CS 211: Intro to Computer Architecture

11.2: Combinational Logic, Flip-Flops, and Clocks

Minesh Patel

Spring 2025 – Thursday 10 April

Announcements

- Ongoing
  - **Extra Credit:** replaces WA6, due **Friday (April 11) @ 11:59 pm**
  - **PA4:** due **Next Friday (April 18) @ 11:59 pm**

- Upcoming
  - **WA8:** Tomorrow (?)

Toward a RV64I CPU

- **Recall:** We can build **anything we want** with {AND, OR, NOT}
- Q: what should we build?

Goal:
Design HW to execute all RV64I instructions

Let's start simple

Logical Operations

```
and(i) rd, rs1, rs2/imm
or(i) rd, rs1, rs2/imm
xor(i) rd, rs1, rs2/imm
```

Add/Subtract

```
add(i)(w) rd, rs1, rs2/imm
sub(w) rd, rs1, rs2
```

Compare

```
slt(u)(i) rd, rs1, rs1/imm
```

Shift

```
sll(i)(w) rd, rs1, rs2/imm
srl(i)(w) rd, rs1, rs2/imm
sra(i)(w) rd, rs1, rs2/imm
```

Memory

```
l{b/h/w/d} rd, offset(rs)
lu{b/h/w/d} rd, offset(rs)
s{b/h/w/d} rd, offset(rs)
```

Branch/Jump

```
beq rs1, rs2, <label>
bne rs1, rs2, <label>
blt(u) rs1, rs2, <label>
bge(u) rs1, rs2, <label>
jal rd, <label>
jalr rd, offset(rs1)
```

Load/Add Upper Immediate

```
lui rd, imm
auipc rd, imm
```

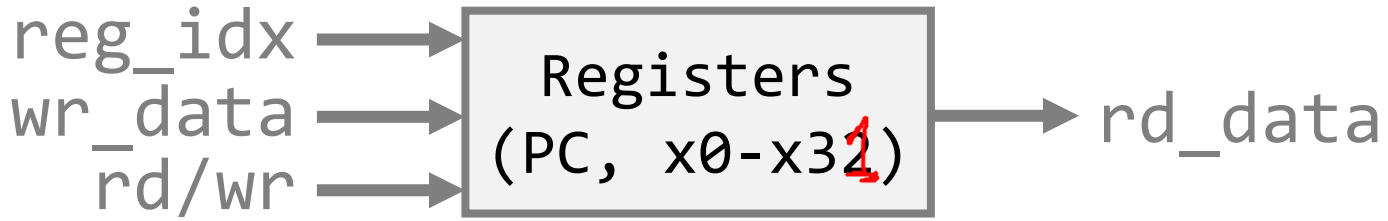
Implementing Logical Operations

```
and(i) rd, rs1, rs2/imm
or(i) rd, rs1, rs2/imm
xor(i) rd, rs1, rs2/imm
```

- We must build circuits to:

1. Perform **and/or/xor** calculations

2. **Registers** to hold data

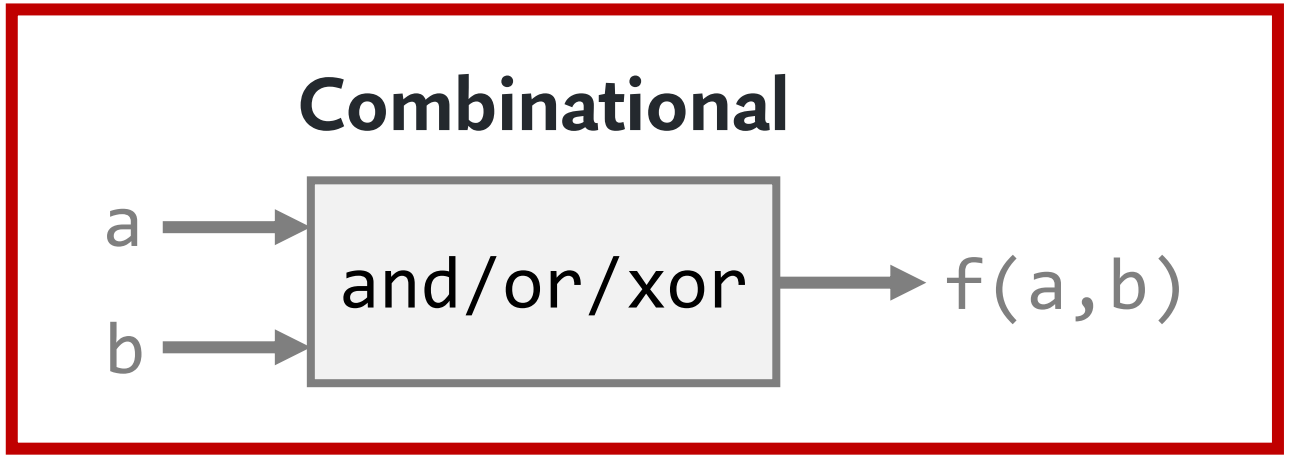


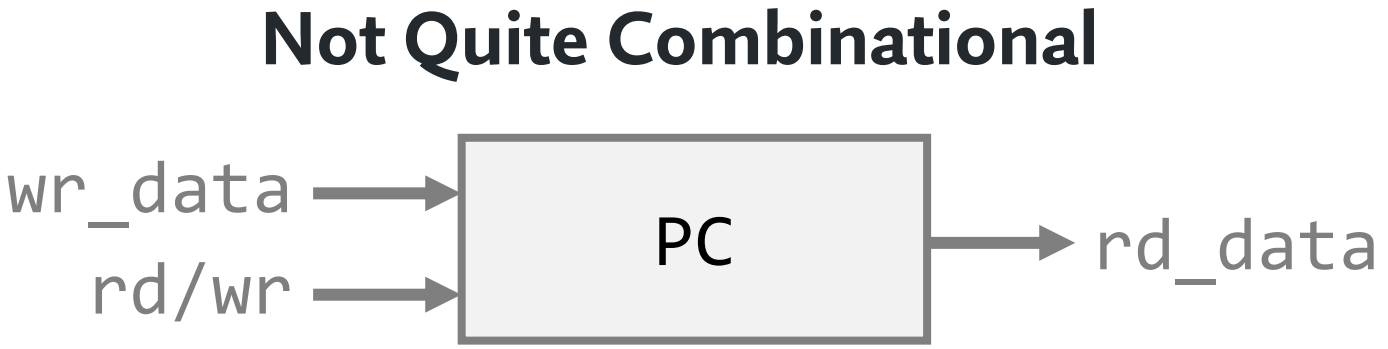
Agenda

- **Combinational Logic**
  - Logical Operations
  - Addition/Subtraction
  - Comparing Numbers
  - Bit Shifting
- Building Registers

Combinational Logic

- Any function where:
  - Output depends **only on inputs**
  - **No memory** of past inputs (i.e., previous states)



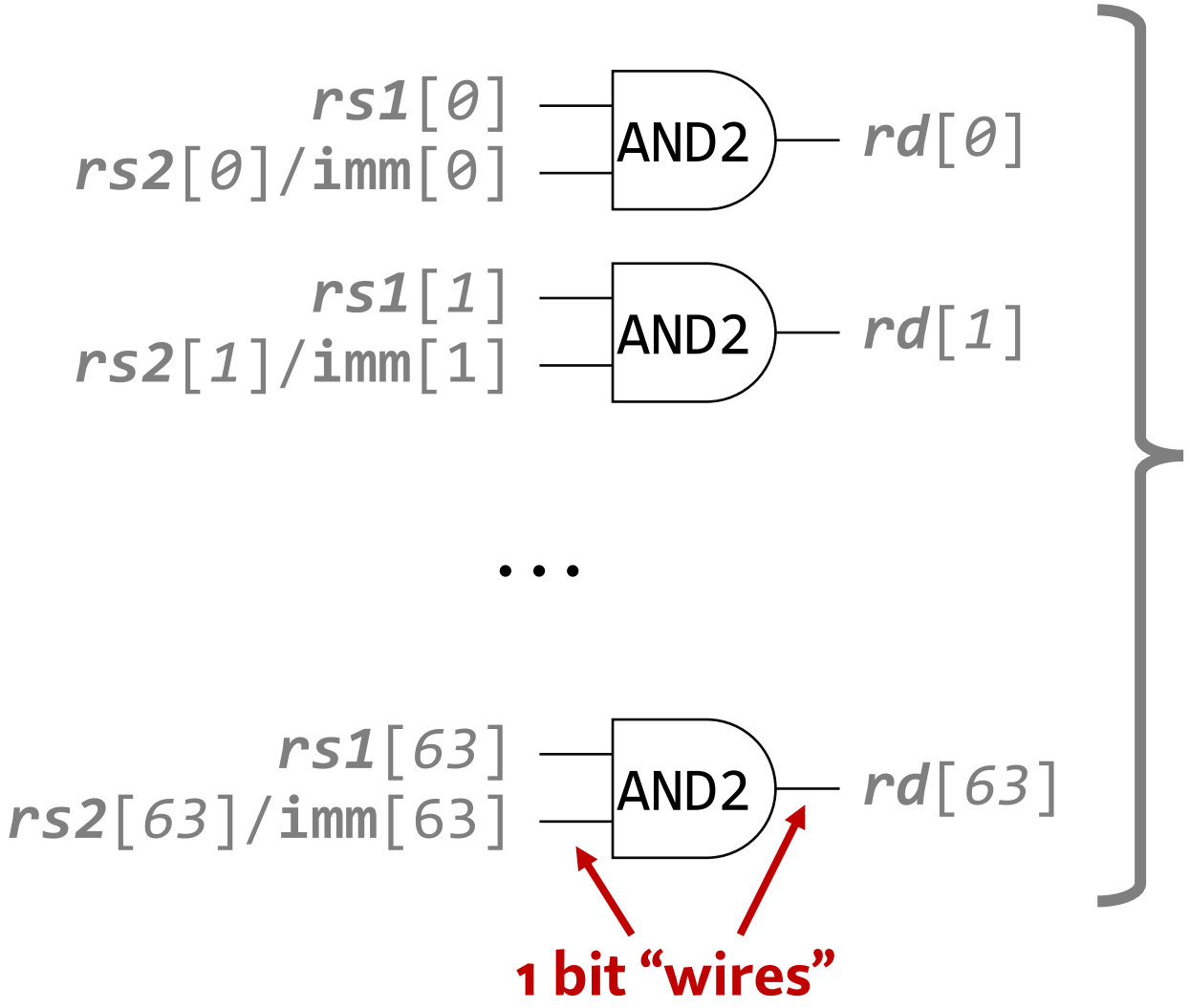


Agenda

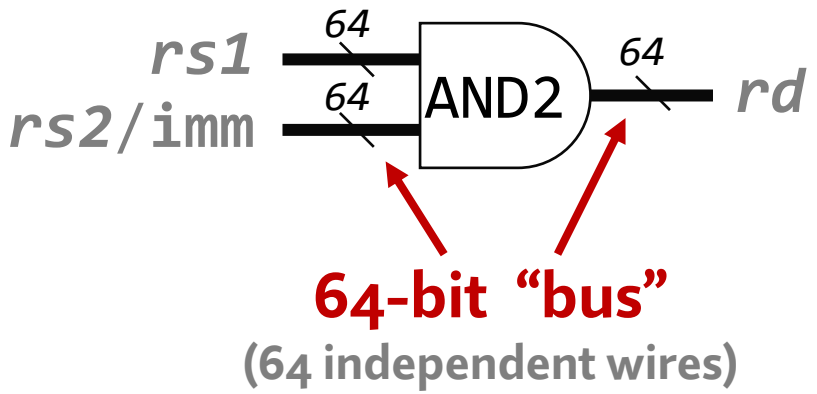
- Combinational Logic
  - **Logical Operations**
  - Addition/Subtraction
  - Comparing Numbers
  - Bit Shifting
- Building Registers

and(i) rd, rs1, rs2/imm

- It's just **64 AND gates**

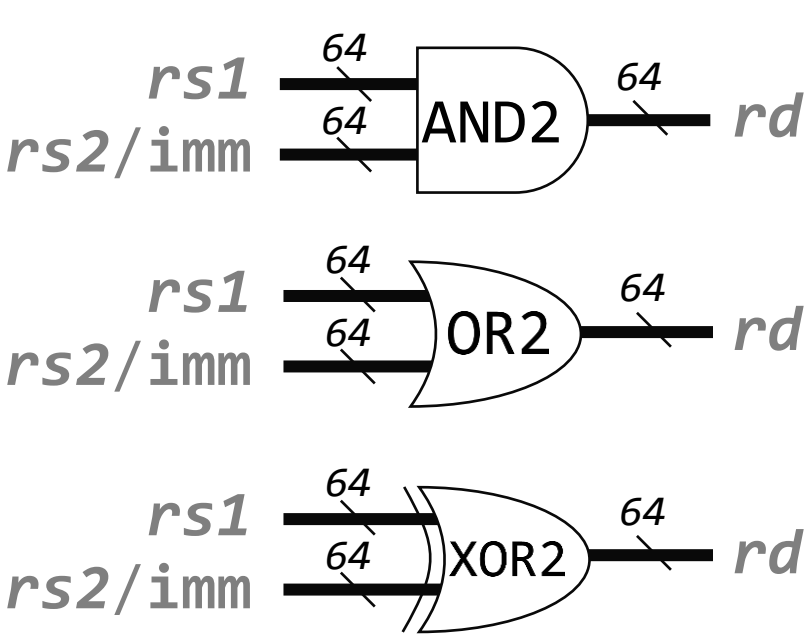


64 ANDs in Parallel



Supporting Multiple Logic Operations

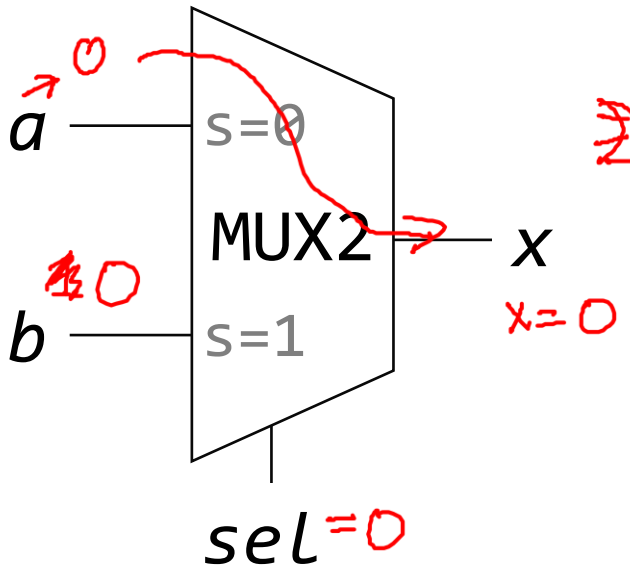
```
and(i) rd, rs1, rs2/imm
or(i) rd, rs1, rs2/imm
xor(i) rd, rs1, rs2/imm
```



