Digital Logic - Sequential

Roadmap

C:

```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
    c.getMPG();
```

Assembly language:

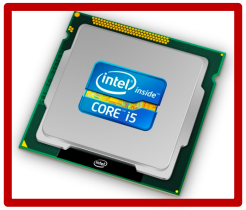
```
get_mpg(car*):
        lw      a5,0(a0)
        lw      a4,4(a0)
        divw    a5,a5,a4
        fcvt.s.w        fa0,a5
        ret
```

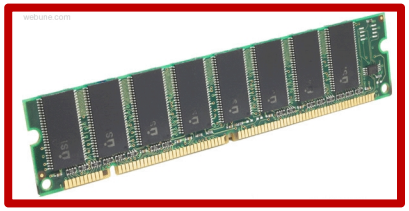
Machine code:

```
01110100000110000
10001101000001000000010
1000100111000010
11000001111110100011111
```

OS:

Computer system:





- Memory & data
- Arrays & structs
- Integers & floats
- RISC V assembly
- Procedures & stacks
- Executables
- Memory & caches
- **Processor Pipeline**
- Performance
- Parallelism

Agenda

- Muxes
- **Sequential Logic Timing**
- Maximum Clock Frequency
- Finite State Machines
- Functional Units
- Summary

Bonus Slides

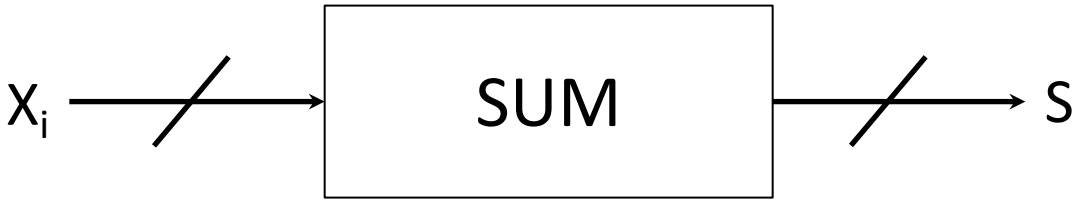
- Logisim Intro

Type of Circuits

- *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
    - e.g. circuits to add A, B (ALUs)
  - Sequential Logic (SL)
    - Circuits that "remember" or store information
    - a.k.a. "State Elements"
    - e.g. memory and registers (Registers)

