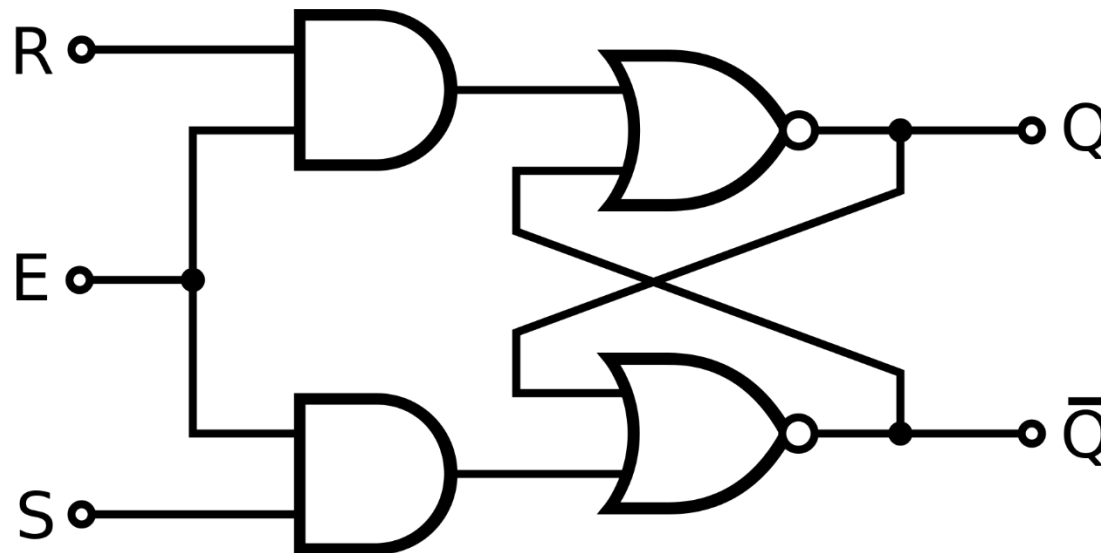
When both S and R are 1 \rightarrow Undefined



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

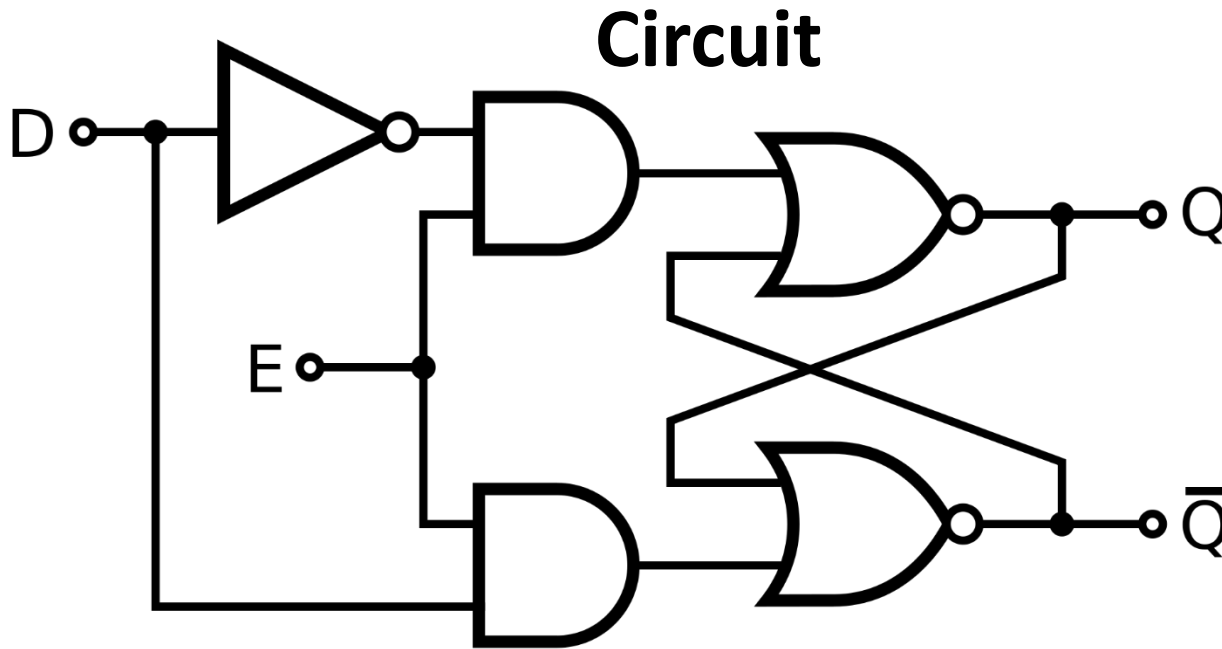
- ❖ Only changes state when $E = 1$.
- ❖ Stays the same when $E = 0$



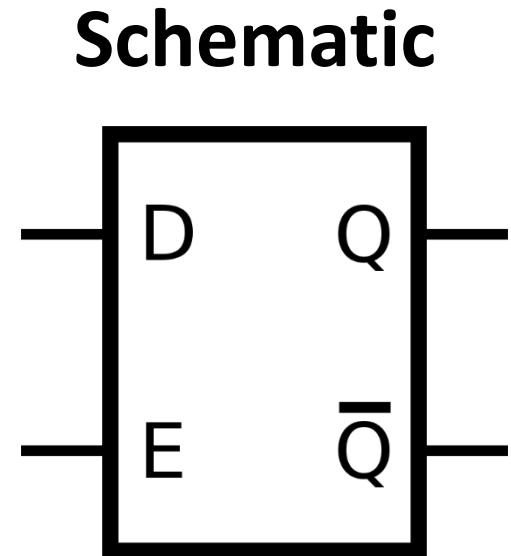
By Inductiveload - Own Drawing in Inkscape 0.43, Public Domain,
<https://commons.wikimedia.org/w/index.php?curid=873598>

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 $E = 0$



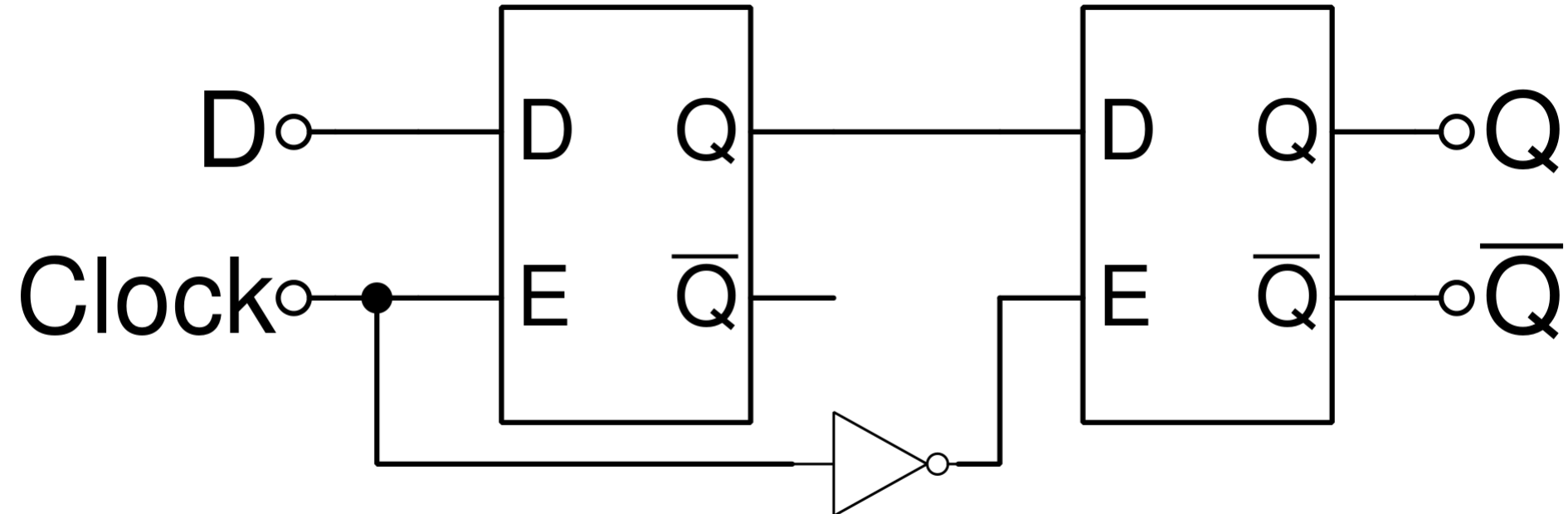
By Inductiveload - Own work, Public Domain,
<https://commons.wikimedia.org/w/index.php?curid=6712572>



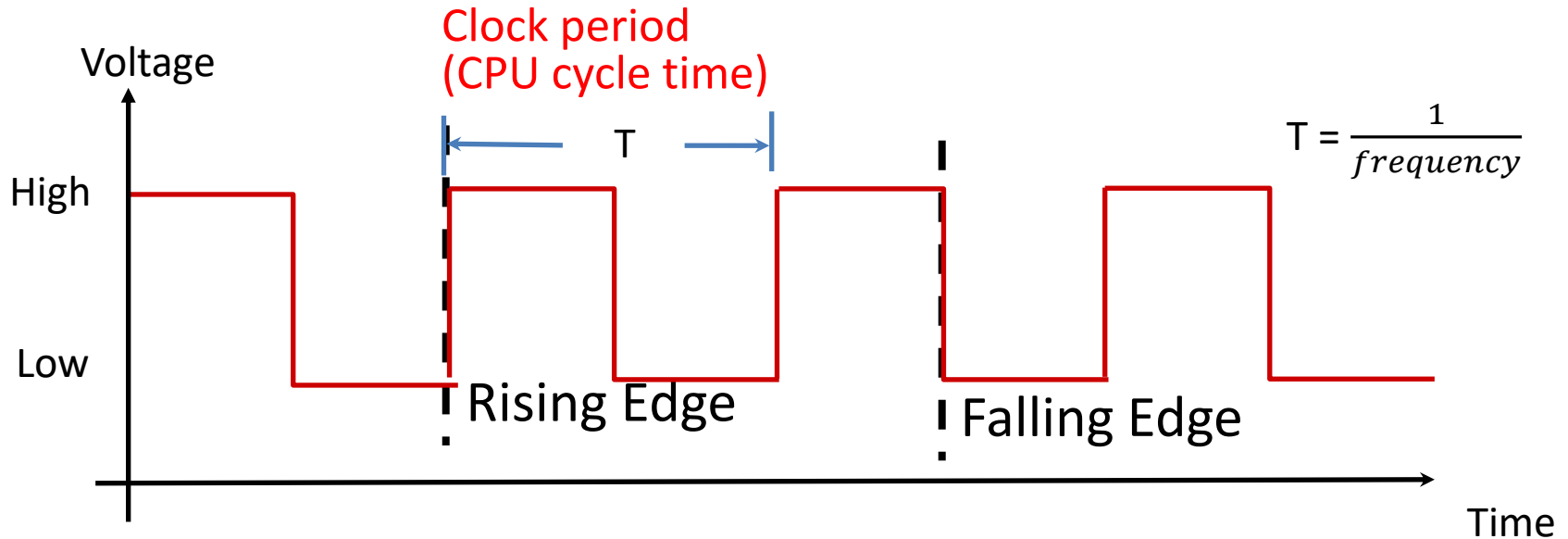
By Inductiveload - Own work, Public Domain,
<https://commons.wikimedia.org/w/index.php?curid=6712594>