Need to **choose** between these depending on the **instruction**

Multiplexer: Choosing A or B

abs	x
000	0
001	0
010	0
011	1
100	1
101	0
110	1
111	1



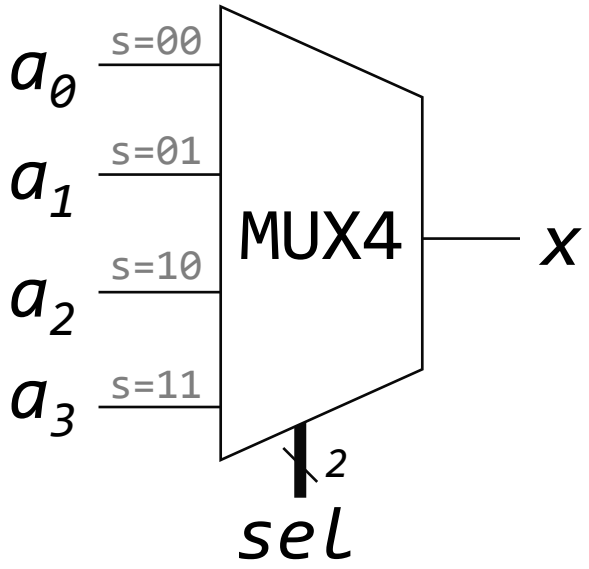
Simpler Truth Table

s	x
0	a
1	b

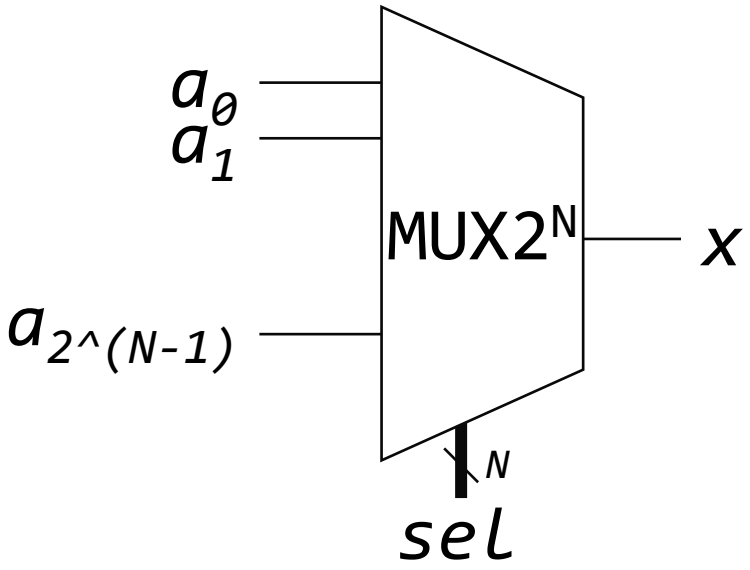
Algebraic Representation

```
f(a,b,s) = a'bs + ab's' + abs' + abs
         = ...
         = as' + bs
```

Multiplexer: Choosing Between N

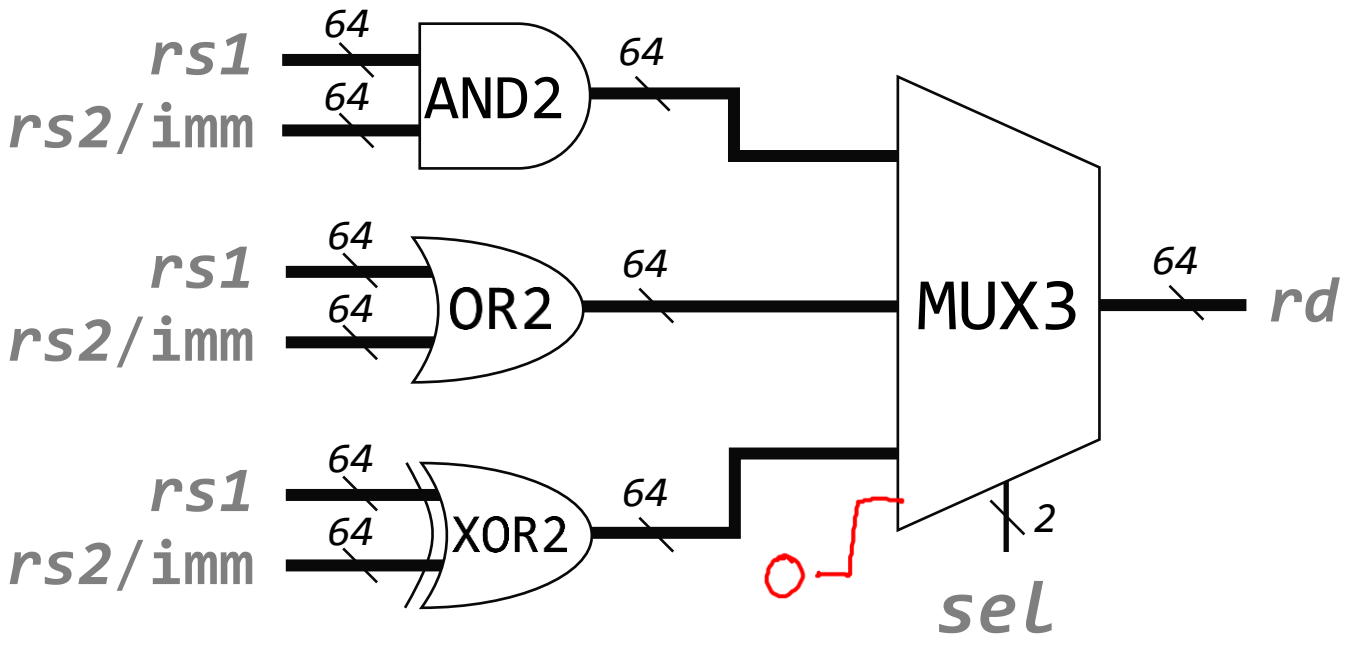


s	x
00	a_0
01	a_1
10	a_2
11	a_3



Re: Choosing Between Logic Operations

Key idea: calculate everything, and choose **the right output**

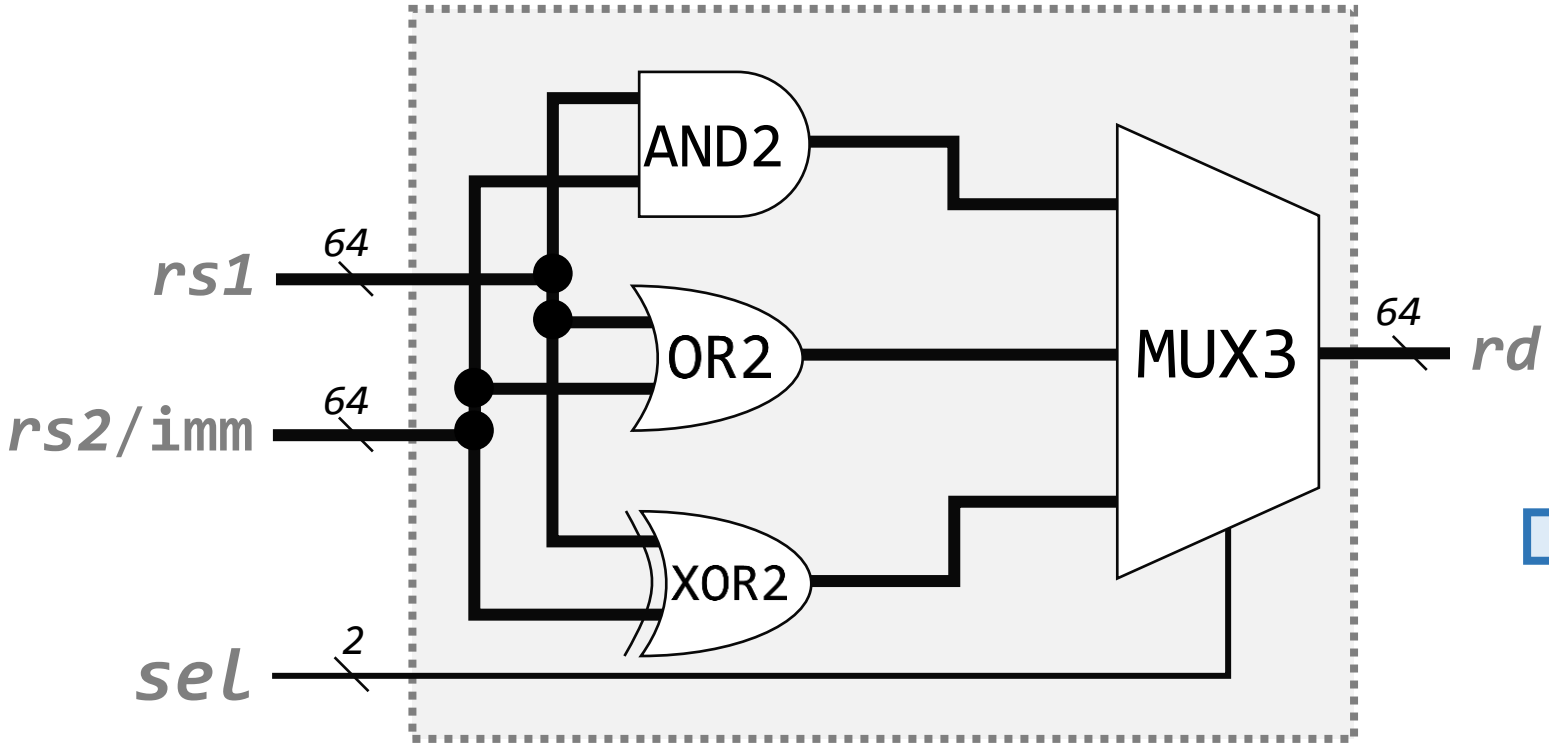


sel	op
00	AND
01	OR
10	XOR
11	-

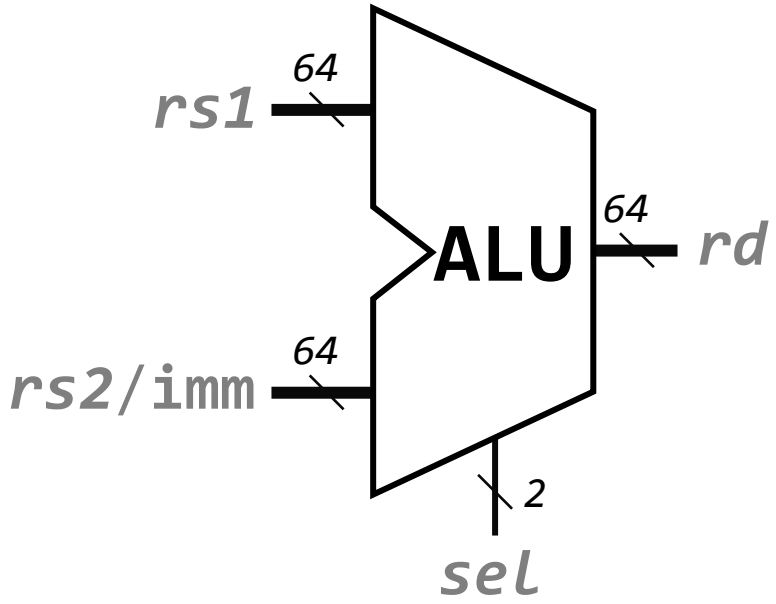
```
and(i) rd, rs1, rs2/imm
or(i) rd, rs1, rs2/imm
xor(i) rd, rs1, rs2/imm
```

Arithmetic and Logic Unit (ALU)

ALU (*Work in Progress*)



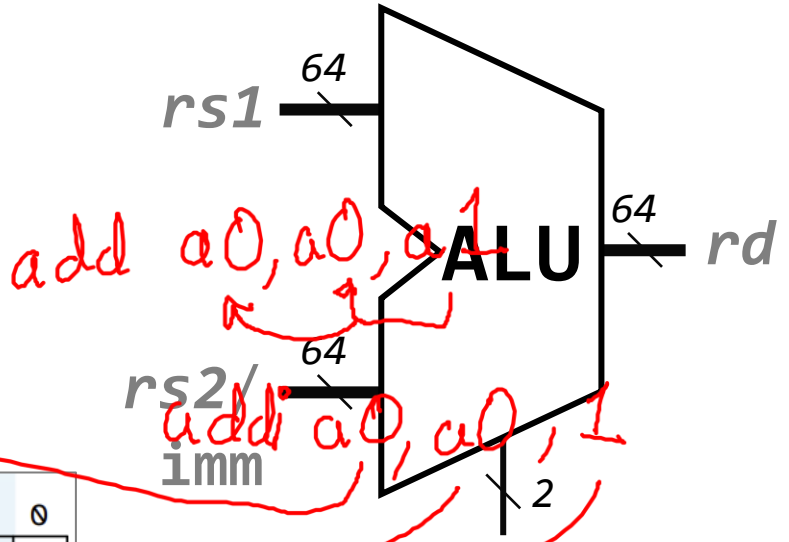
"Logic Block"



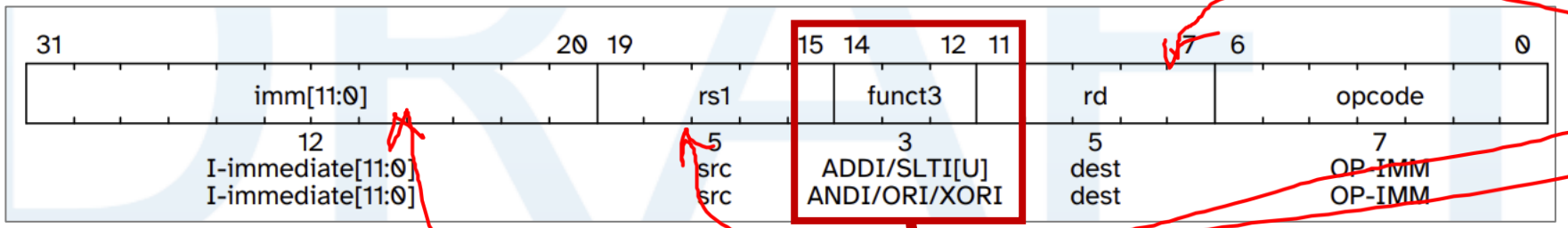

```
and(i) rd, rs1, rs2/imm
or(i) rd, rs1, rs2/imm
xor(i) rd, rs1, rs2/imm
```

Choosing the Right Operation

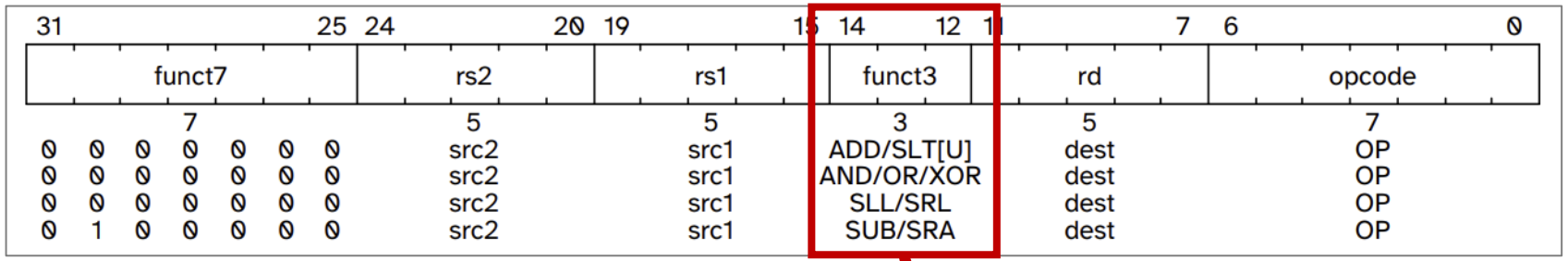
- Can figure it out from the machine code