Accumulator Example

An example of why we would need sequential logic



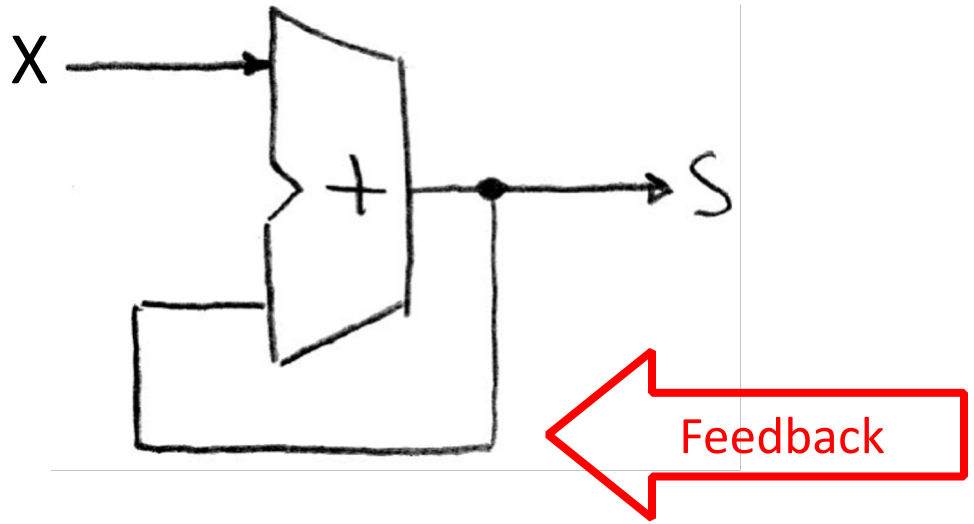
Want:

```
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`

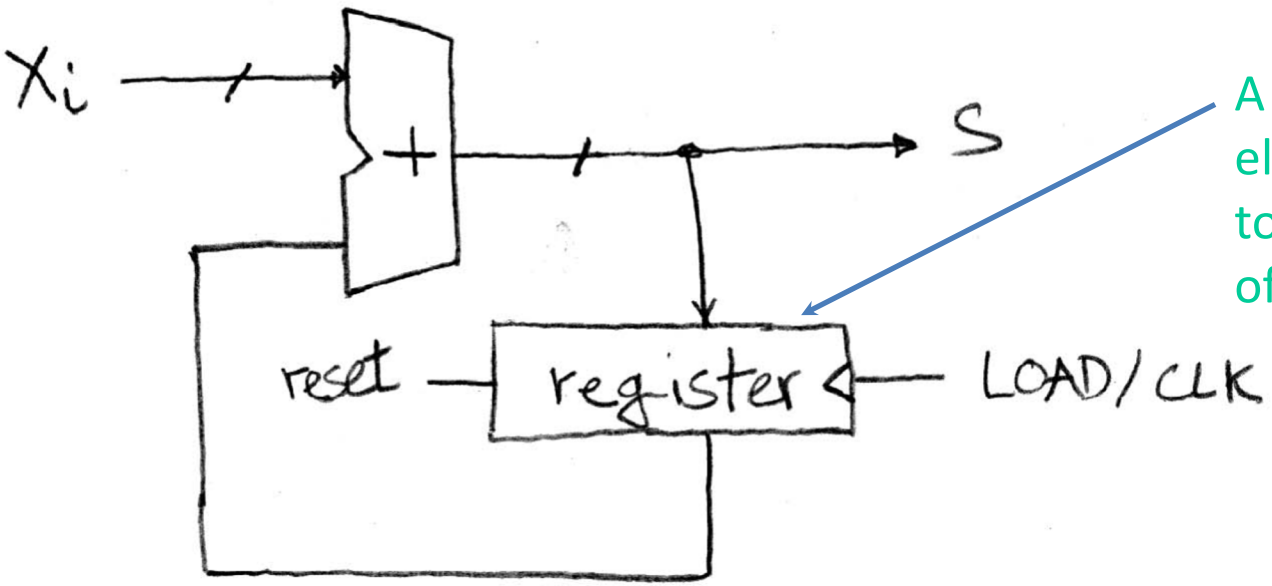
First Try: Does this work?



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