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

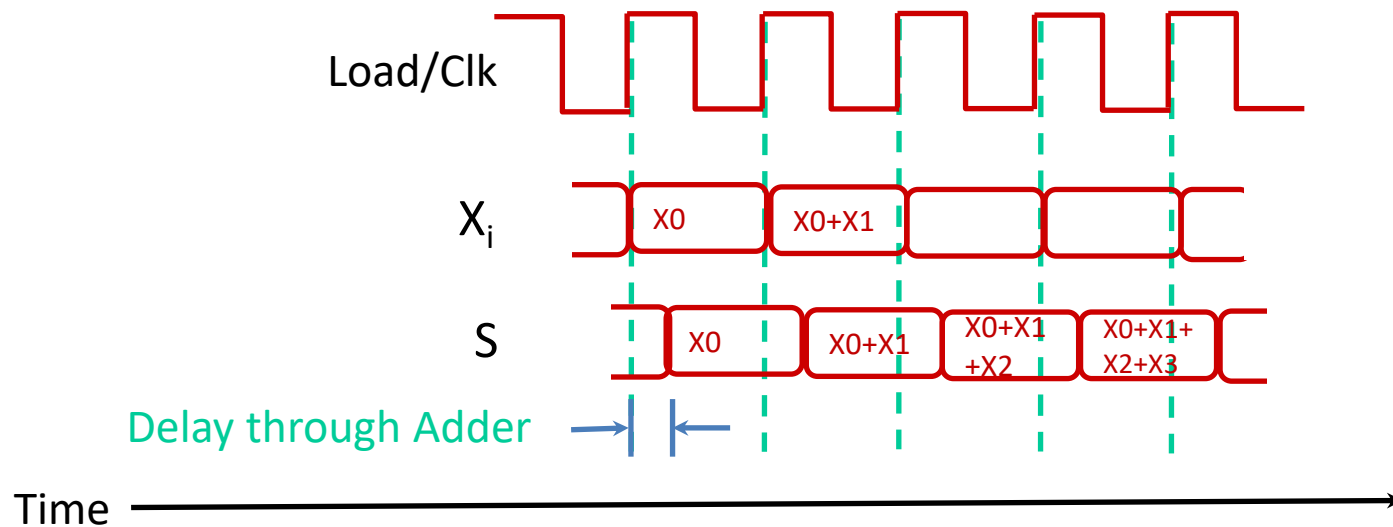
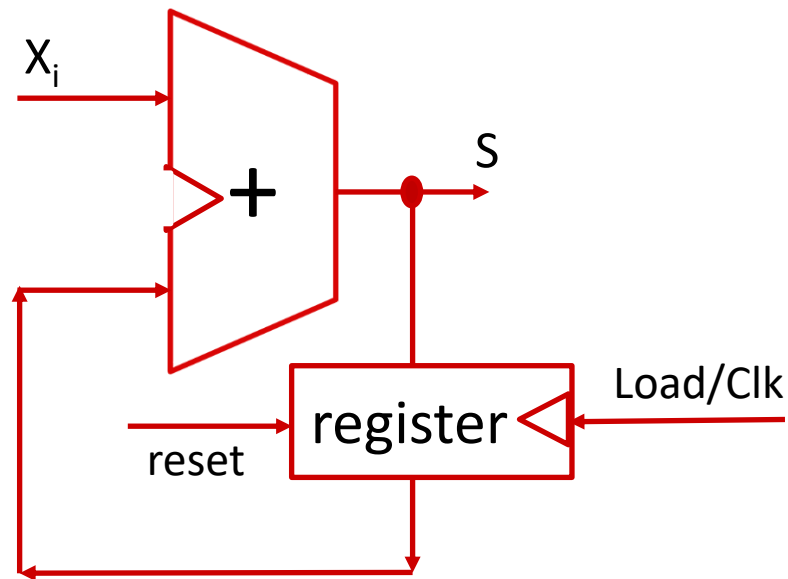


- **Signals** transmitted over wires continuously
- Transmission is effectively instantaneous
 - Implies that any wire only contains one value at any given time

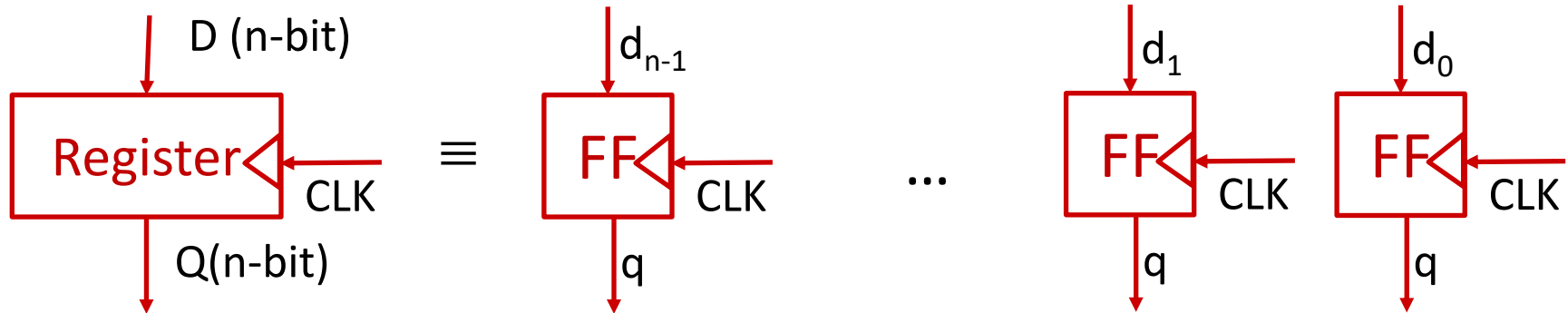
Dealing with Waveform Diagrams

- 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

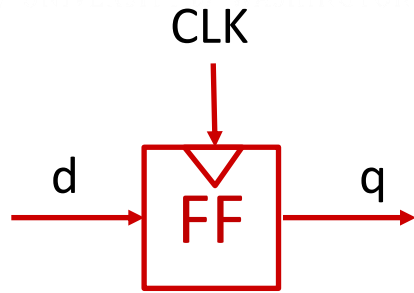
Accumulator 2nd Try: How About This?



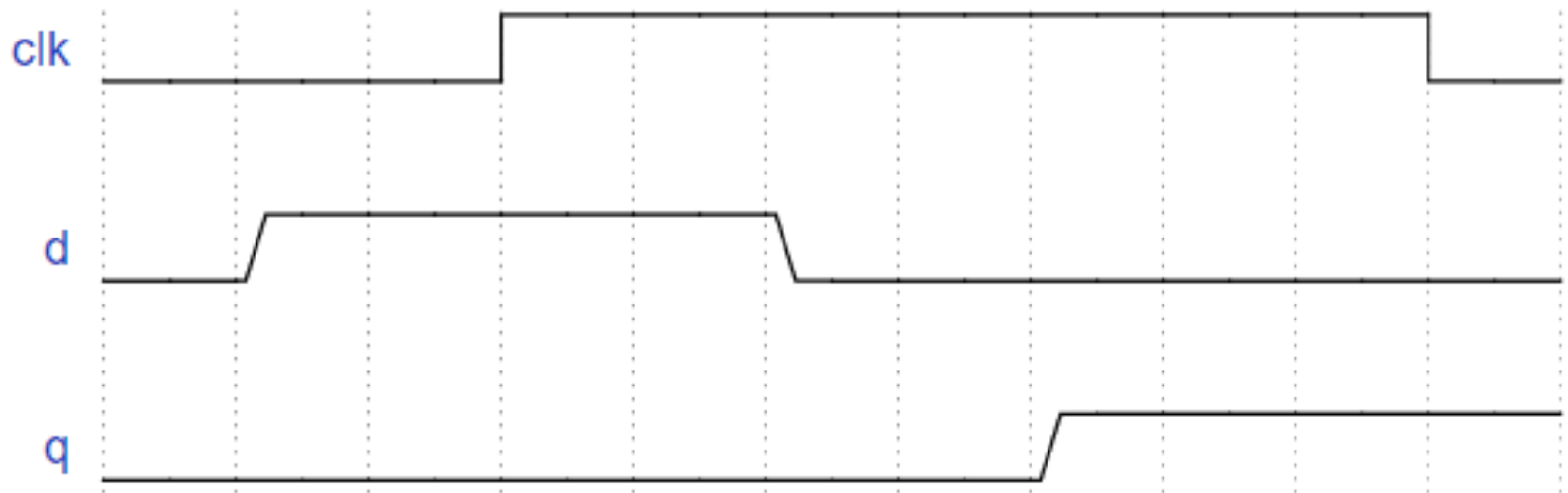
Register Internals

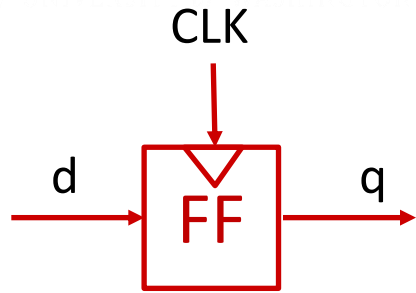


- N-bit register \equiv n instances of a *“Flip-Flop”*
 - Output flips and flops between 0 and 1
- Specifically this is a “D-type Flip-Flop”
 - D is “data input”, Q is “data output”
 - A group of wires when interpreted as a bit field is called a *bus*