I-Type Encoding (Register-Immediate)



R-Type Encoding (Register-Register)

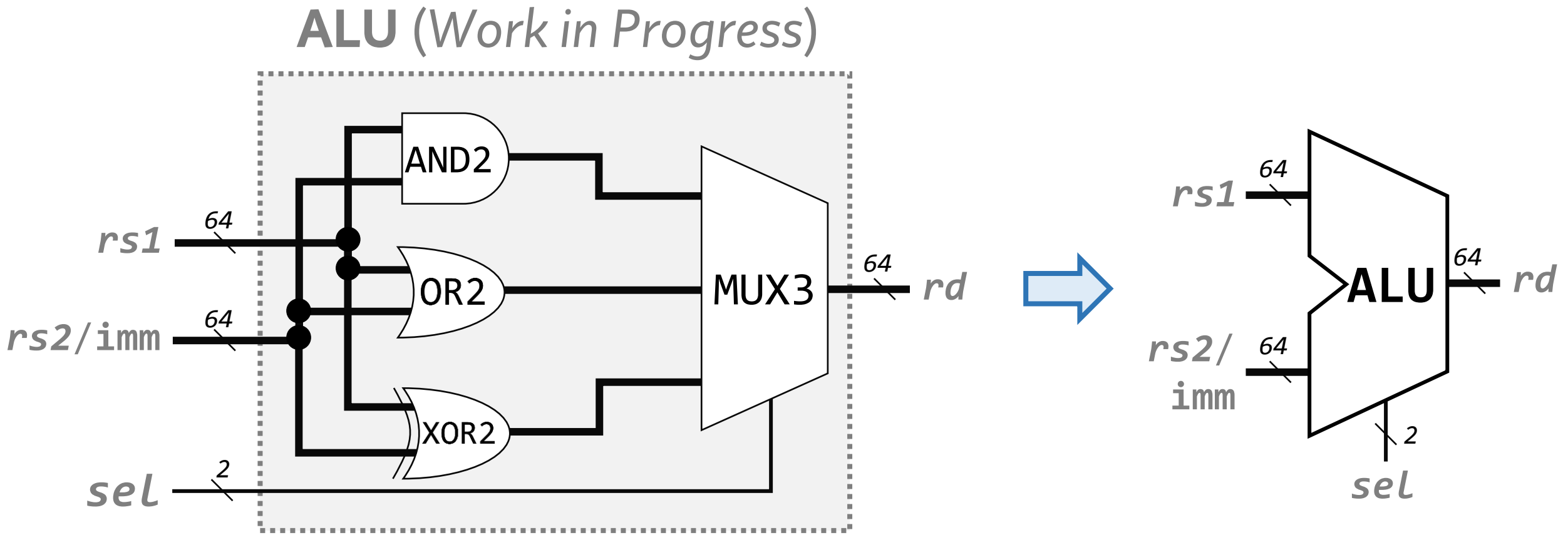



```
00 (and)
01 (or)
10 (xor)
11 (...)
```

Agenda

- Combinational Logic
  - Logical Operations
  - **Addition/Subtraction**
  - Comparing Numbers
  - Bit Shifting
- Building Registers

Extending the ALU



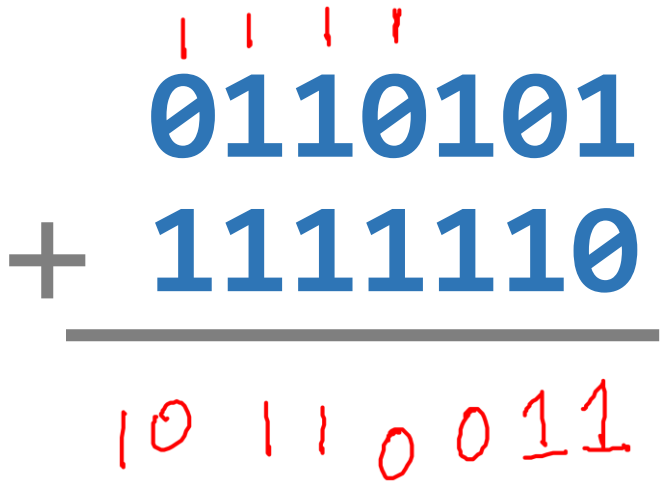
Logical Operations	Add/Subtract	Compare	Shift
`and(i) rd, rs1, rs2/imm`	`add(i)(w) rd, rs1, rs2/imm`	`slt(u)(i) rd, rs1, rs1/imm`	`sll(i)(w) rd, rs1, rs2/imm`
`or(i) rd, rs1, rs2/imm`	`sub(w) rd, rs1, rs2`		`srl(i)(w) rd, rs1, rs2/imm`
`xor(i) rd, rs1, rs2/imm`			`sra(i)(w) rd, rs1, rs2/imm`

Adder/Subtractor

Adding 64-bit Numbers

`add(i) rd, rs1, rs2/imm`

- How do we do binary addition?



Adding 64-bit Numbers

```
add(i) rd, rs1, rs2/imm
```

- How do we do binary addition?

Hard case:

Both c_{in}/c_{out}

```
  1 1 1 1
  0 1 1 0 1 0 1 1
+ 1 1 1 1 1 1 1 0
-----------------
1 0 1 1 0 0 1 1 1
```

Easy case:

no c_{in}/c_{out}

Worst case

Both c_{in}/c_{out}

```
  1 1 1 1 1 1 1
  1 1 1 1 1 1 1 1
+ 1 1 1 1 1 1 1 1
-----------------
1 1 1 1 1 1 1 1 0
```

No c_{in}

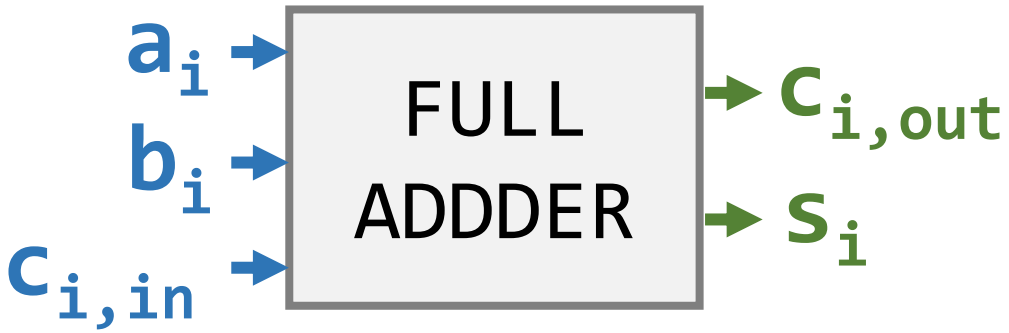
Binary Addition Algorithm: Carry Ripple

$$
\begin{array}{rccccccccc}
 & c_{64} & c_{63} & & & c_3 & c_2 & c_1 & 0 \\
 & & a_{63} & \cdots & a_3 & a_2 & a_1 & a_0 \\
+ & & b_{63} & \cdots & b_3 & b_2 & b_1 & b_0 \\
\hline
 & s_{64} & s_{63} & \cdots & s_3 & s_2 & s_1 & s_0
\end{array}
$$

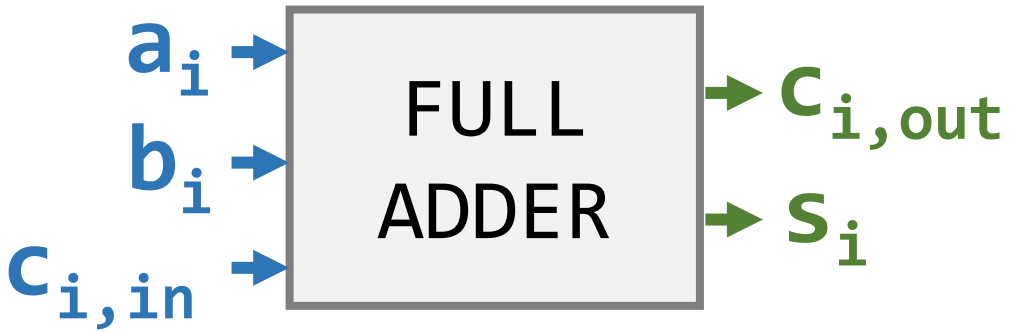
bit-by-bit

For Each Bit

$$
\begin{array}{rcc}
 & c_{i,out} & c_{i,in} \\
 & & a_i \\
+ & & b_i \\
\hline
 & & s_i
\end{array}
$$

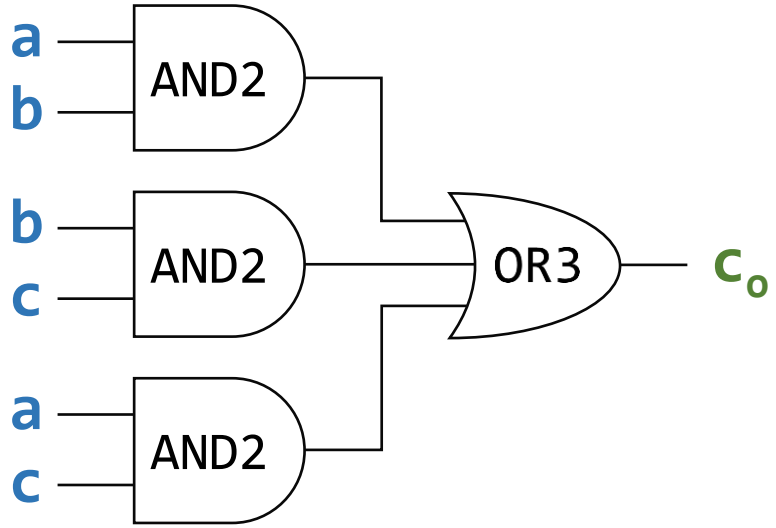


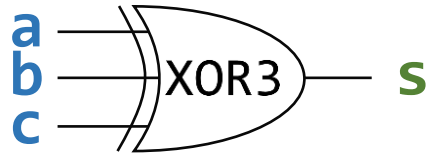
Full Adder Implementation



$$c_o = MAJ(a, b, c_i)$$
$$s = a \oplus b \oplus c_i$$

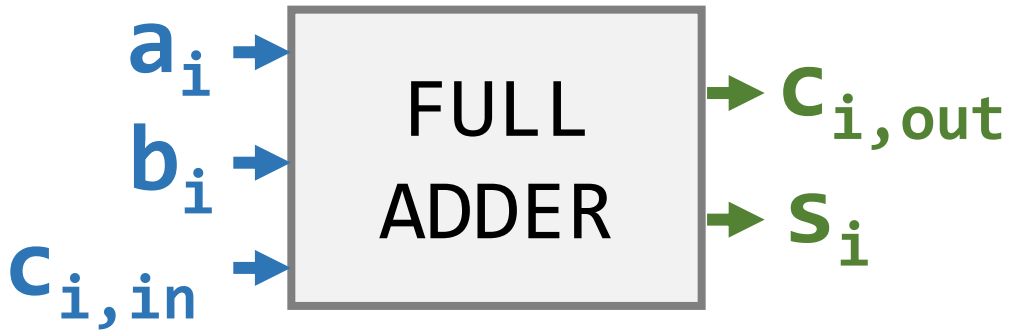
abc_i	$c_o s$
000	0 0
001	0 1
010	0 1
011	1 0
100	0 1
101	1 0
110	1 0
111	1 1



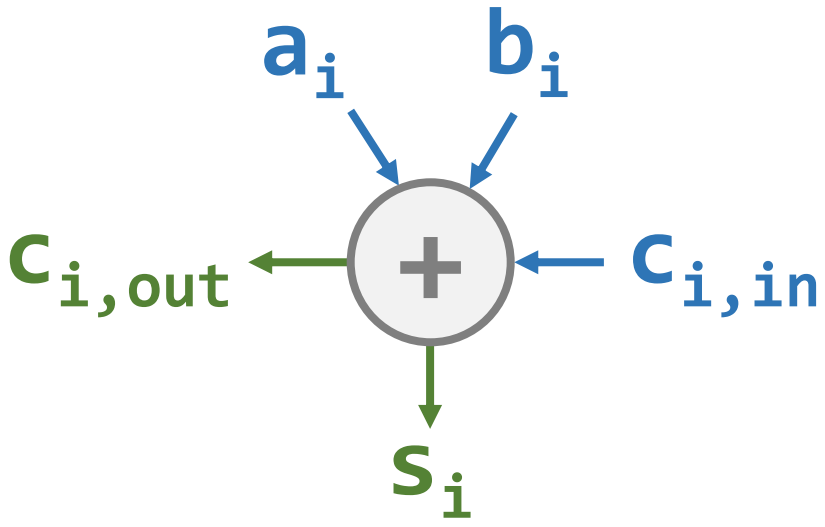


Chaining Full Adders

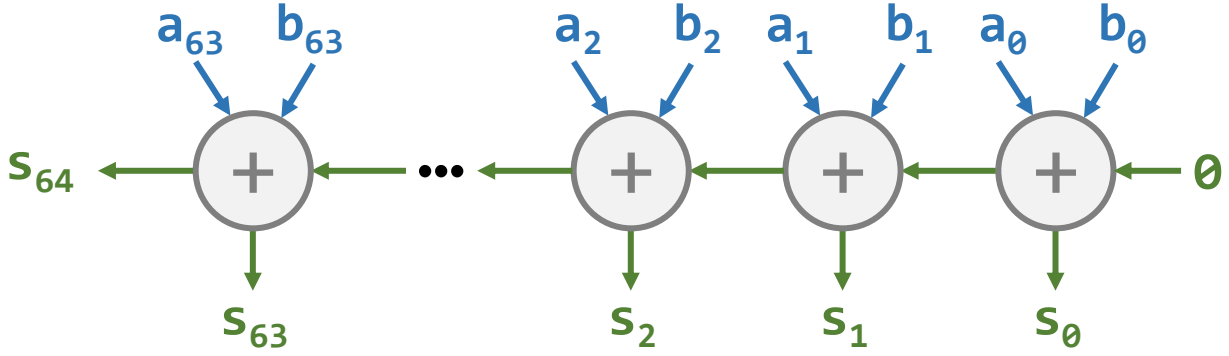
"Full Adder"