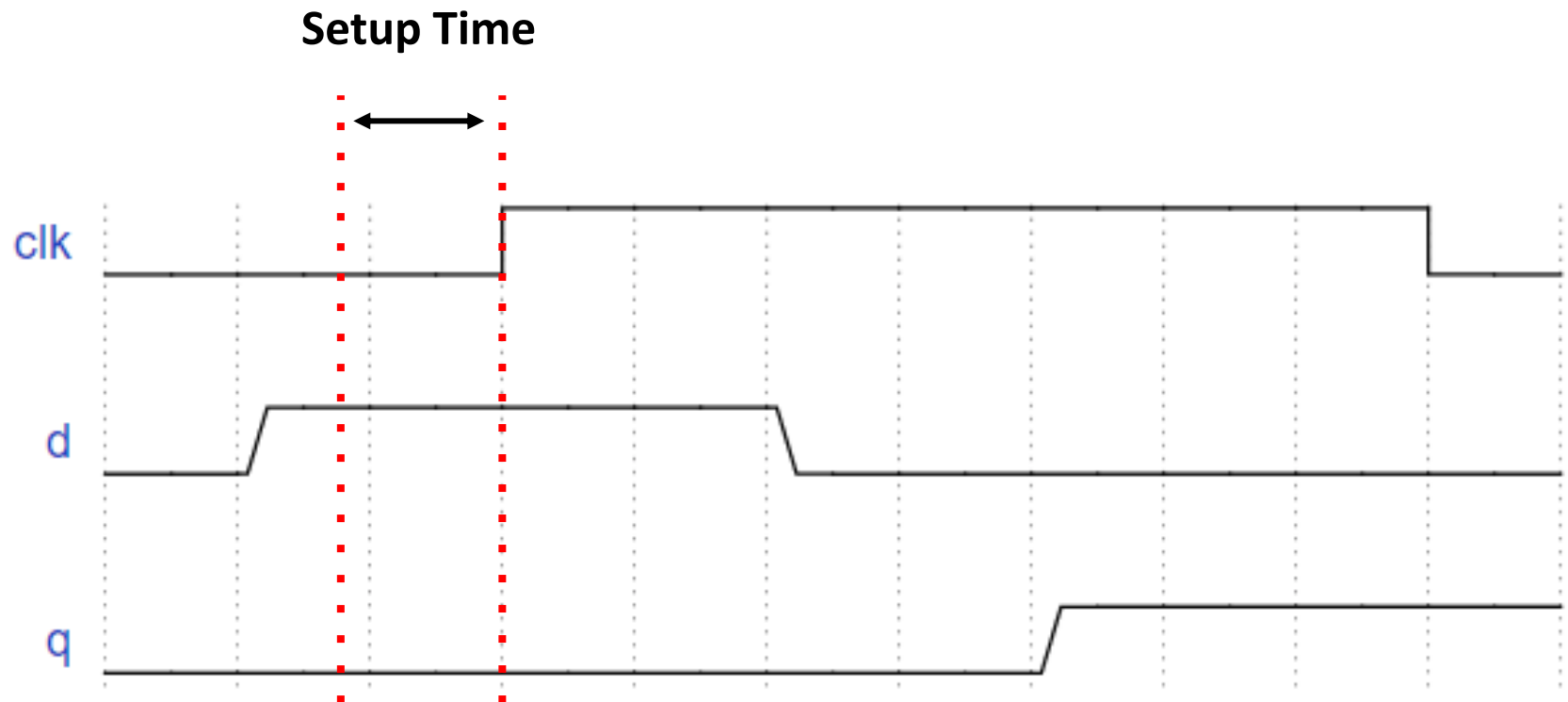


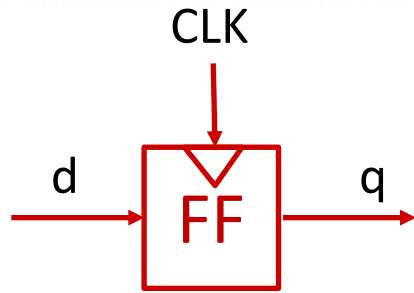
Flip-Flop Timing Behavior



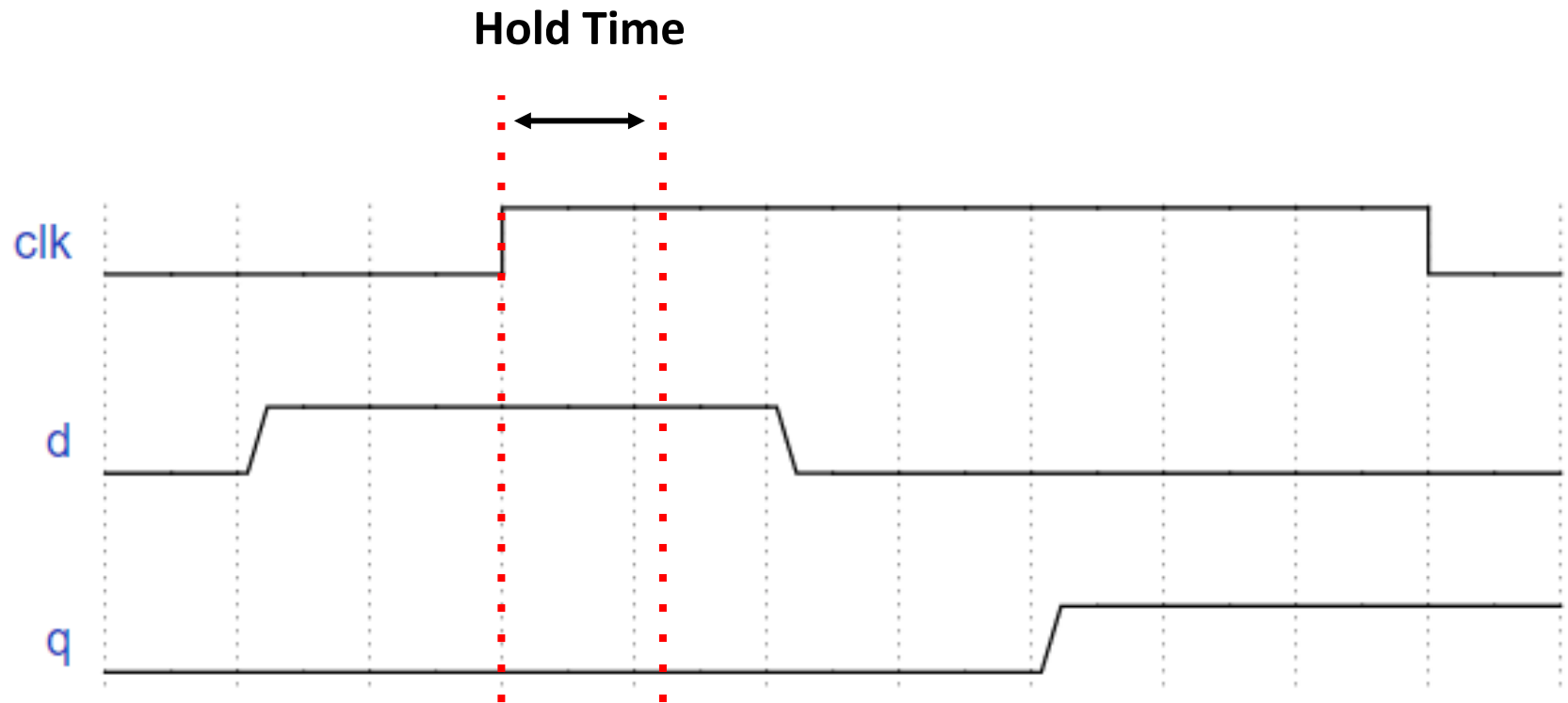


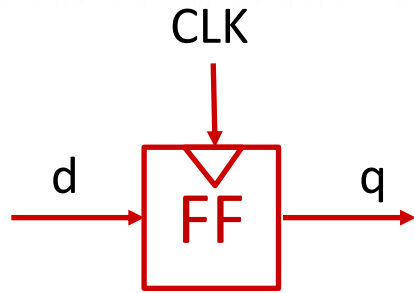
Flip-Flop Timing Behavior



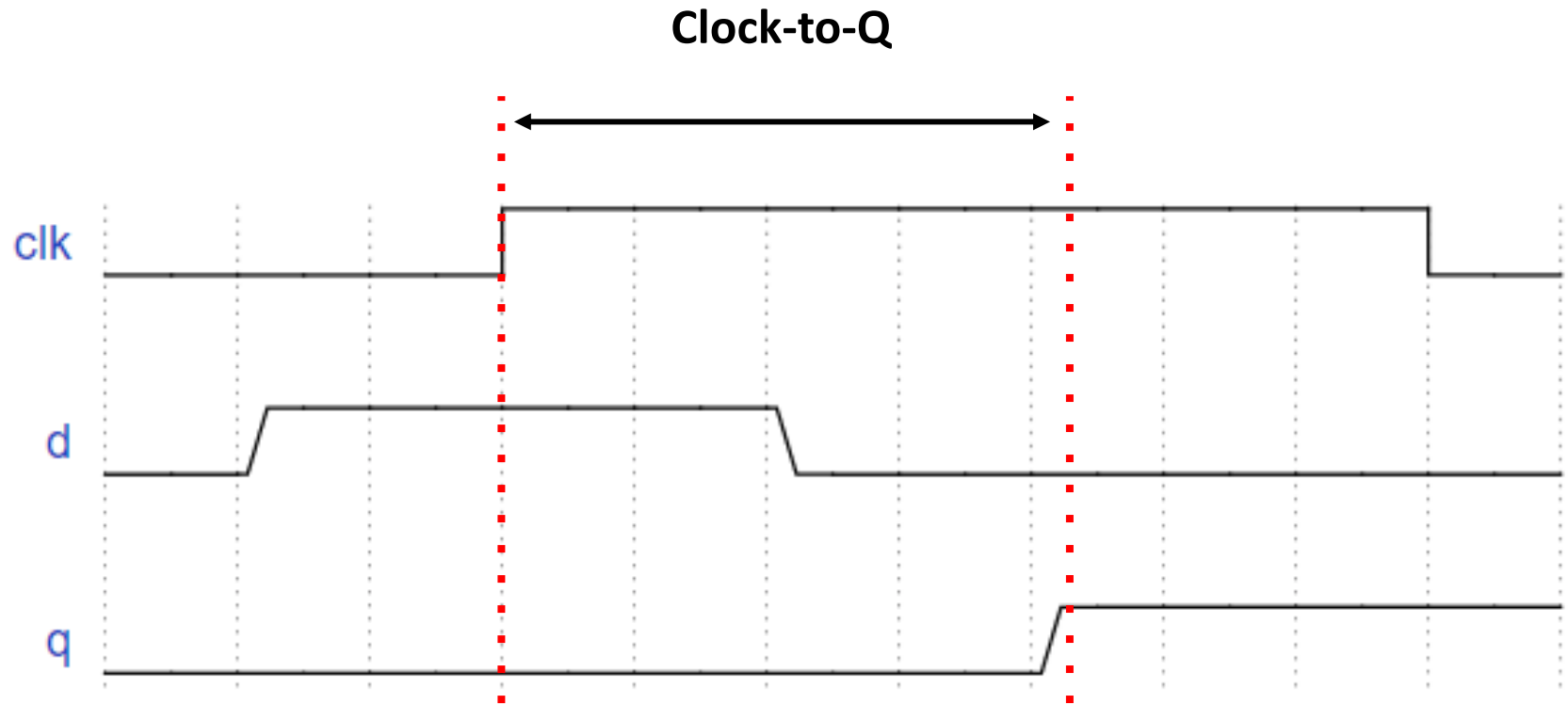


Flip-Flop Timing Behavior

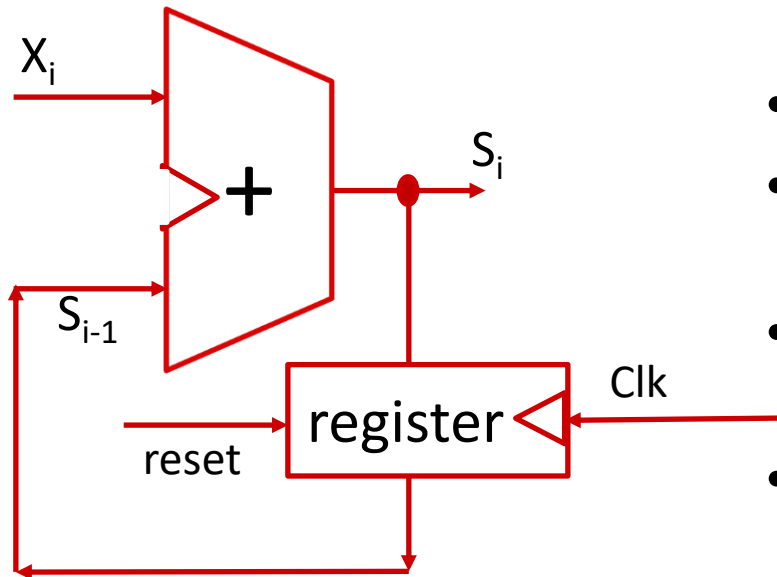




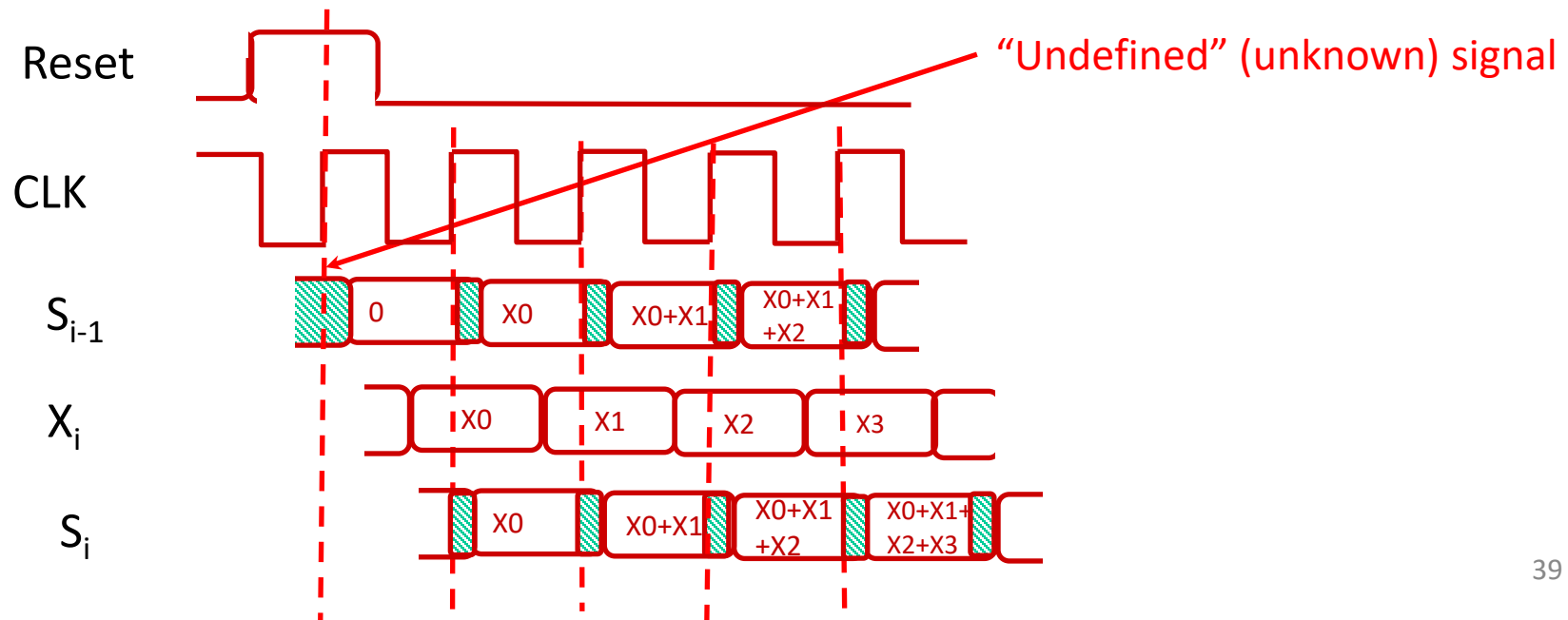
Flip-Flop Timing Behavior



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

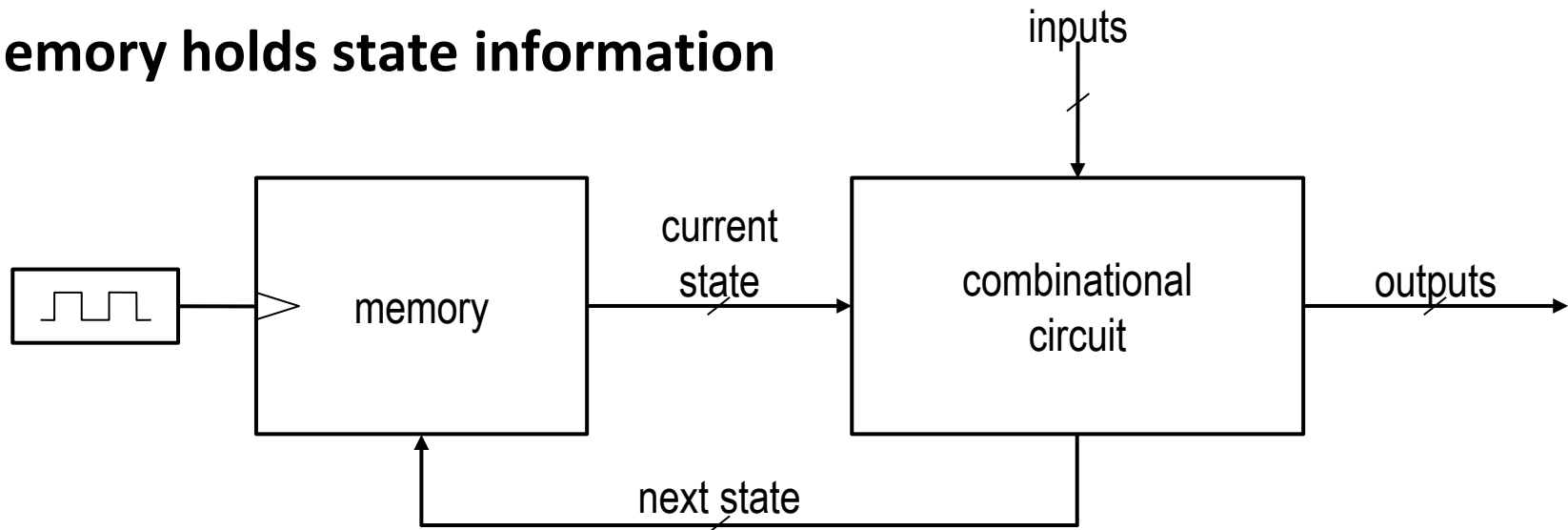


Timing Terms

- **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



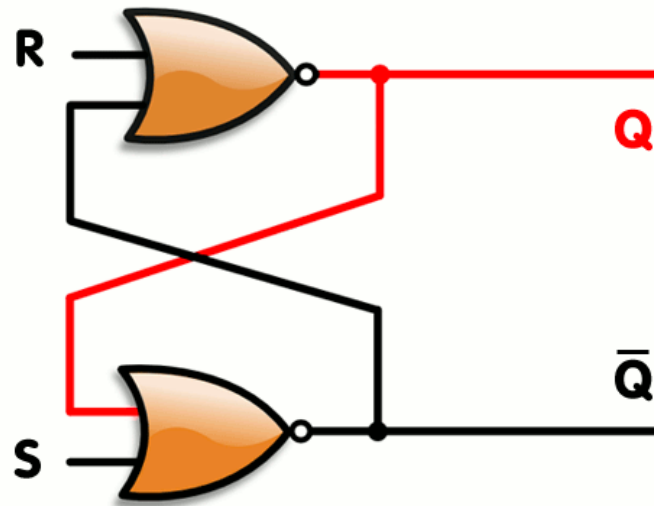
- compute next state based on (current state, inputs)
- compute outputs based on (current state, inputs)

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



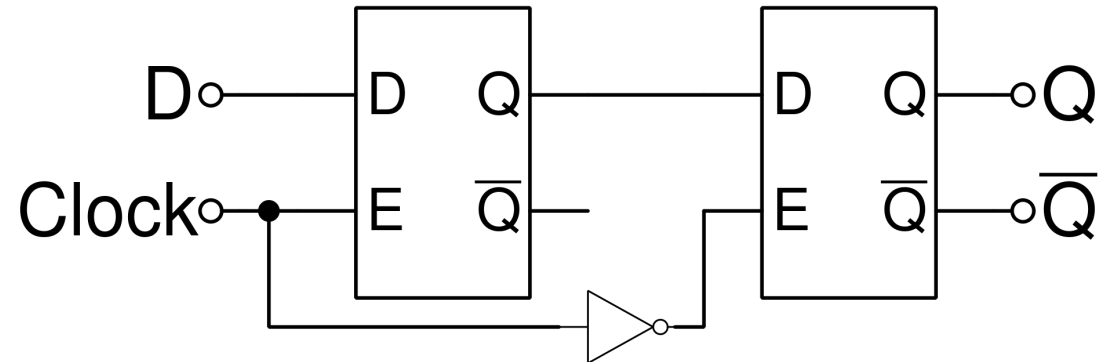
By Napalm Llama - Modification of Wikimedia Commons file R-S.gif (shown below), CC BY 2.0, <https://commons.wikimedia.org/wiki/index.php?curid=4845402>

Q
 \bar{Q}
 R
 S



Time ⁴²

Waveform Example: D Flip-Flop

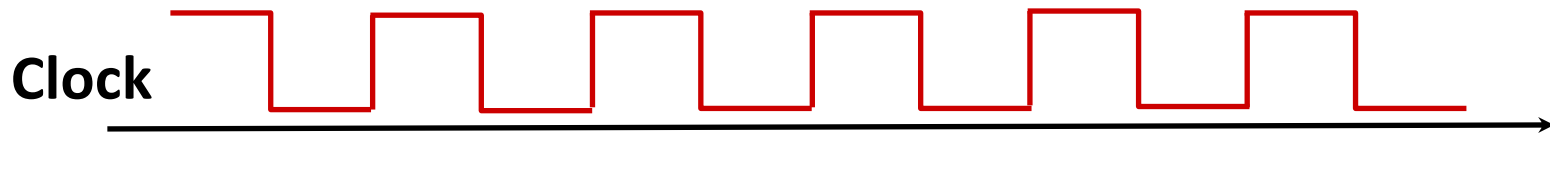


By Nolanjshettle at English Wikipedia, CC BY-SA 3.0,
<https://commons.wikimedia.org/w/index.php?curid=40852395>

Q

\overline{Q}

D



CPU Hardware

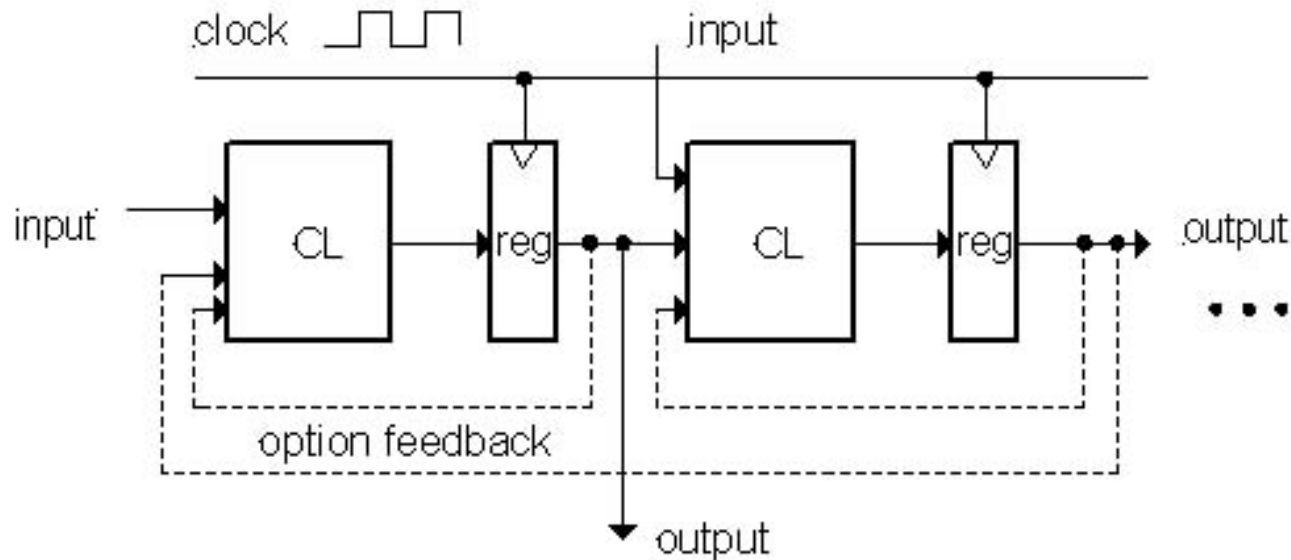
Goal: Given an instruction set architecture, construct a machine that reliably executes instructions.