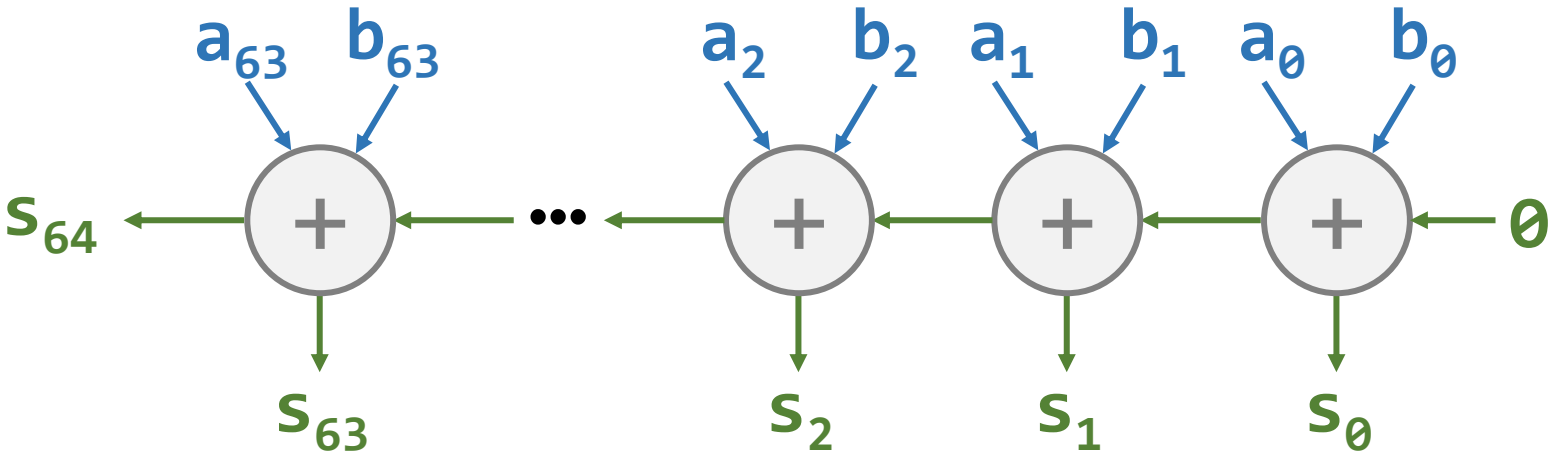
$$\begin{array}{cccccccc} c_{64} & c_{63} & & & c_3 & c_2 & c_1 & 0 \\ & a_{63} & \cdots & & a_3 & a_2 & a_1 & a_0 \\ + & b_{63} & \cdots & & b_3 & b_2 & b_1 & b_0 \\ \hline s_{64} & s_{63} & \cdots & & s_3 & s_2 & s_1 & s_0 \end{array}$$

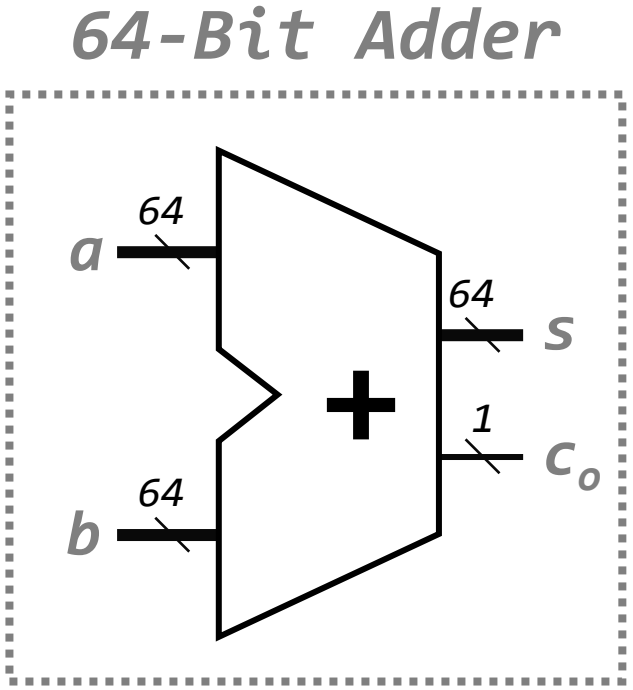


64-Bit Adder



64-Bit Adder

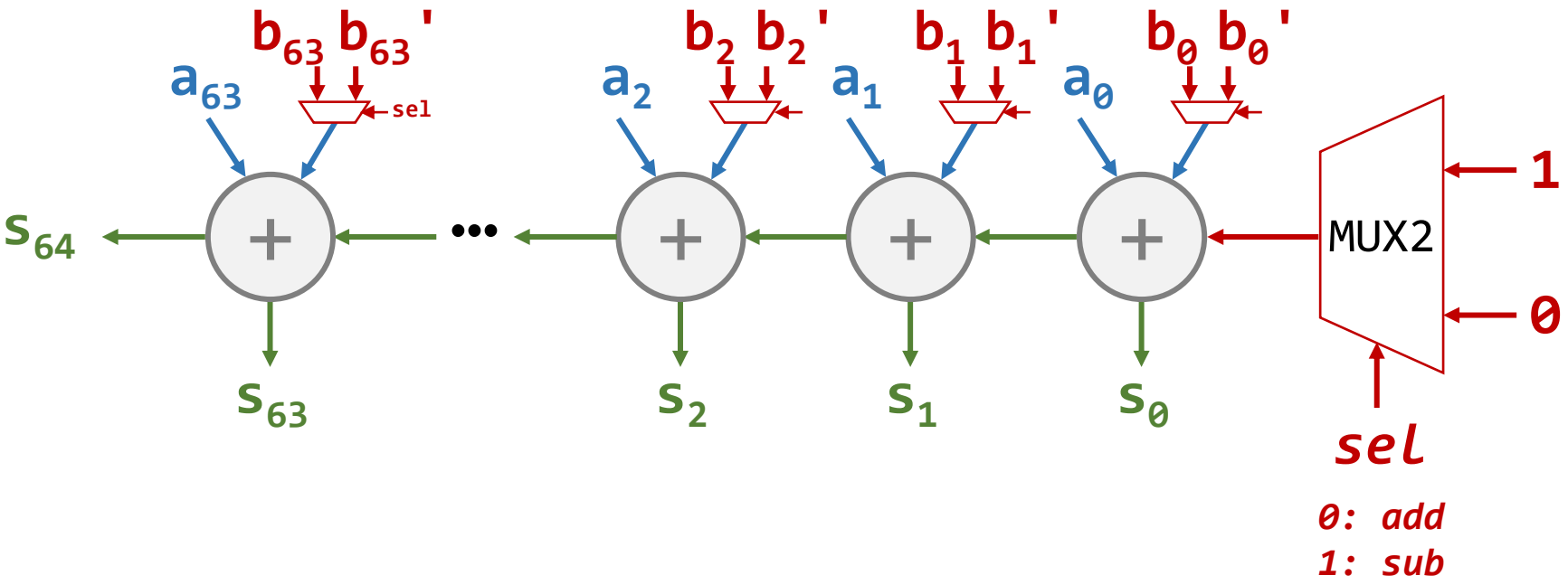

a_{63} b_{63} a_2 b_2 a_1 b_1 a_0 b_0
s_{64}
+
•••
0
s_{63} s_2 s_1 s_0


64-Bit Adder
a
64
b
64
+
64
s
1
c_o

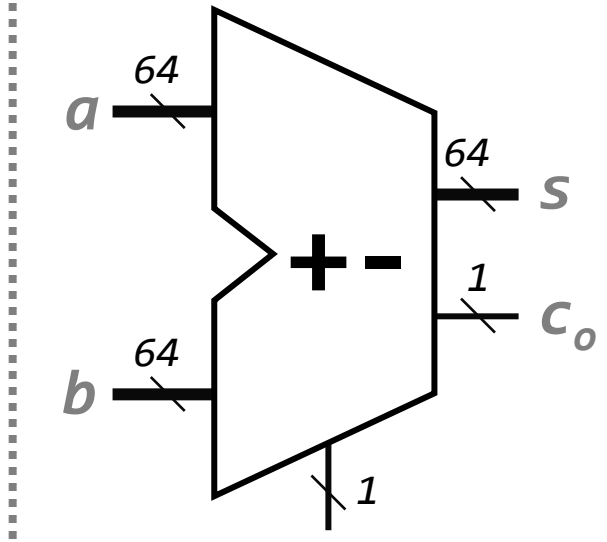
64-Bit Subtractor

- Exploit two's complement!

$$A - B = A + B' + 1$$



64-Bit Add/Sub



Extra-Fancy Integer Addition Algorithms

TABLE 11.3 Comparison of adder architectures

Architecture	Classification	Logic Levels	Max Fanout	Tracks	Cells
Carry-Ripple		$N-1$	1	1	N
Carry-Skip ($n=4$)		$N/4+5$	2	1	$1.25N$
Carry-Increment ($n=4$)		$N/4+2$	4	1	$2N$
Carry-Increment (variable group)		$\sqrt{2N}$	$\sqrt{2N}$	1	$2N$
Brent-Kung	$(L-1, 0, 0)$	$2\log_2 N-1$	2	1	$2N$
Sklansky	$(0, L-1, 0)$	$\log_2 N$	$N/2+1$	1	$0.5\,N\log_2 N$
Kogge-Stone	$(0, 0, L-1)$	$\log_2 N$	2	$N/2$	$N\log_2 N$
Han-Carlson	$(1, 0, L-2)$	$\log_2 N+1$	2	$N/4$	$0.5\,N\log_2 N$
Ladner Fischer ($l=1$)	$(1, L-2, 0)$	$\log_2 N+1$	$N/4+1$	1	$0.25\,N\log_2 N$
Knowles [2,1,…,1]	$(0, 1, L-2)$	$\log_2 N$	3	$N/4$	$N\log_2 N$

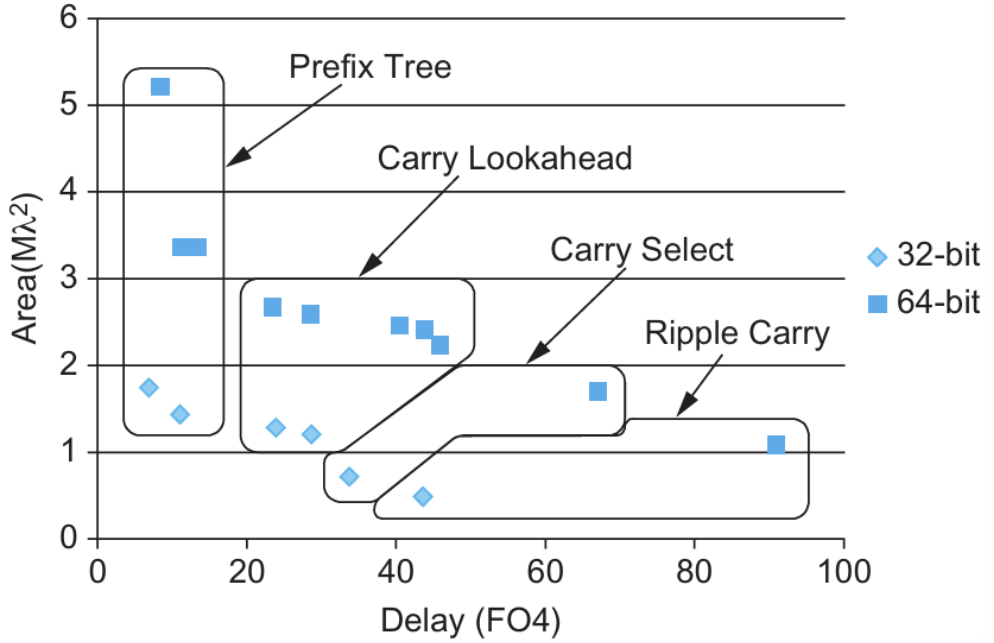
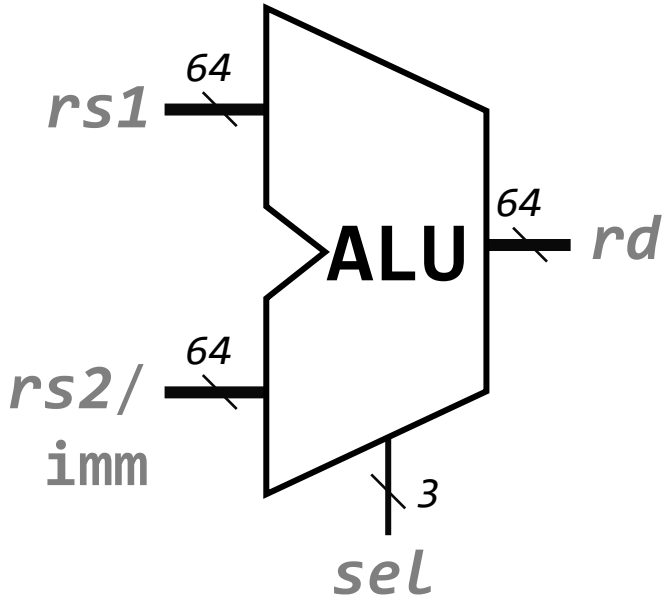
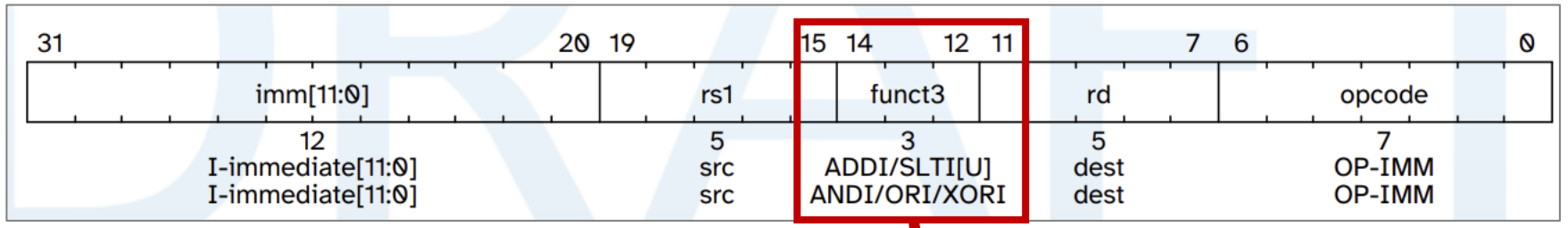


FIGURE 11.40 Area vs. delay of synthesized adders

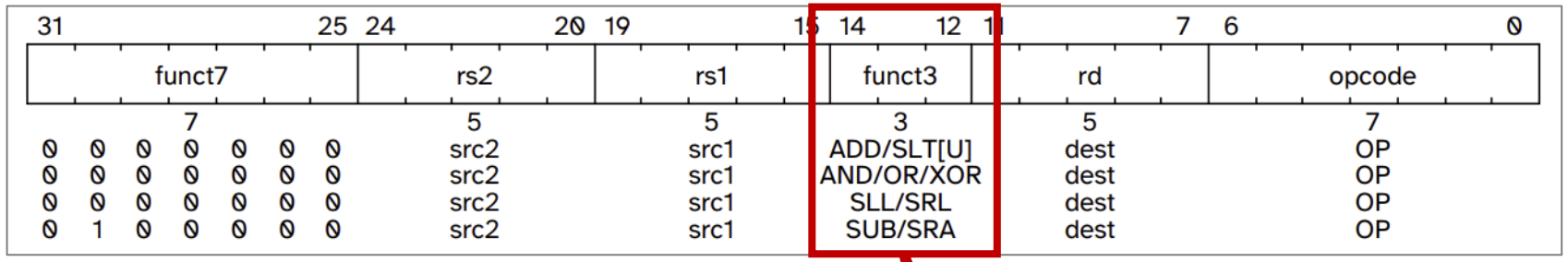
Our ALU (work in progress)