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”

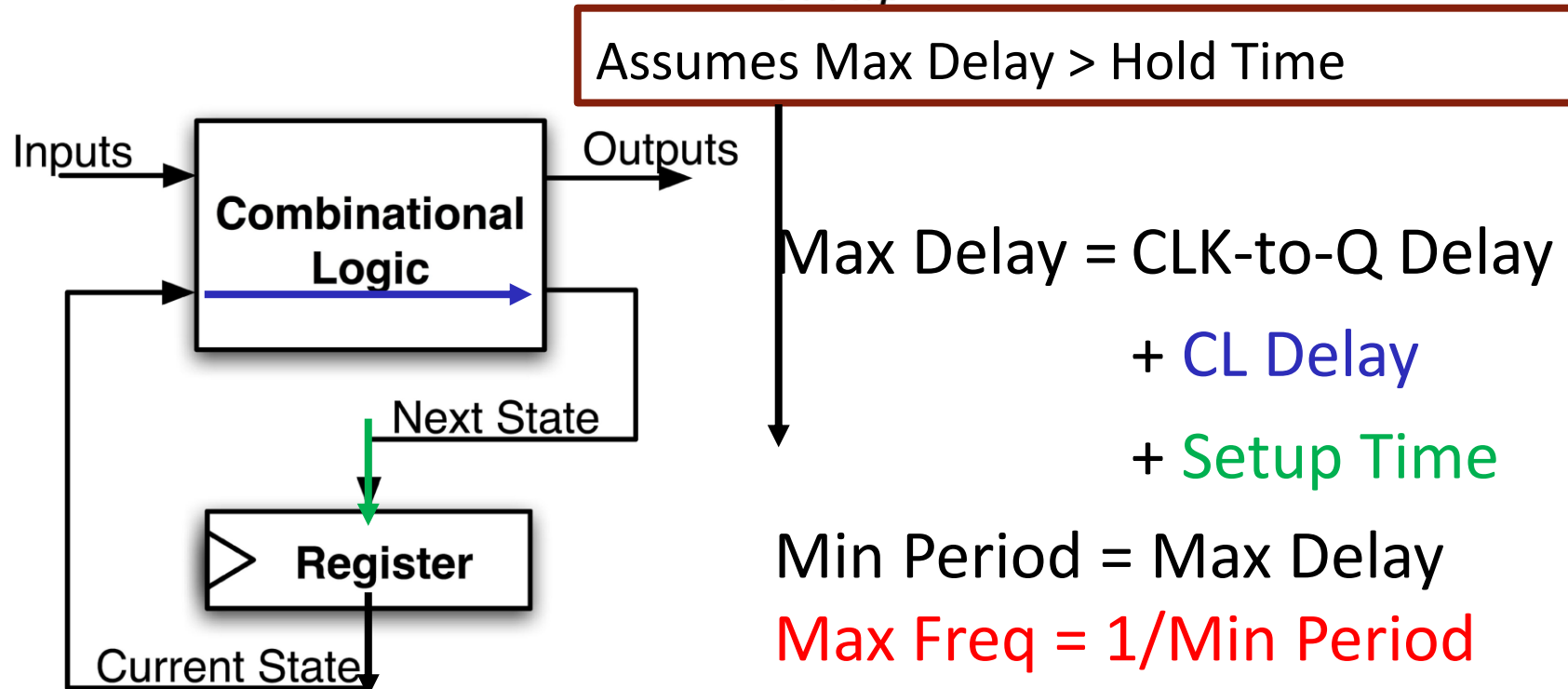
Model for Synchronous Systems



- 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?

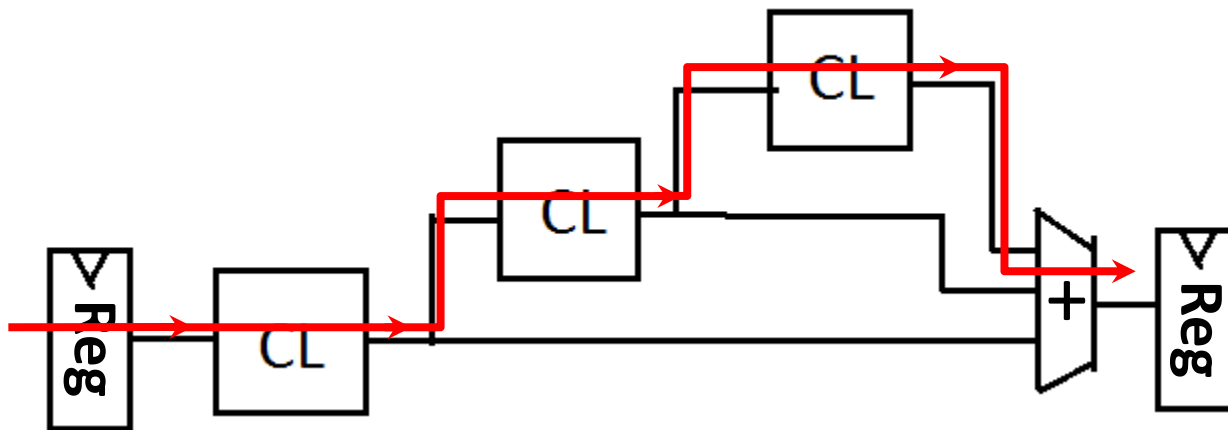
Maximum Clock Frequency

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



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 do we go faster?

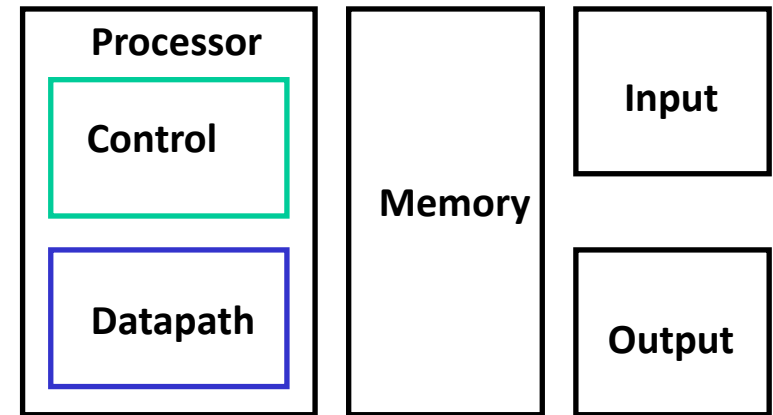
Pipelining!

- Split operation into smaller parts and add a register between each one.

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:

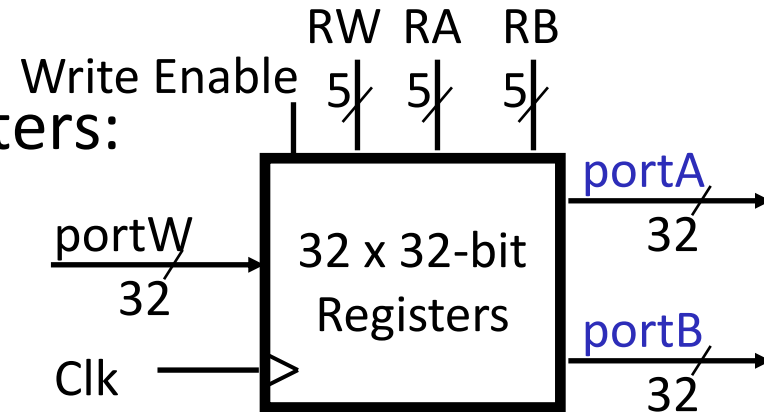
- Output ports *portA* and *portB*
- Input port *portW*

- Register selection

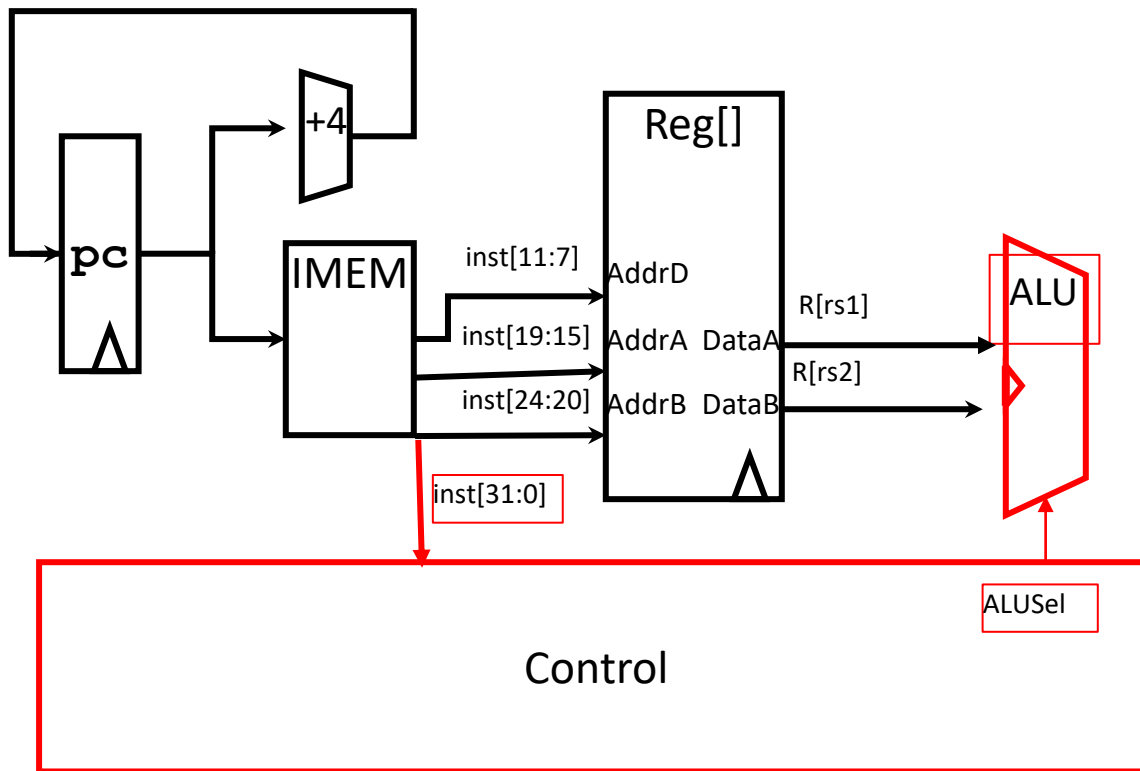
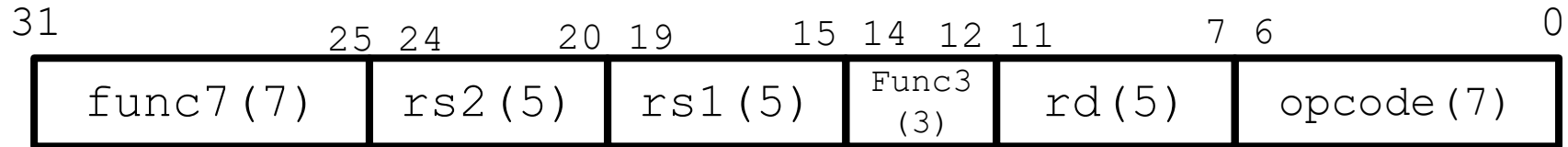
- Place data of register *RA* (number) onto *portA*
- Place data of register *RB* (number) onto *portB*
- Store data on *portW* into register *RW* (number) when Write Enable is 1

- 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



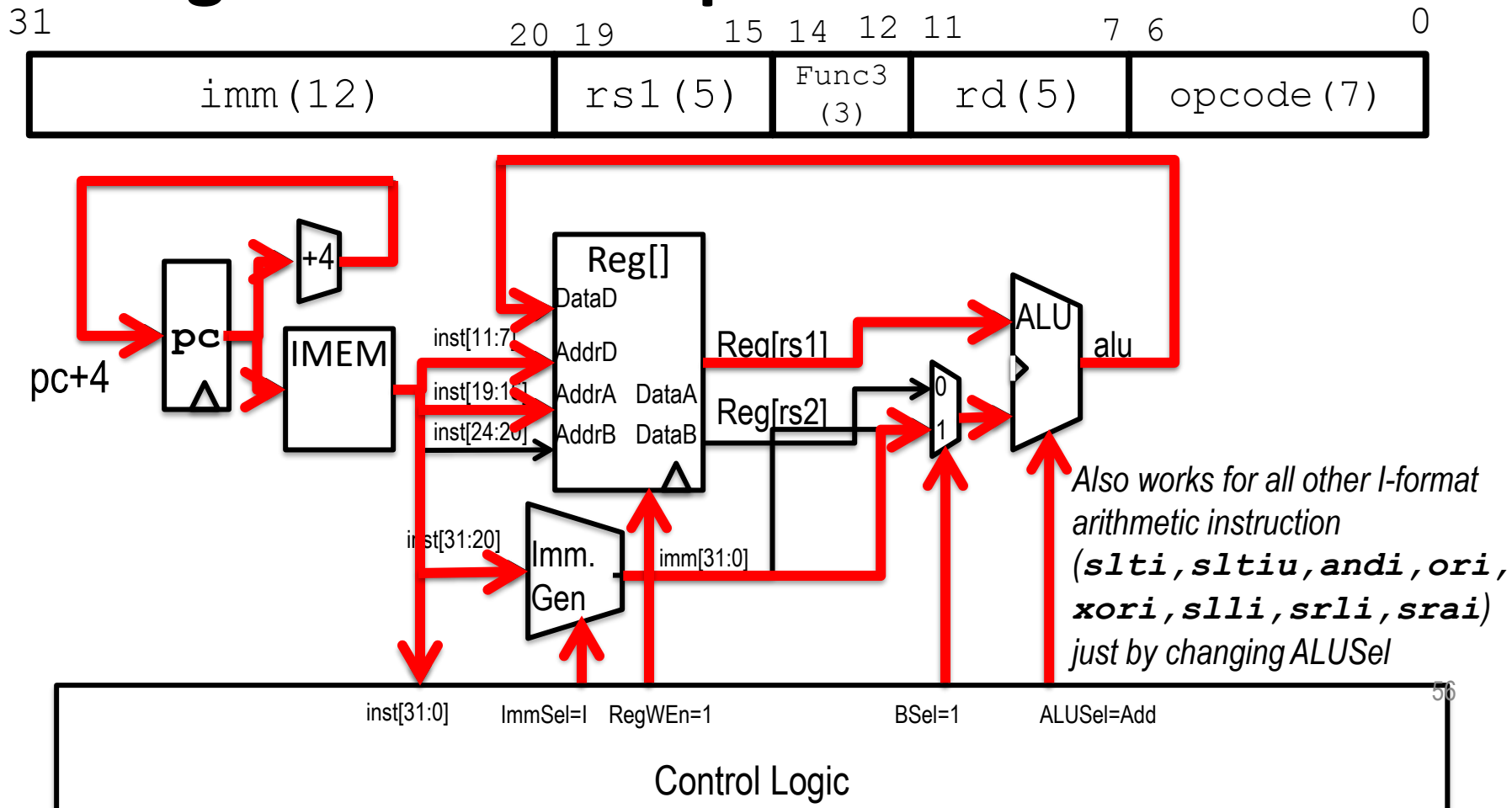
Implementing R-Types



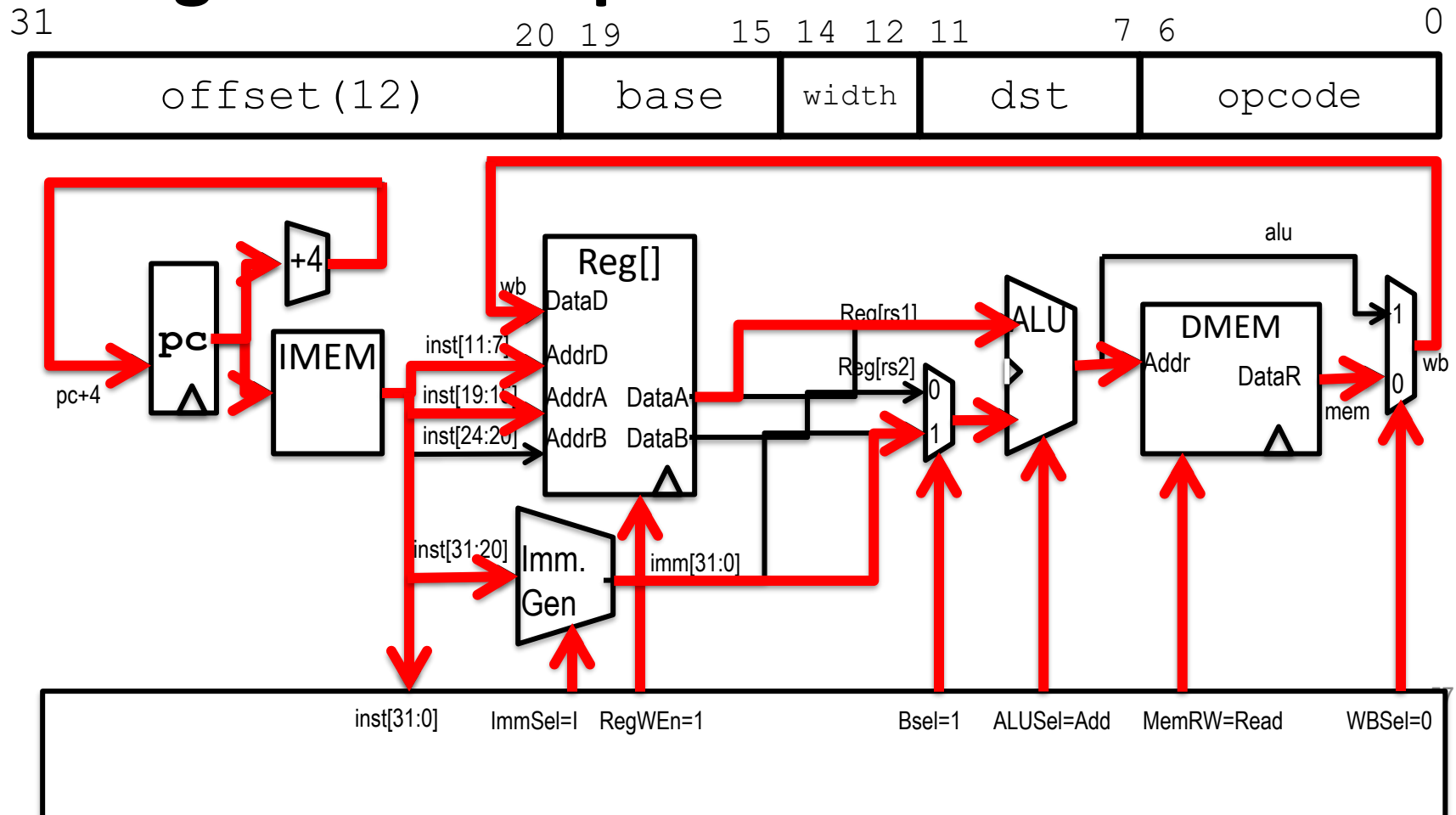
Perform operation

- 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)**

Adding addi to datapath

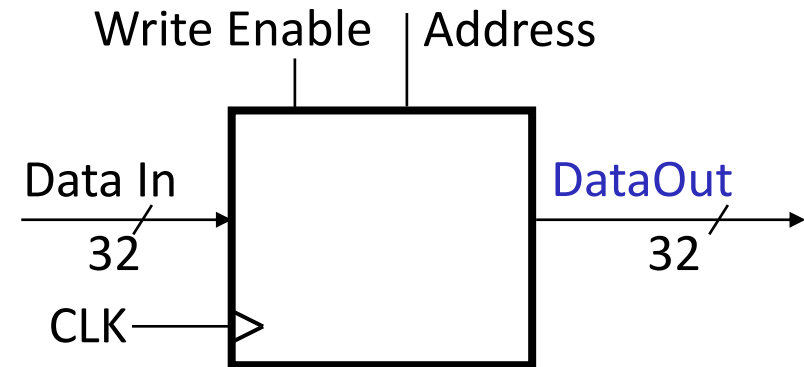