I-Type Encoding (Register-Immediate)



R-Type Encoding (Register-Register)




```
000 (and)
001 (or)
010 (xor)
011 (add)
100 (sub)
else (-)
```

Agenda

- Combinational Logic
  - Logical Operations
  - Addition/Subtraction
  - **Comparing Numbers**
  - Bit Shifting
- Building Registers

Comparing Two Numbers

```
slt(u)(i) rd, rs1, rs2/imm
```

```
slt rd, rs1, rs2
```

```
rd = { 0   (rs1 < rs2)
     { 1   (rs1 >= rs2)
```

```
b<cond> rs1, rs2, <label>
```

```
beq rs1, rs2, <label>
```

```
pc += { offset  (rs1 == rs2)
      { 4       (rs1 != rs2)
```

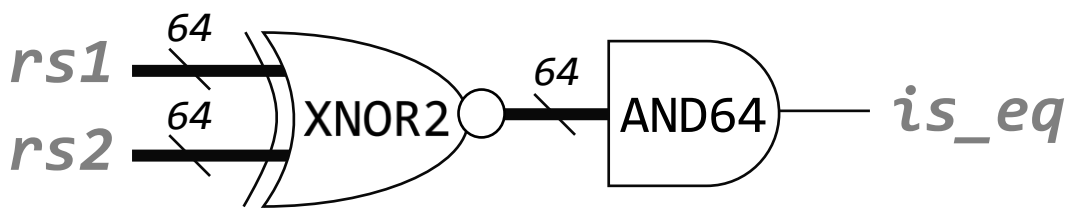
- Just use subtraction! **`X = rs1 – rs2`**
- Some gotcha's with signed/unsigned
  - May need to check sign bits, overflow bit (i.e., ALU carry out)
  - Some further reading

Faster Options Example: Checking Equality

```
beq rs1, rs2, <label>
```

```
pc += { offset  (rs1 == rs2)
      { 4       (rs1 != rs2)
```

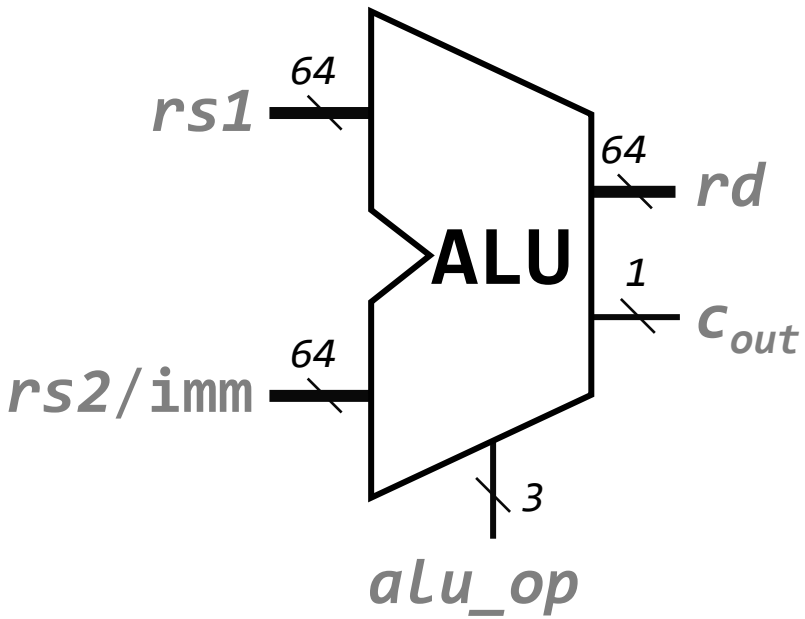
rs1	rs2	out
0	0	1
0	1	0
1	0	0
1	1	1



XNOR

ab	f
00	1
01	0
10	0
11	1

Extending the ALU: Work in Progress



alu_op

```
000 (and)
001 (or)
010 (xor)
011 (add)
100 (sub)
101 (slt)
110 (sltu)
else (-)
```

Logical Operations

```
and(i) rd, rs1, rs2/imm
or(i) rd, rs1, rs2/imm
xor(i) rd, rs1, rs2/imm
```

Add/Subtract

```
add(i)(w) rd, rs1, rs2/imm
sub(w) rd, rs1, rs2
```

Compare

```
slt(u)(i) rd, rs1, rs1/imm
```

Shift

```
sll(i)(w) rd, rs1, rs2/imm
srl(i)(w) rd, rs1, rs2/imm
sra(i)(w) rd, rs1, rs2/imm
```

Next

Agenda

- Combinational Logic
  - Logical Operations
  - Addition/Subtraction
  - Comparing Numbers
  - **Bit Shifting**

- Building Registers

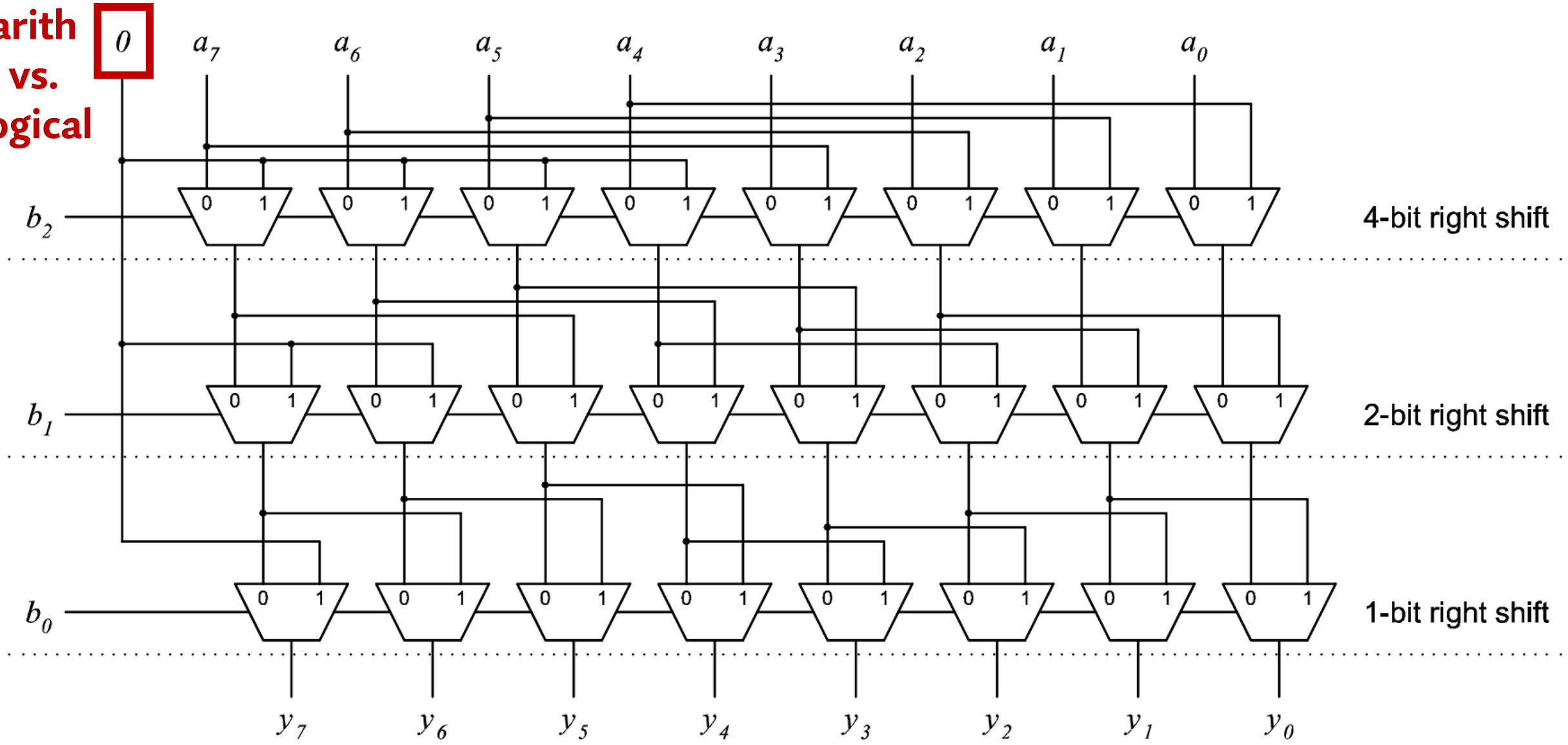
Hardware Bit Shifting

```
sll(i)(w) rd, rs1, rs2/imm // logical shift left
srl(i)(w) rd, rs1, rs2/imm // logical shift right
sra(i)(w) rd, rs1, rs2/imm // arith. shift right
```

- Efficient shifting is **surprisingly difficult**
  - In theory, any bit can be moved to any destination position
  - May need to sign/zero extend

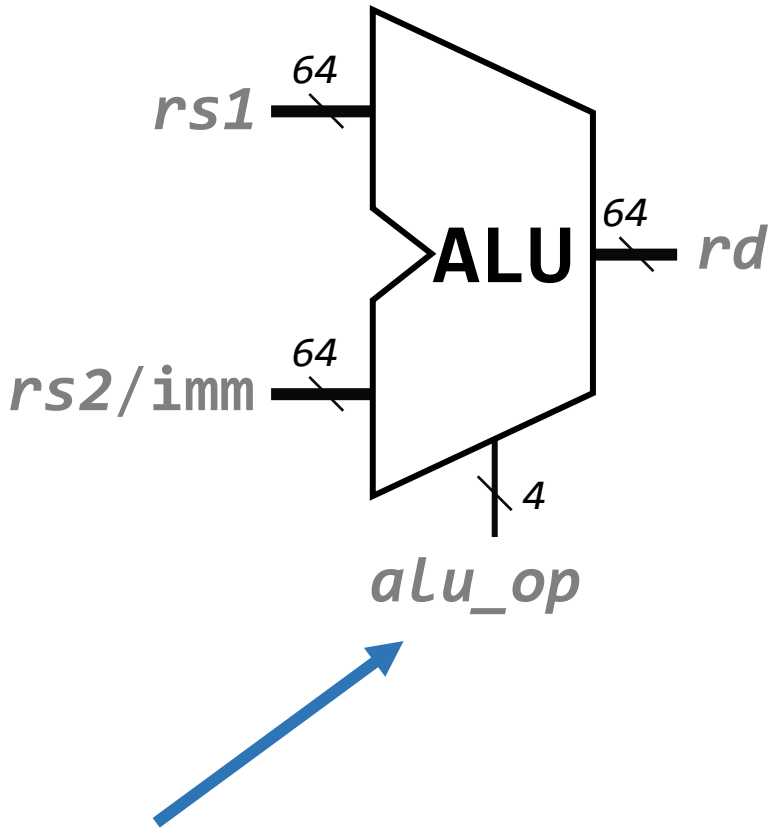
Example: 8-Bit Right Shift (Log(N) Complexity)

arith vs. logical



Pillmeier, "Design Alternatives for Barrel Shifters"

Our ALU



alu_op

```
0000 (and)
0001 (or)
0010 (xor)
0011 (add)
0100 (sub)
0101 (slt)
0110 (sltu)
0111 (sll)
1000 (srl)
1001 (sra)
else (-)
```

Note:
Actual mapping depends on implementation choices

Agenda

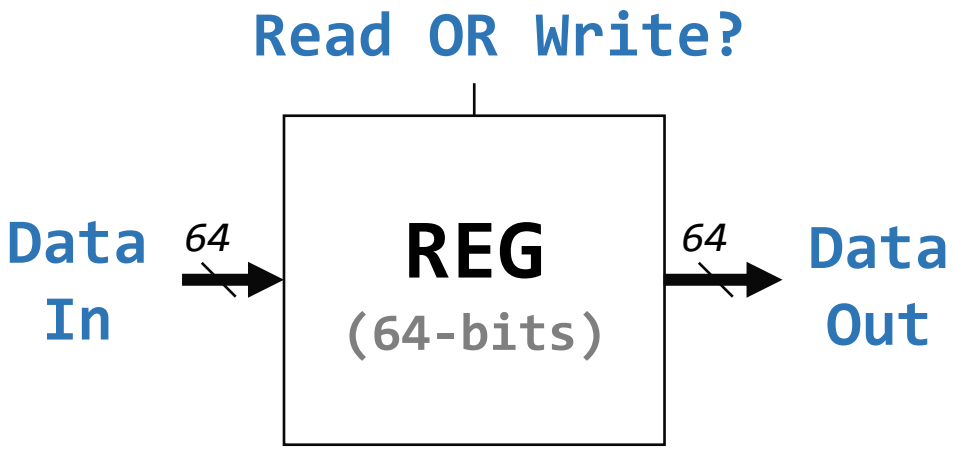
- Combinational Logic
  - Logical Operations
  - Addition/Subtraction
  - Comparing Numbers
  - Bit Shifting