Adding lw to datapath



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

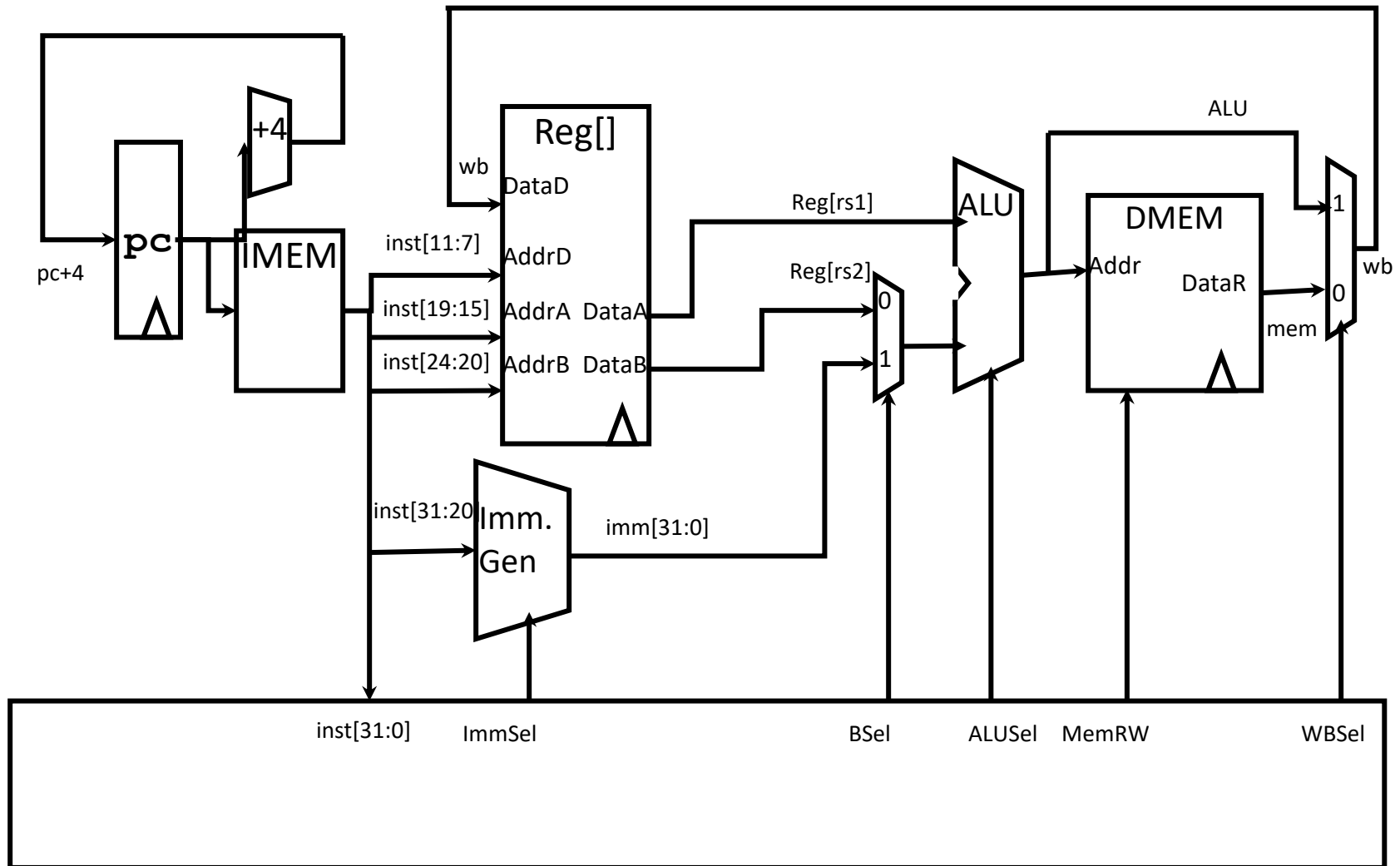
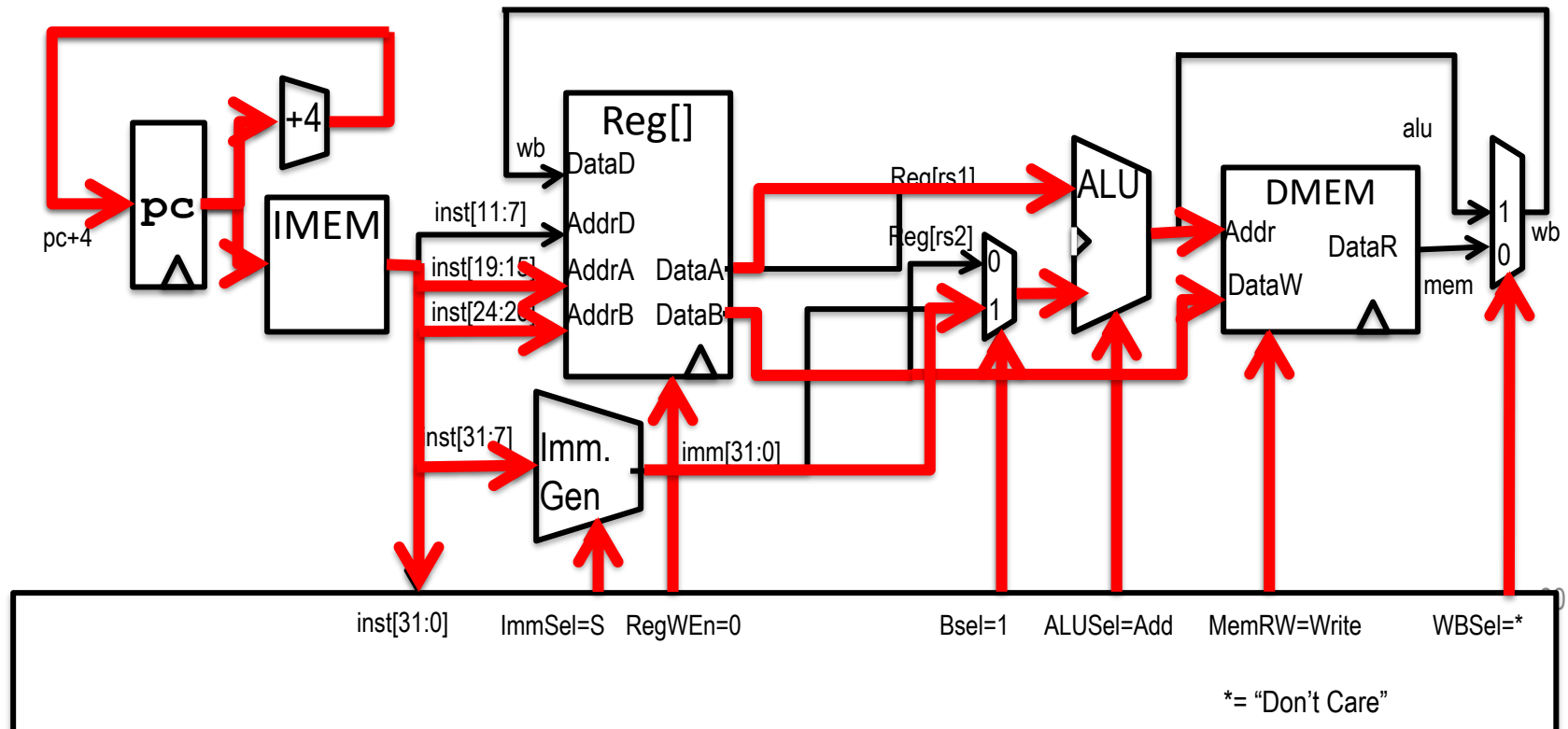
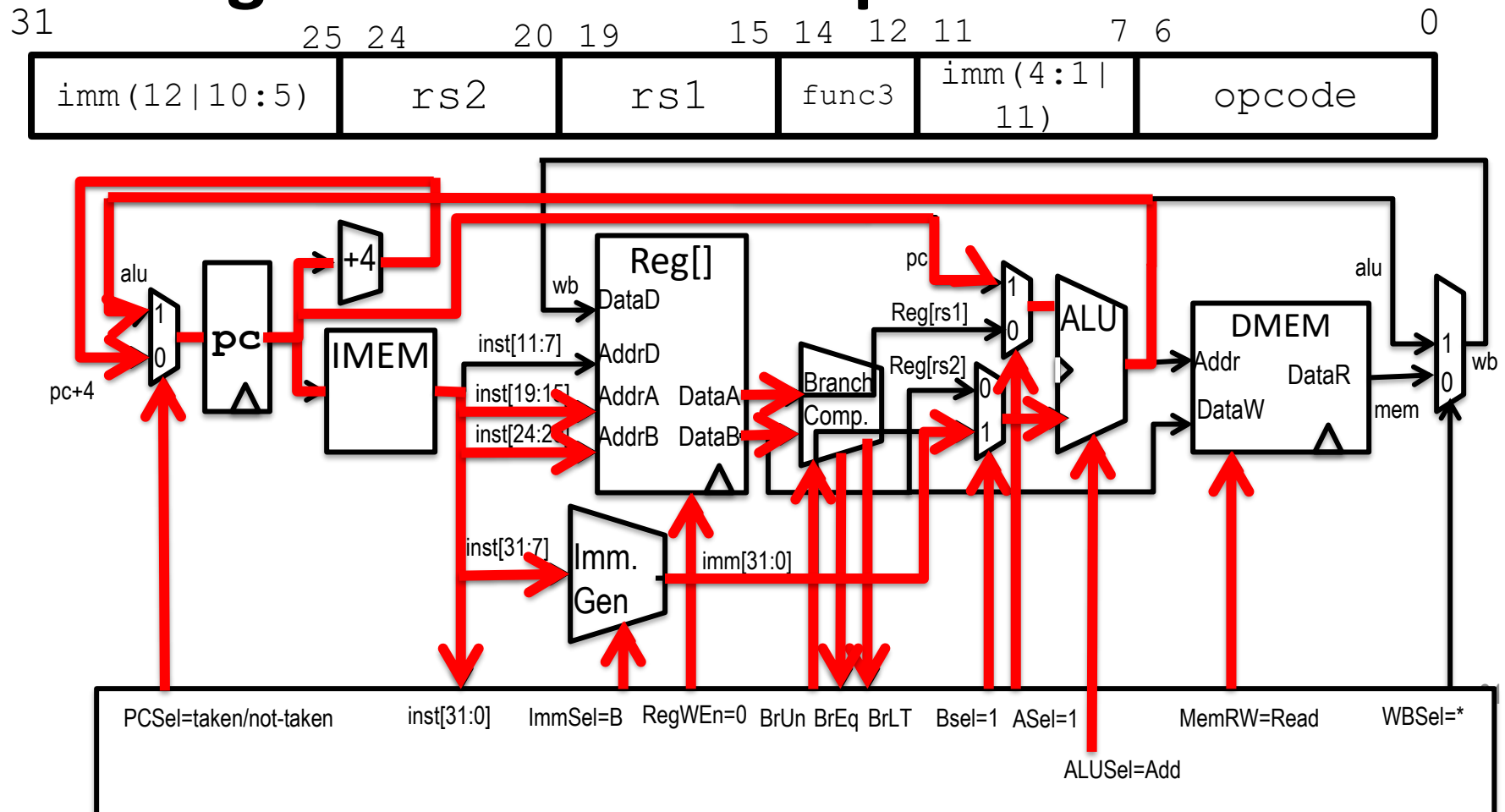


Diagram illustrating the R-format instruction format (32 bits total):