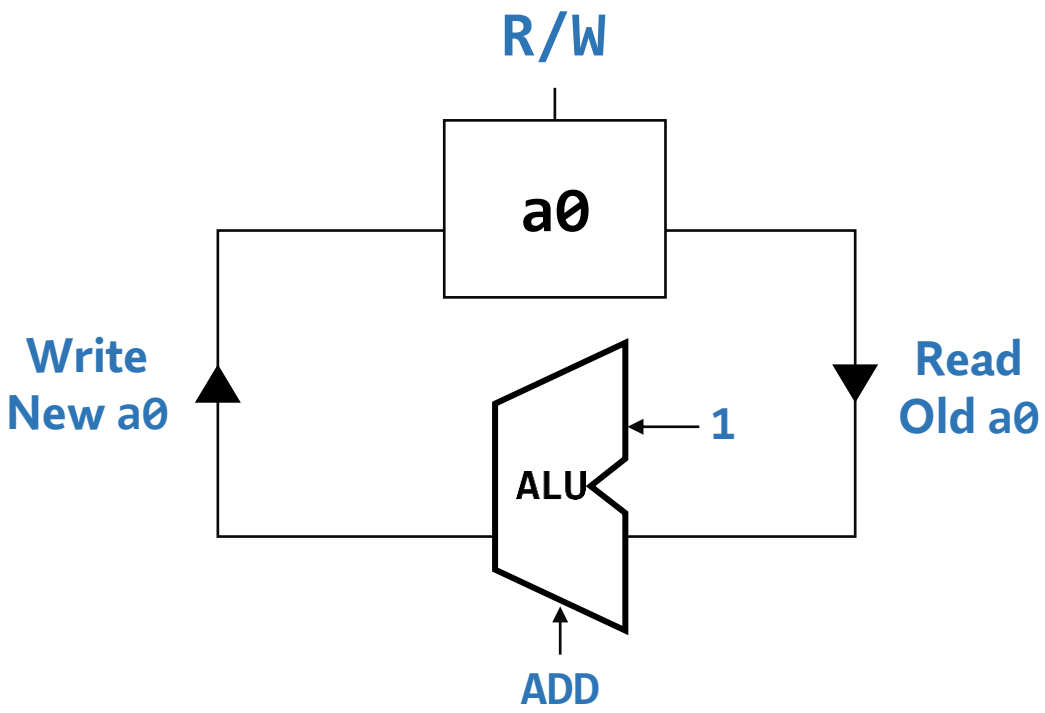
- **Building Registers**

Building Registers to Store Data

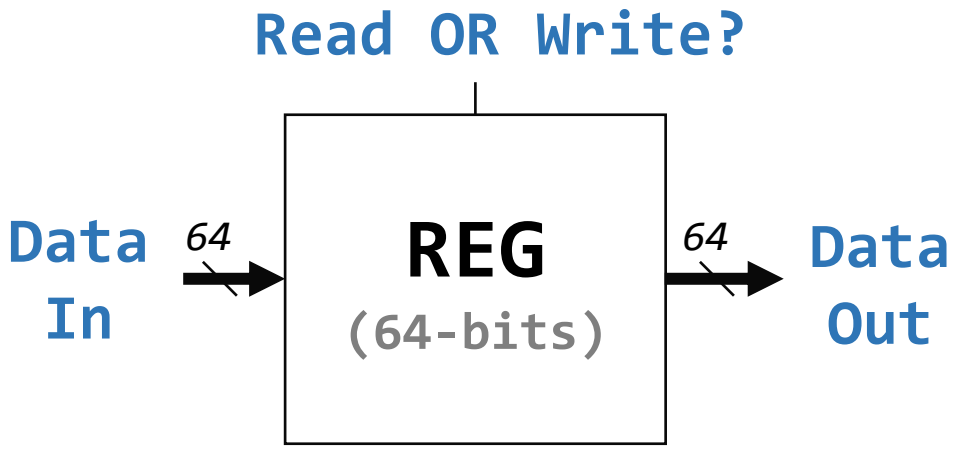
- Registers need to support both **reading and writing**

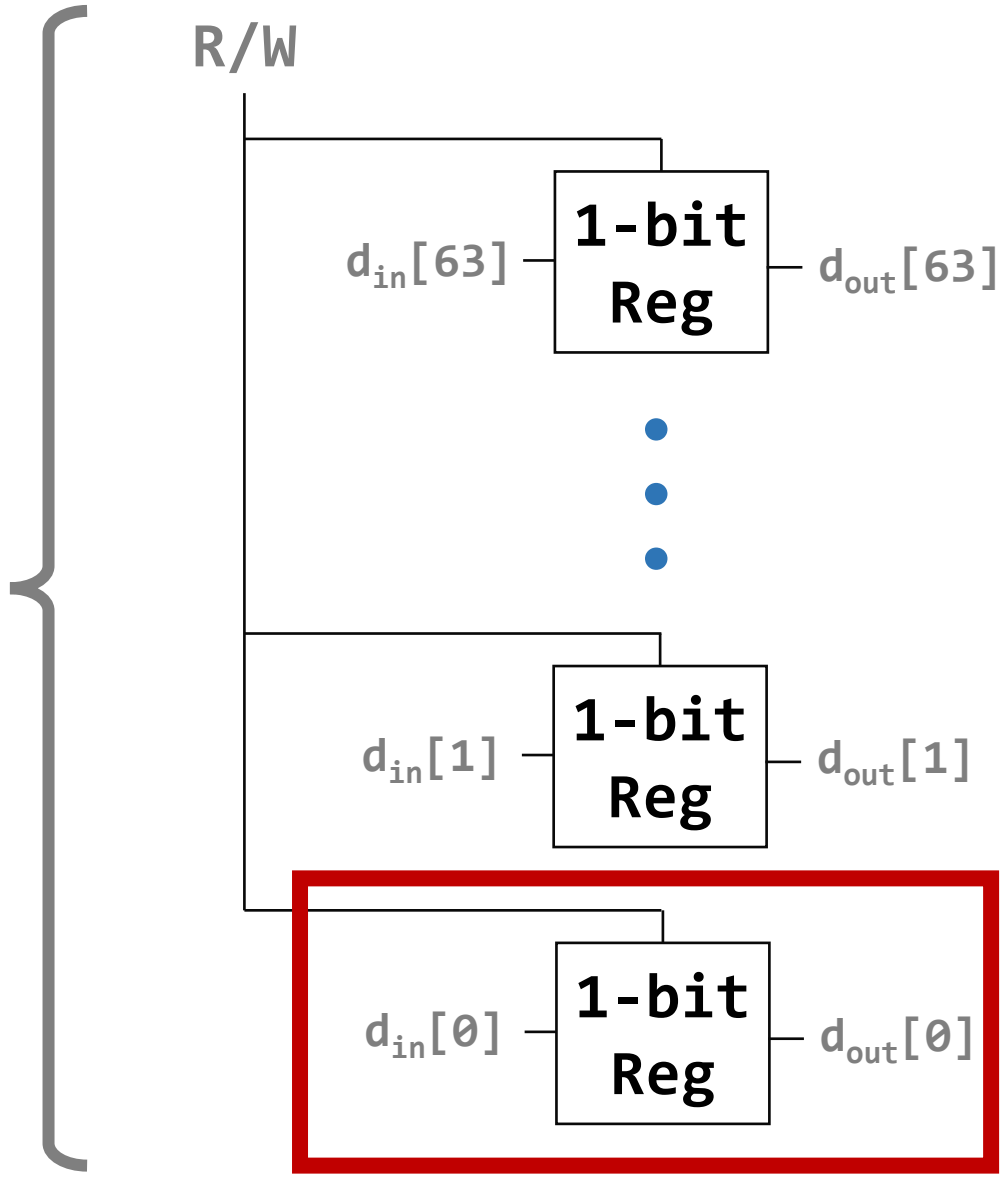
```
addi a0, a0, 1
```





Bit By Bit



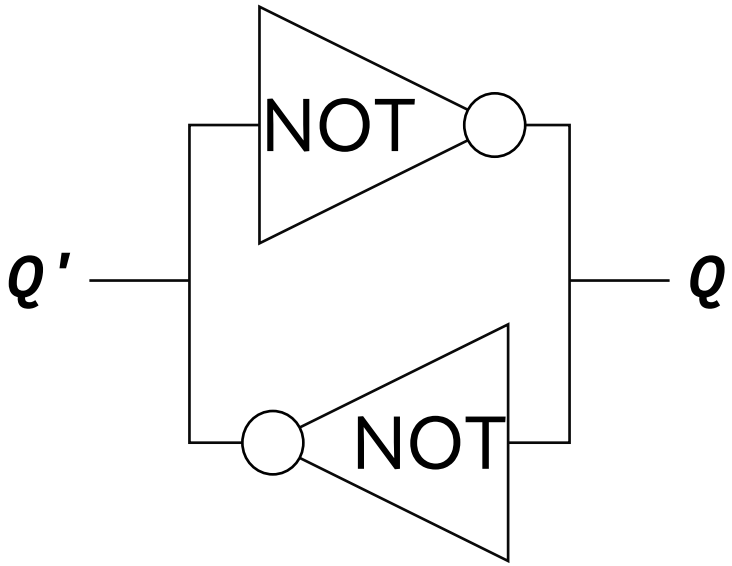


Let's build a 1-bit register

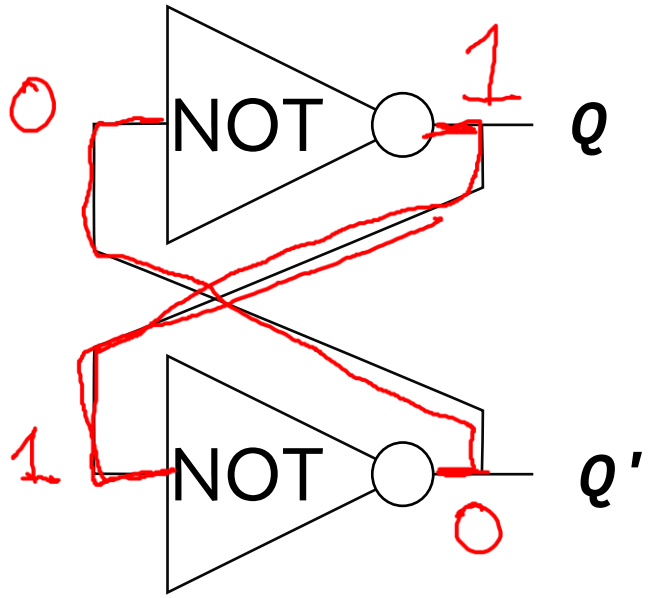
How Can We Store One Bit?

- Need a *bistable element*: anything with **two stable states**

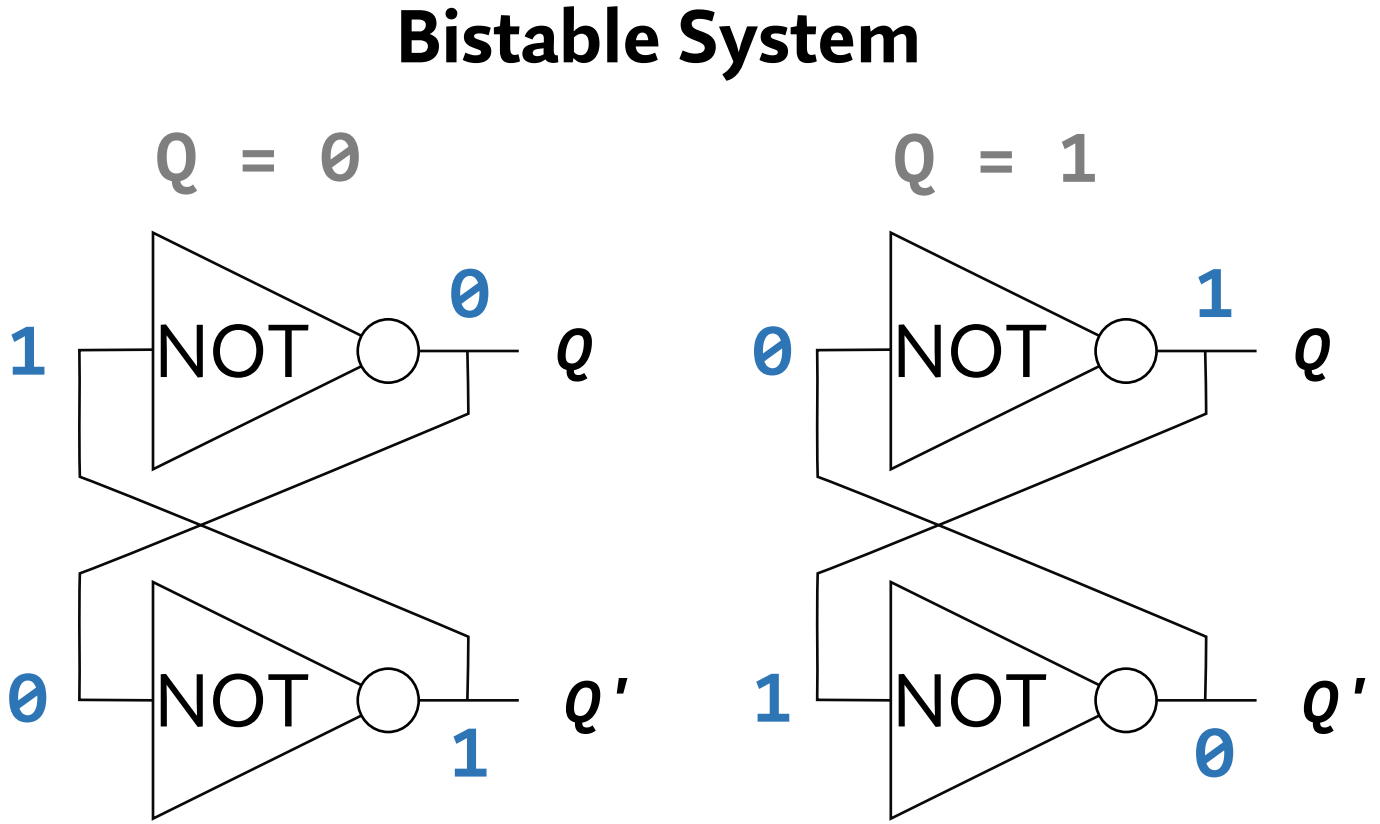
Cross-coupled Inverters



(Equivalent)



Cross-Coupled Inverters



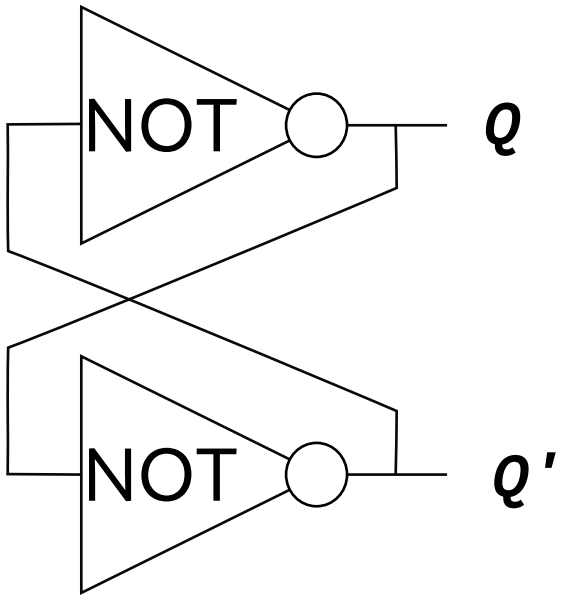
- Any starting state will resolve to either **Q={0,1}**

Set-Reset Latch

- Unfortunately, the user has **no inputs** to change the inverter state!

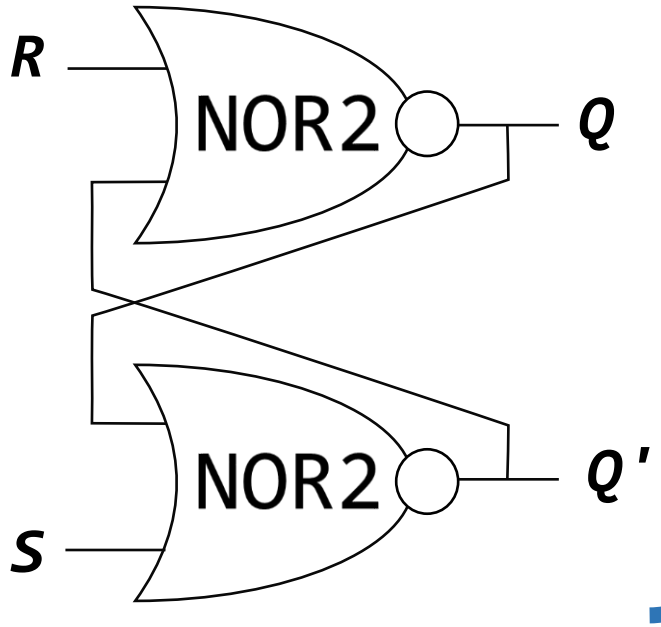
CC Inverters

(lacks input controls)



Set-Reset Latch

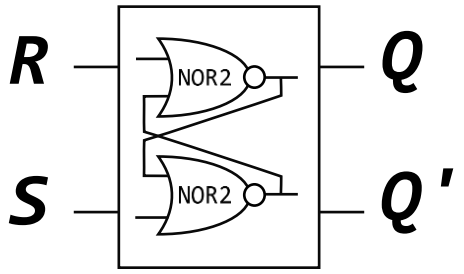
(has input controls)



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

We can control Q via R and S

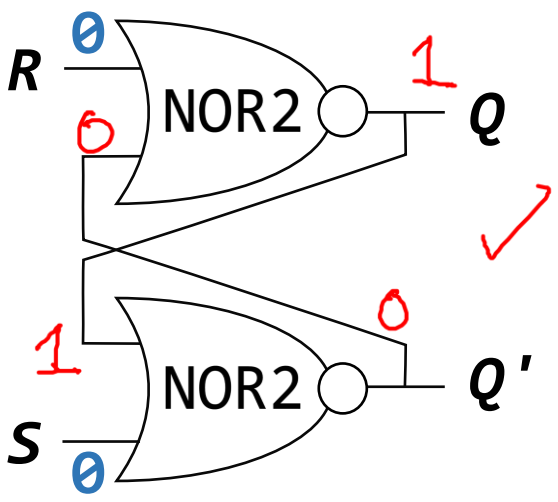
Analyzing SR Latches



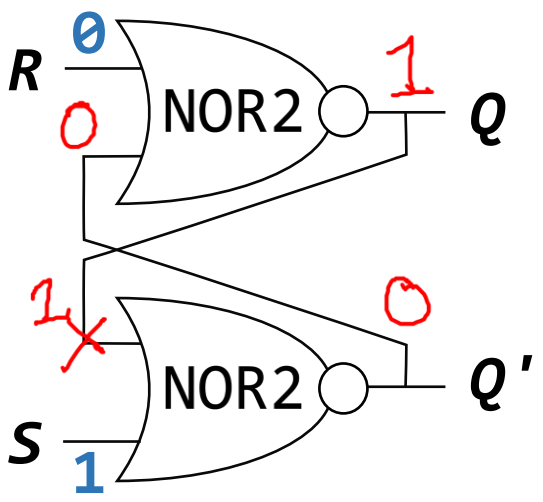
Truth Table

SR	Q
00	Q_{prev}
01	0
10	1
~~11~~	~~0~~

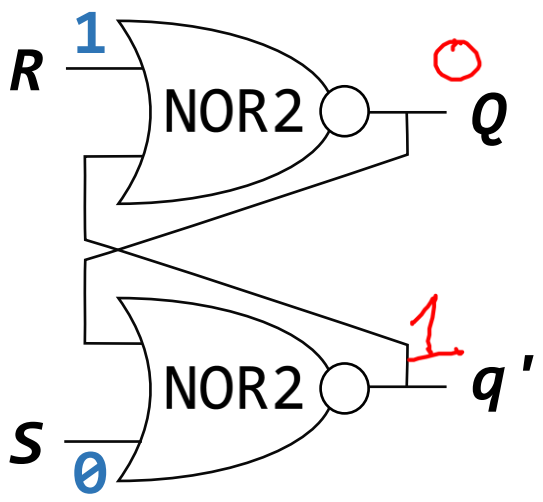
SR = 00

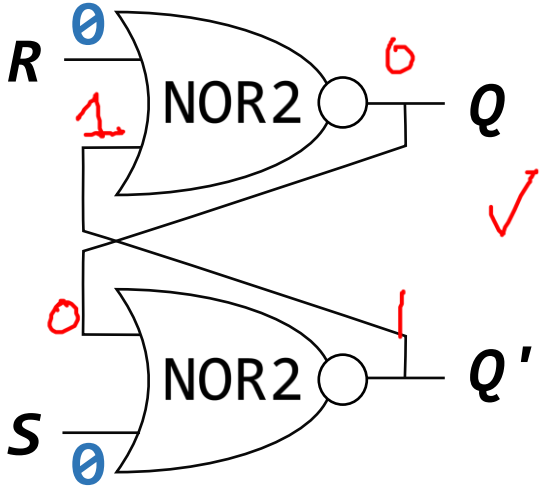


SR = 01

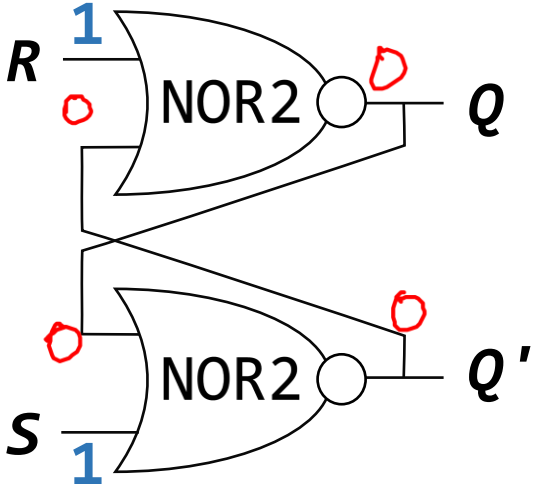


SR = 10

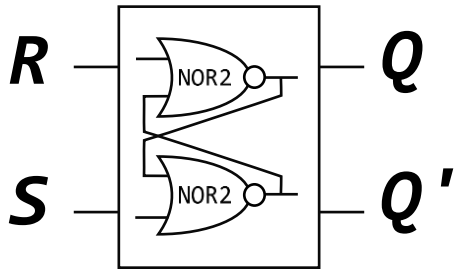




SR = 11



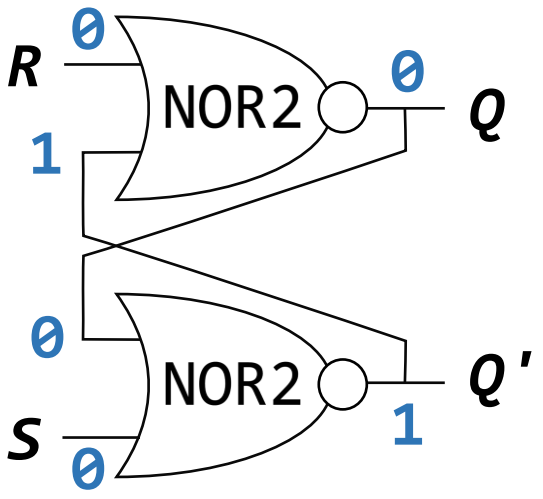
Analyzing SR Latches



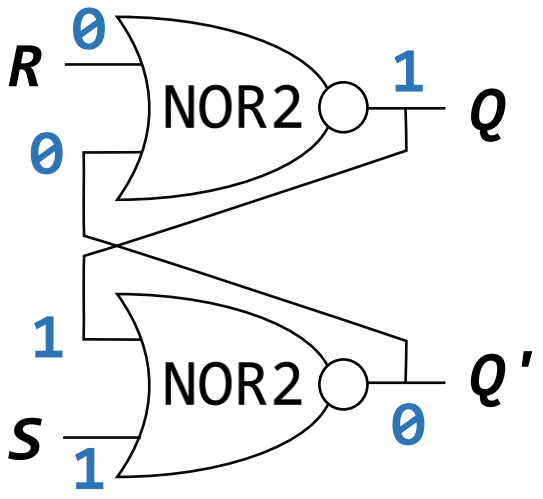
Truth Table

SR	Q
00	Q_{prev}
01	0
10	1
11	0

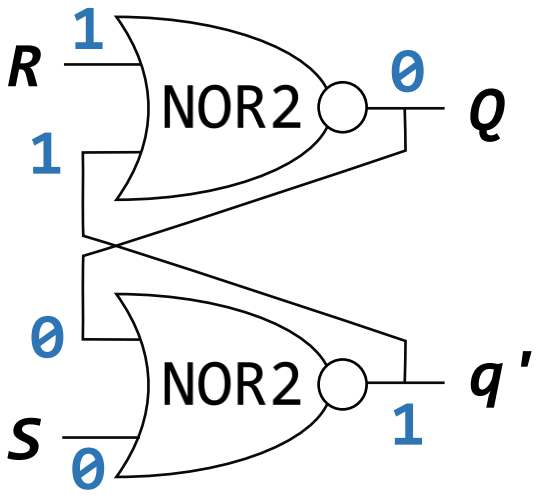
SR = 00



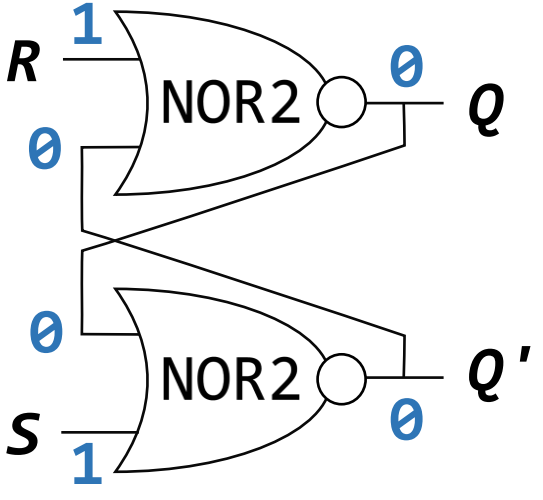
SR = 01



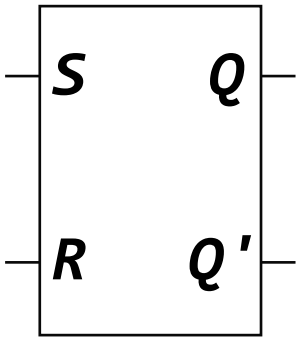
SR = 10



SR = 11



Abstracting the SR Latch



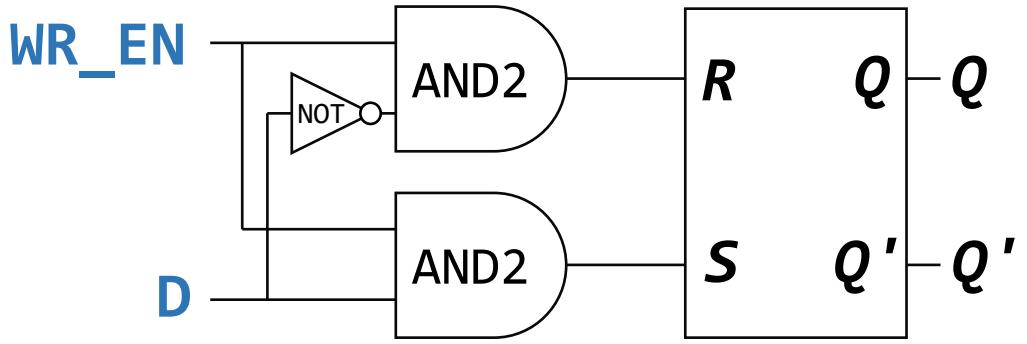
SR Latch

SR	Q	
00	Q_{prev}	"read mode"
01	0	"write mode"
10	1	
11	-	

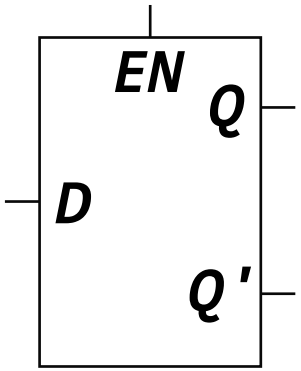
- We need a **better abstraction**
  - Distinguish between read/write mode
  - Get rid of the useless SR=11 case

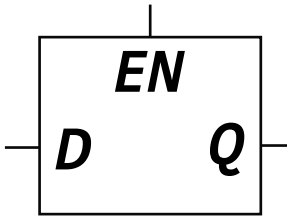
D Latch: An Abstracted SR Latch

- Can control when we update q based on a **write enable signal**



D Latch Symbols





D Latch

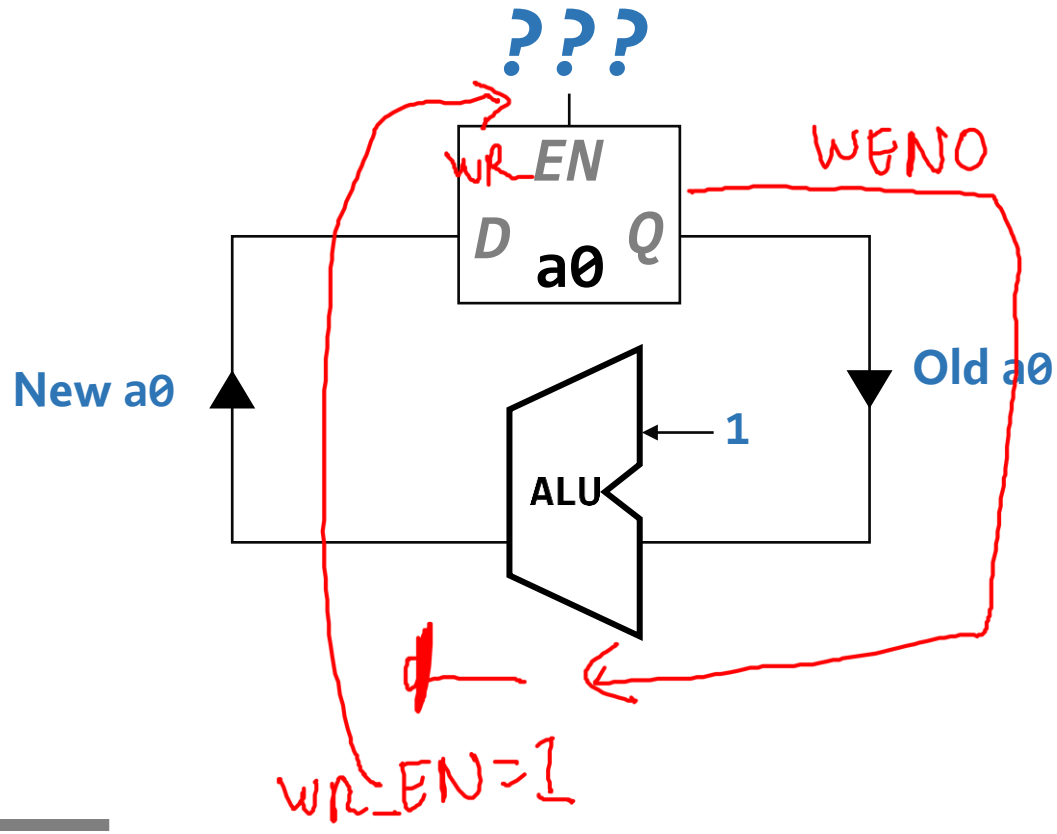
WR_EN	D	S	R	Q
0	X	0	0	Q_{prev}
1	d	d	d'	d

"opaque"

"transparent"