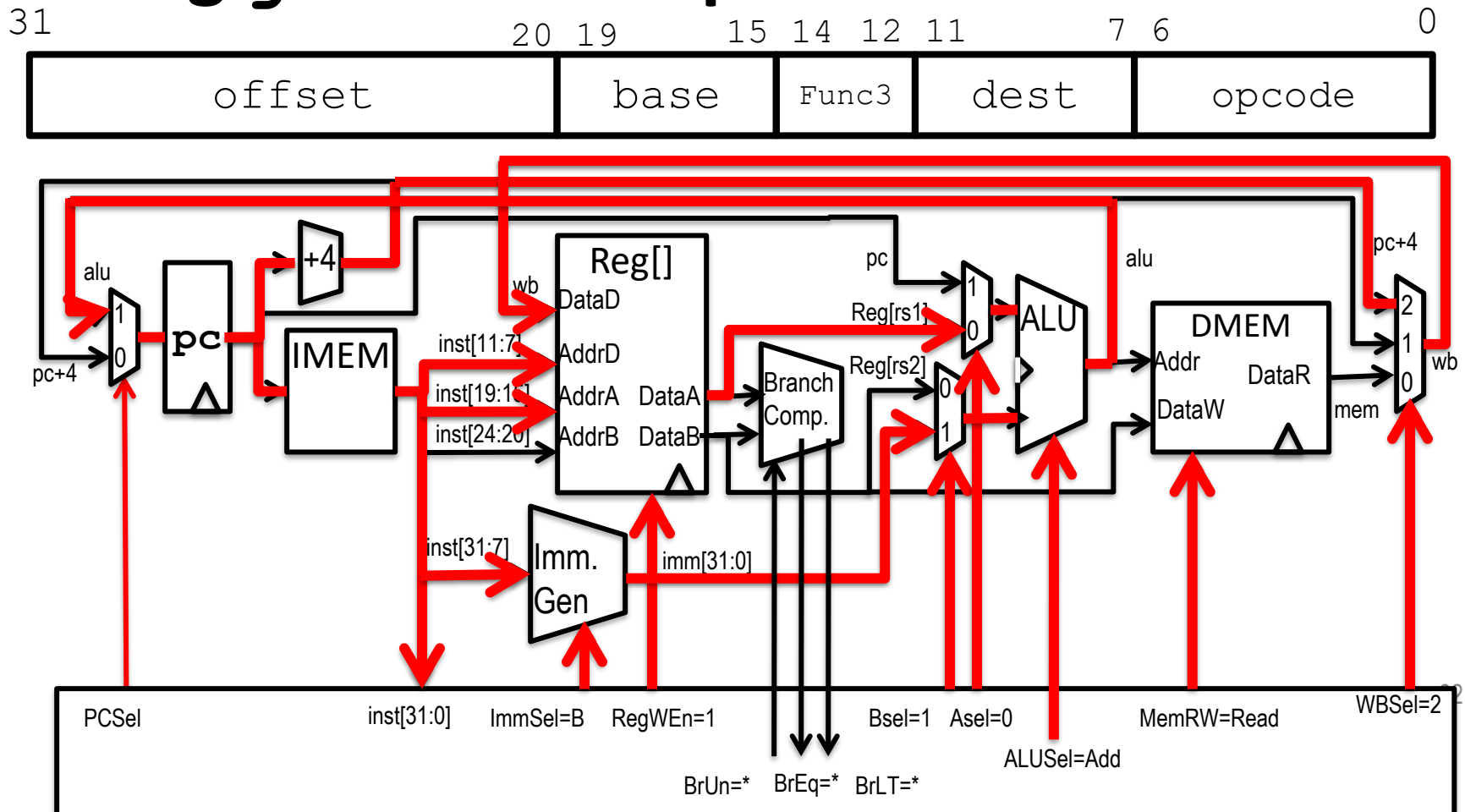
- imm(11:5): 7 bits (bits 31 to 25)
- rs2: 5 bits (bits 24 to 20)
- rs1: 5 bits (bits 19 to 15)
- func3: 3 bits (bits 14 to 12)
- imm(4:0): 5 bits (bits 11 to 7)
- opcode: 6 bits (bits 6 to 0)



Adding branches to datapath



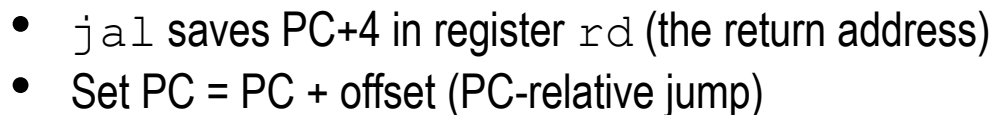
Adding jalr to datapath



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

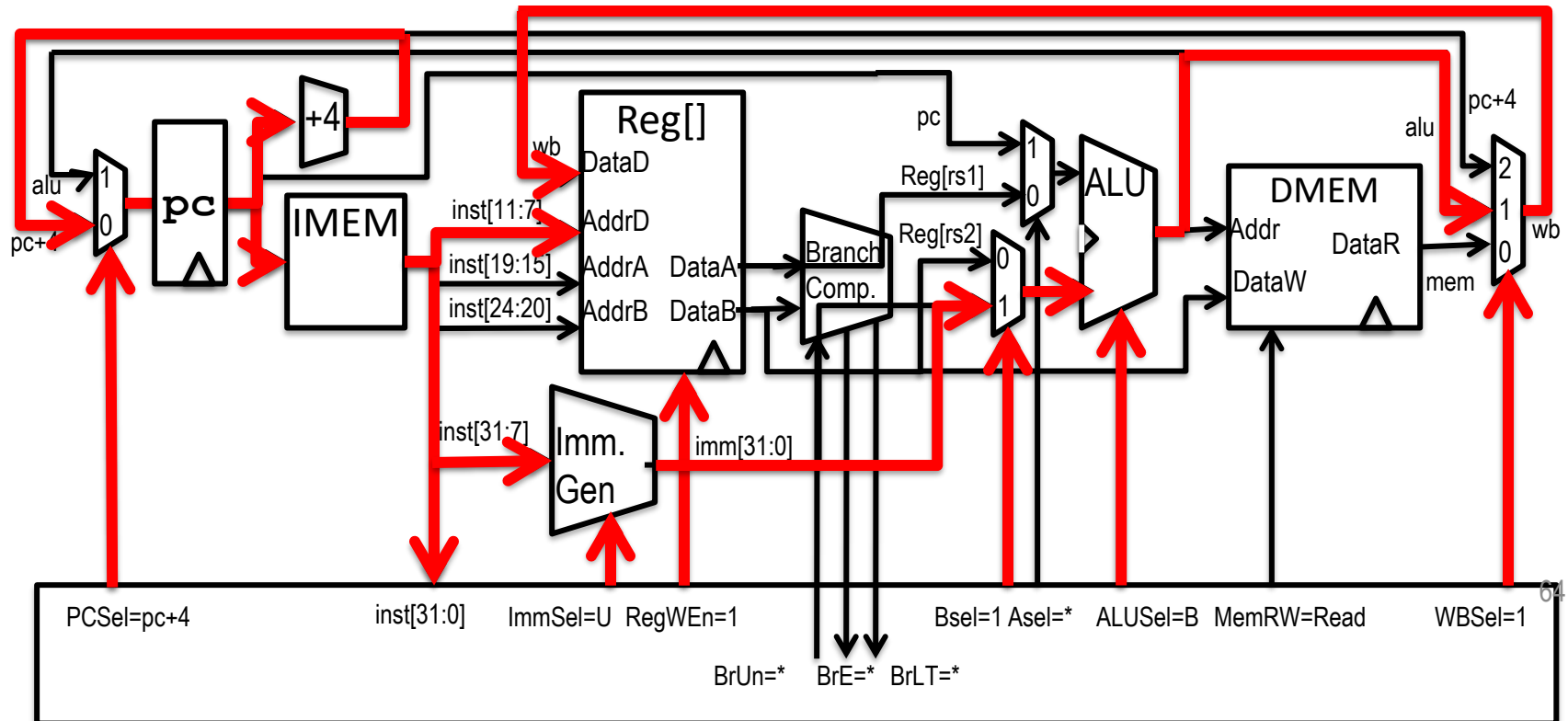
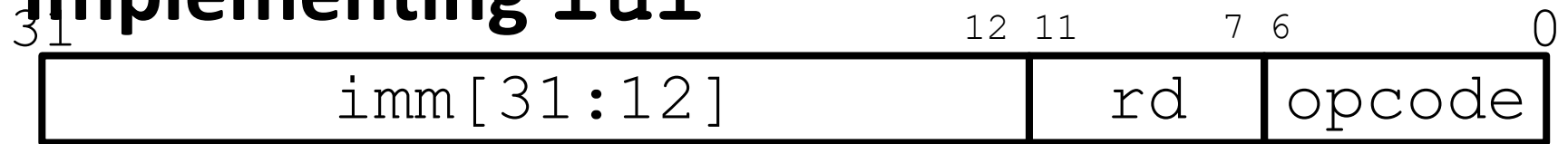
Diagram illustrating the RISC-V instruction format (32 bits):

- imm[20 | 10:1 | 11 | 19:12] (20 bits)
- rd (5 bits)
- opcode (7 bits)



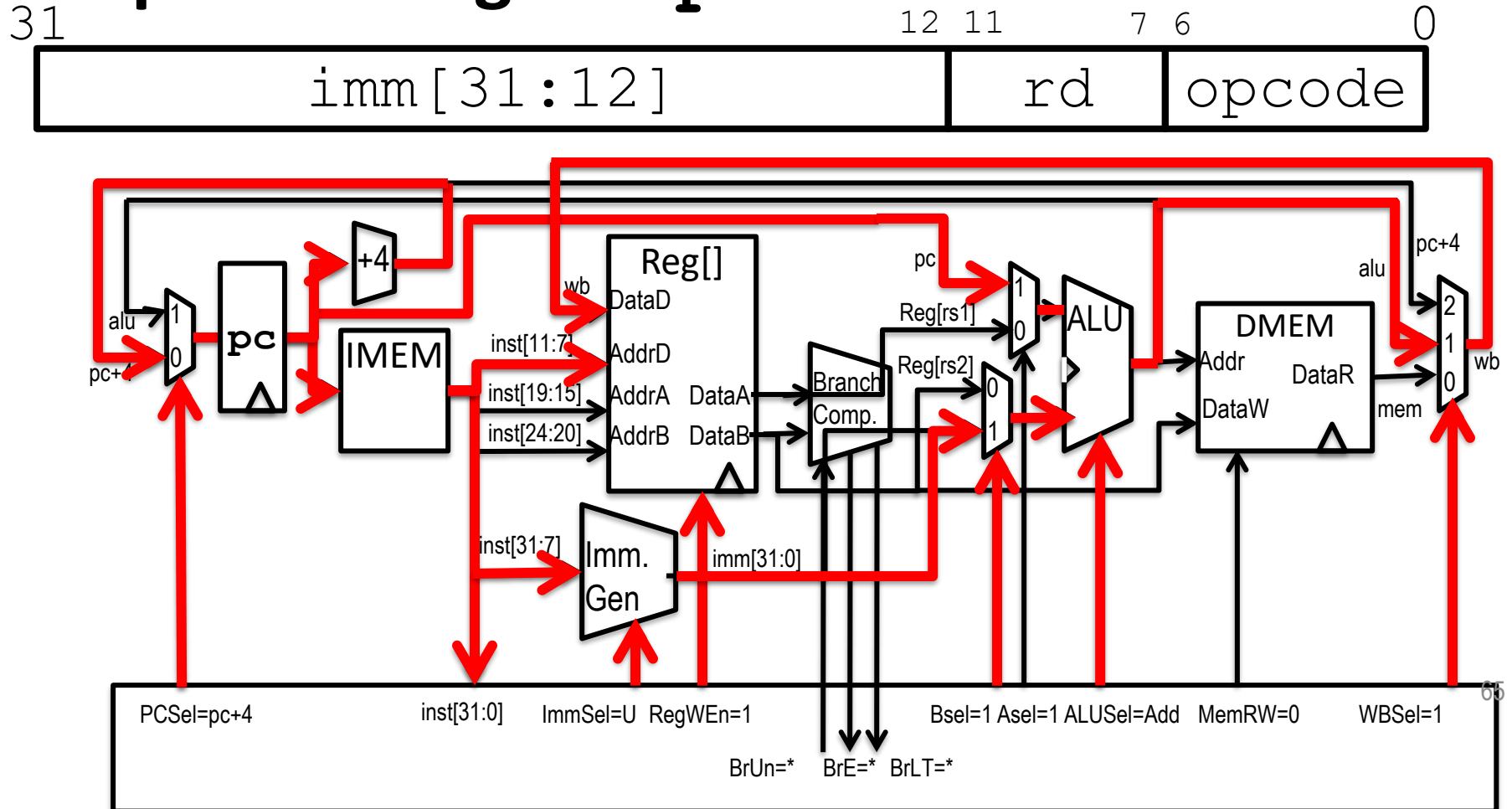
- `jal` saves PC+4 in register `rd` (the return address)
- Set PC = PC + offset (PC-relative jump)

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