Using a D Latch

```
addi a0, a0, 1
```



D Latch

WR_EN	D	Q
0	X	Q_{prev}
1	d	d

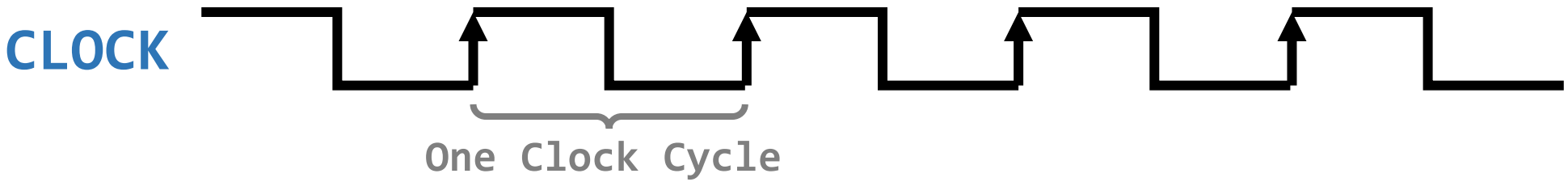
get the old **a0**

write the new **a0**

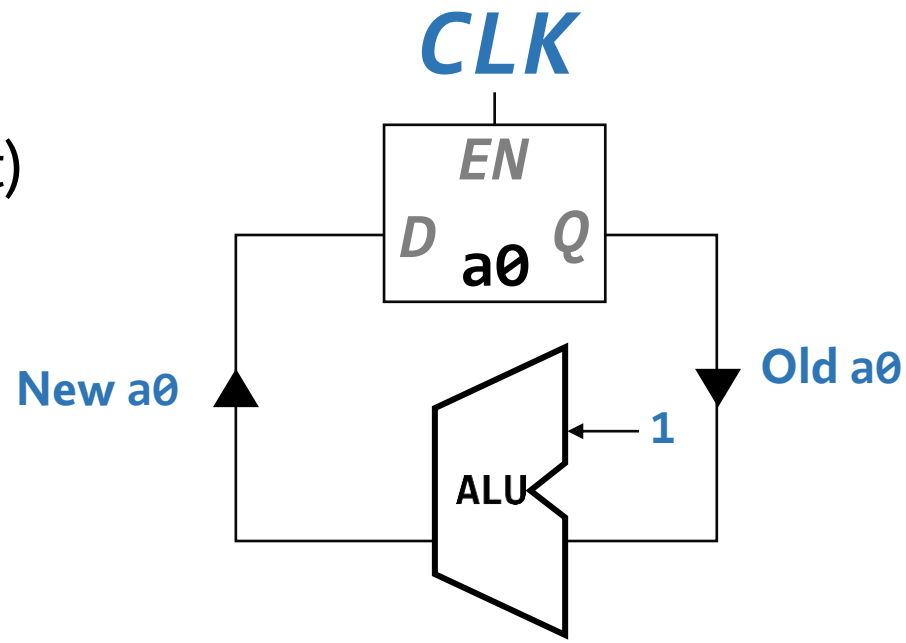
- Need to **carefully control** the WR_EN timing
- **Simpler solution:** use a **clock signal**

Clocked Latches

- **Clock Signal:** alternate `0/1` forever (very easy to build a circuit for this)



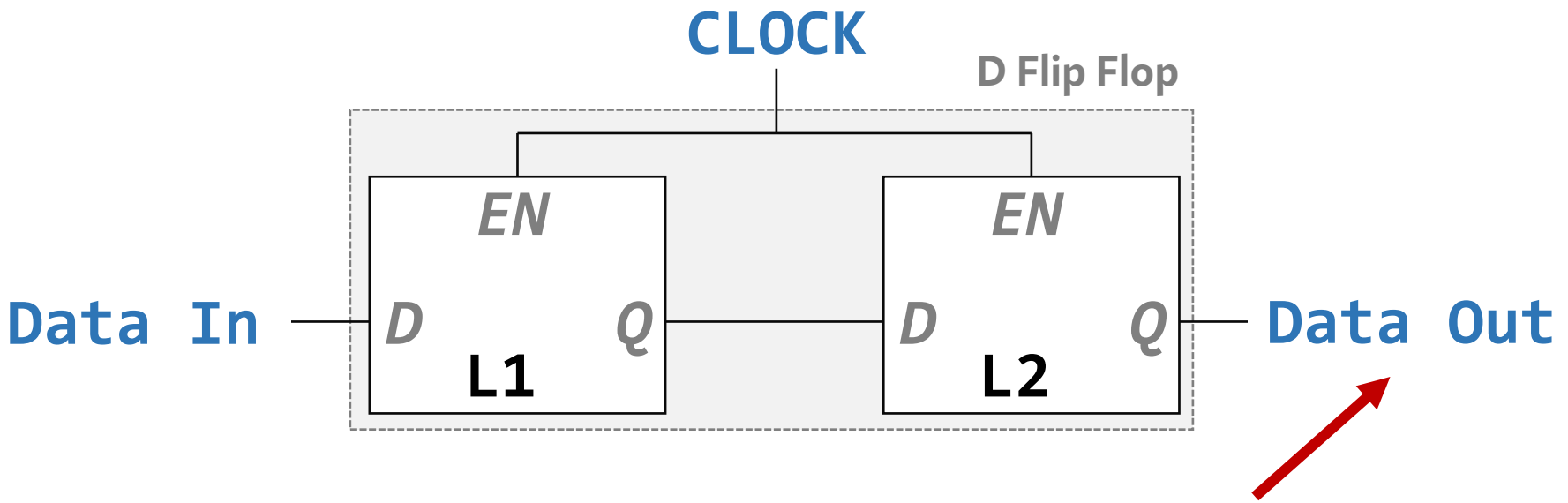
- **With D Latches:**
  - **Do calculations** when clock is `0` (latch is opaque)
  - **Update registers** when clock is 1 (latch is transparent)



- **Pros:** abstracts the rd/wr timing
- **Problem:** half of all time is wasted!

Final Version: Edge-Triggered D Flip-Flop

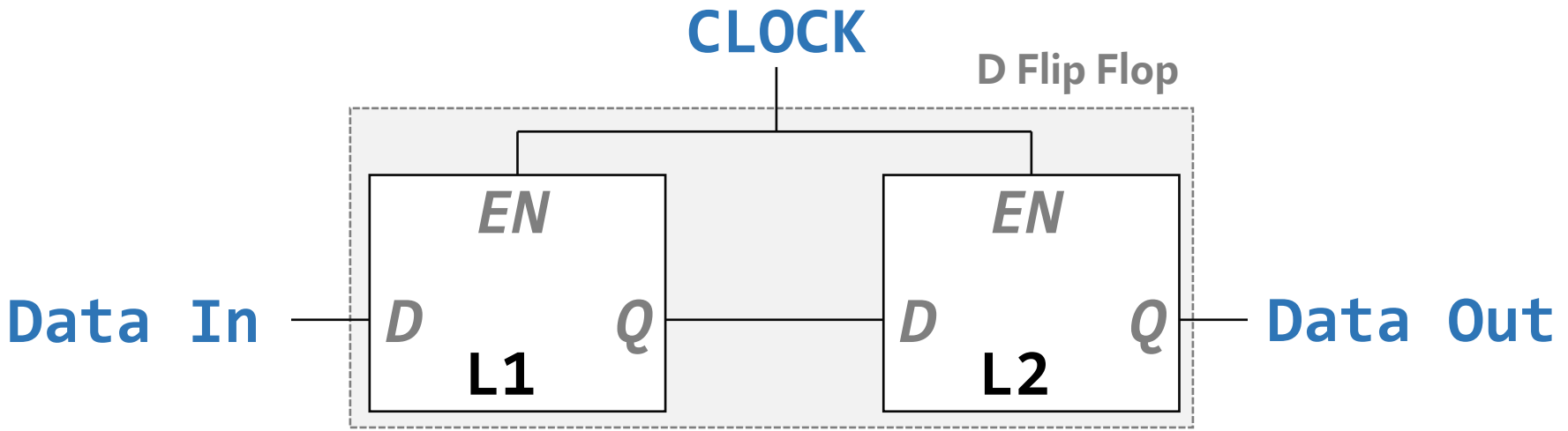
- **Key idea:** Update Q only on a **clock transition (`0 -> 1`)**

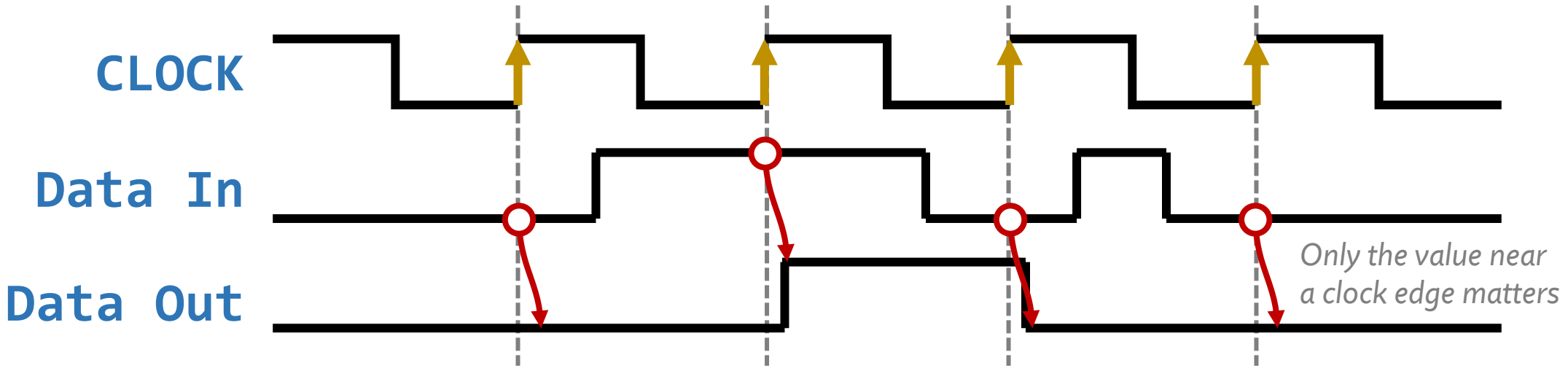


- Only updates on the clock's **rising edge**
- Rest of the time can be spent **computing**

Final Version: Edge-Triggered D Flip-Flop

- **Key idea:** Update Q only on a **clock transition (`0 -> 1`)**

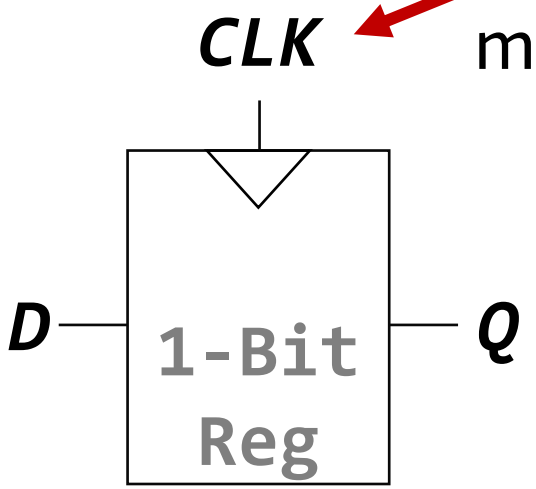


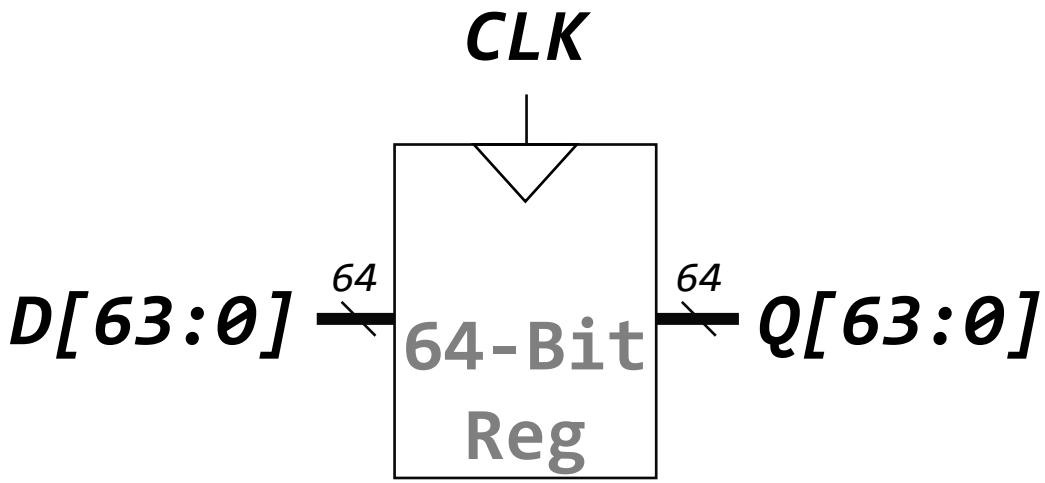


Edge-Triggered D Flip-Flop

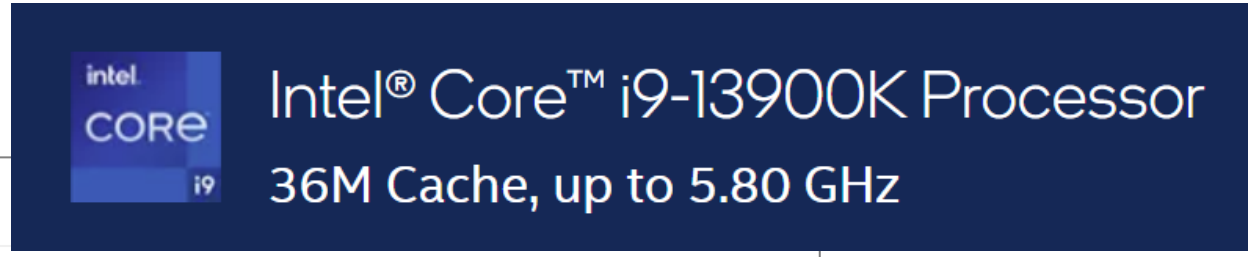
CPU **clock speed** determines how much work we can do per clock cycle

Sampled on every clock **rising edge**





Example: Intel i9



CPU Specifications	
Total Cores	24
# of Performance-cores	8
# of Efficient-cores	16
Total Threads	32
Max Turbo Frequency	5.80 GHz
Intel® Thermal Velocity Boost Frequency	5.80 GHz
Intel® Turbo Boost Max Technology 3.0 Frequency ‡	5.70 GHz
Performance-core Max Turbo Frequency	5.40 GHz
Efficient-core Max Turbo Frequency	4.30 GHz
Performance-core Base Frequency	3.00 GHz
Efficient-core Base Frequency	2.20 GHz

https://www.intel.com/content/www/us/en/products/sku/230496/intel-core-i913900k-processor-36m-cache-up-to-5-80-ghz/specifications.html

CS 211: Intro to Computer Architecture

11.2: Combinational Logic, Flip-Flops, and Clocks

Minesh Patel

Spring 2025 – Thursday 10 April