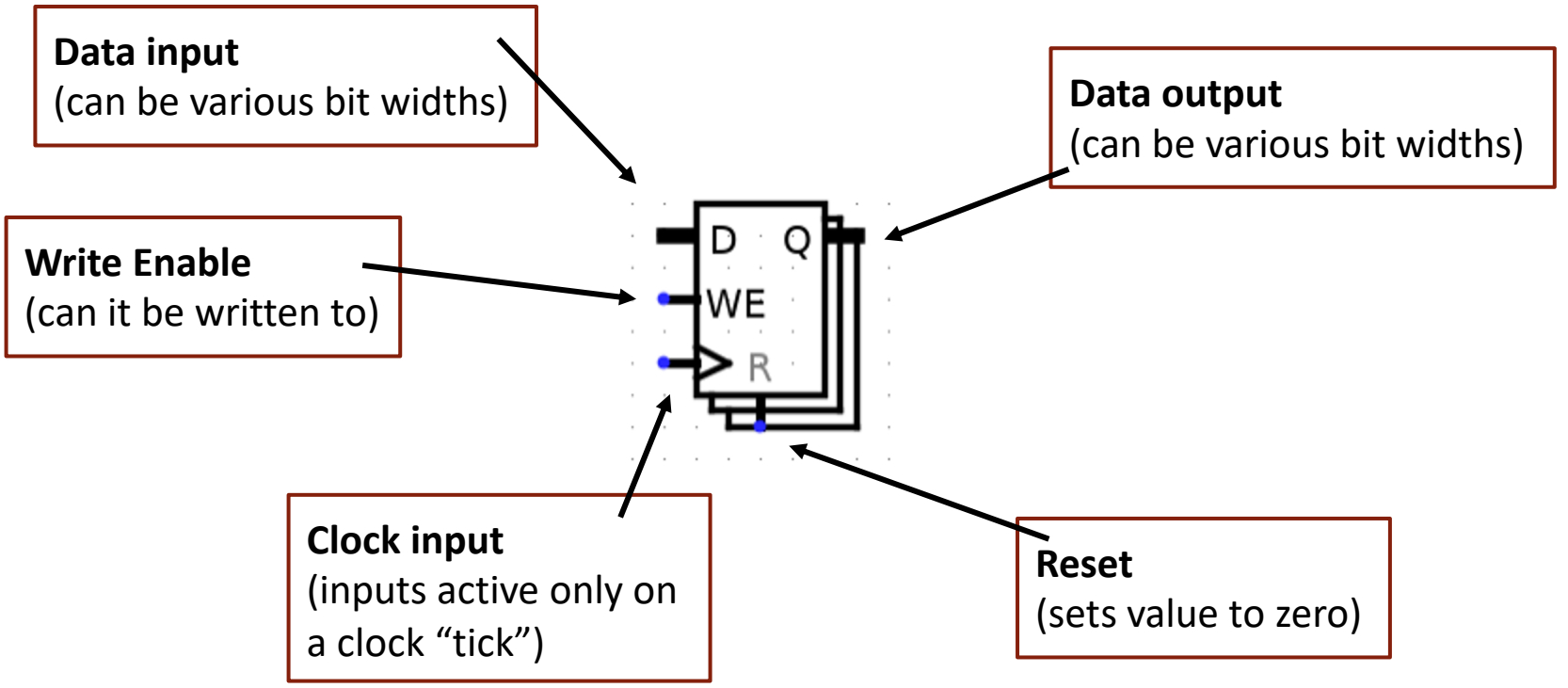
Uses for State Elements

- 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

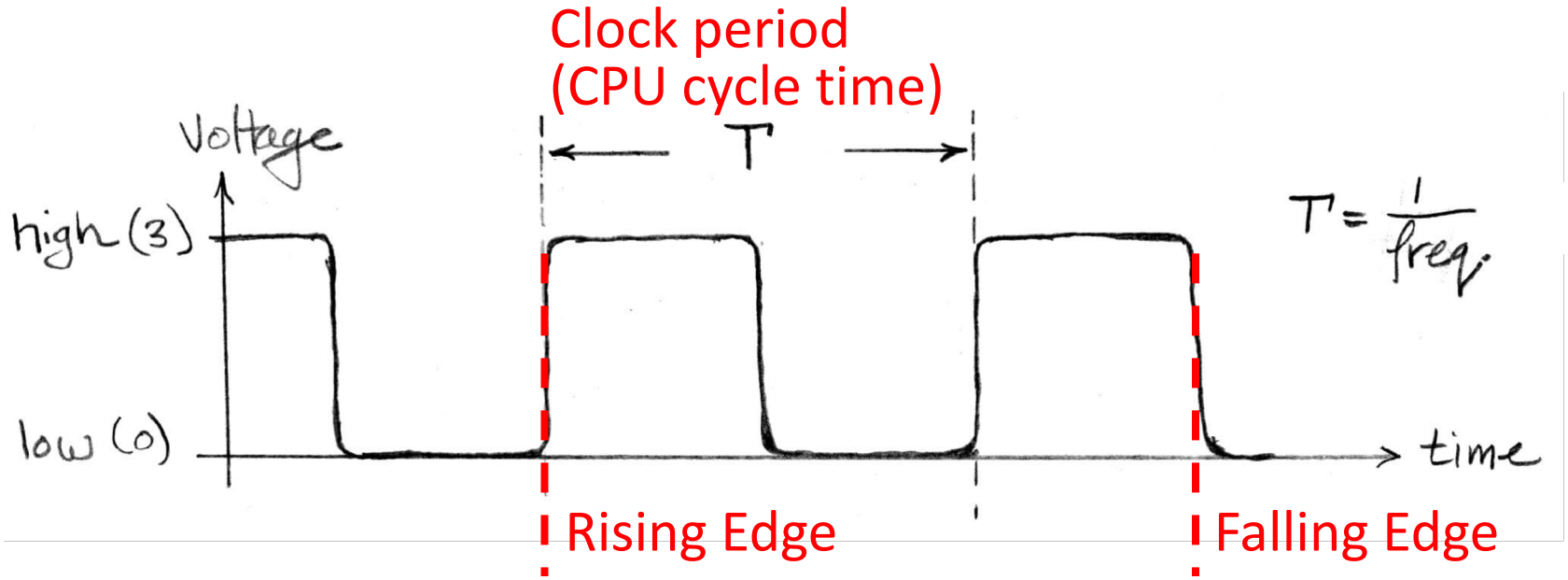
Registers

Same as registers in assembly:

- small memory storage locations

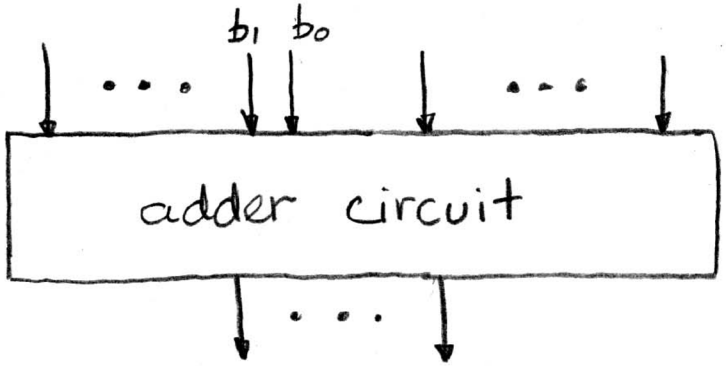


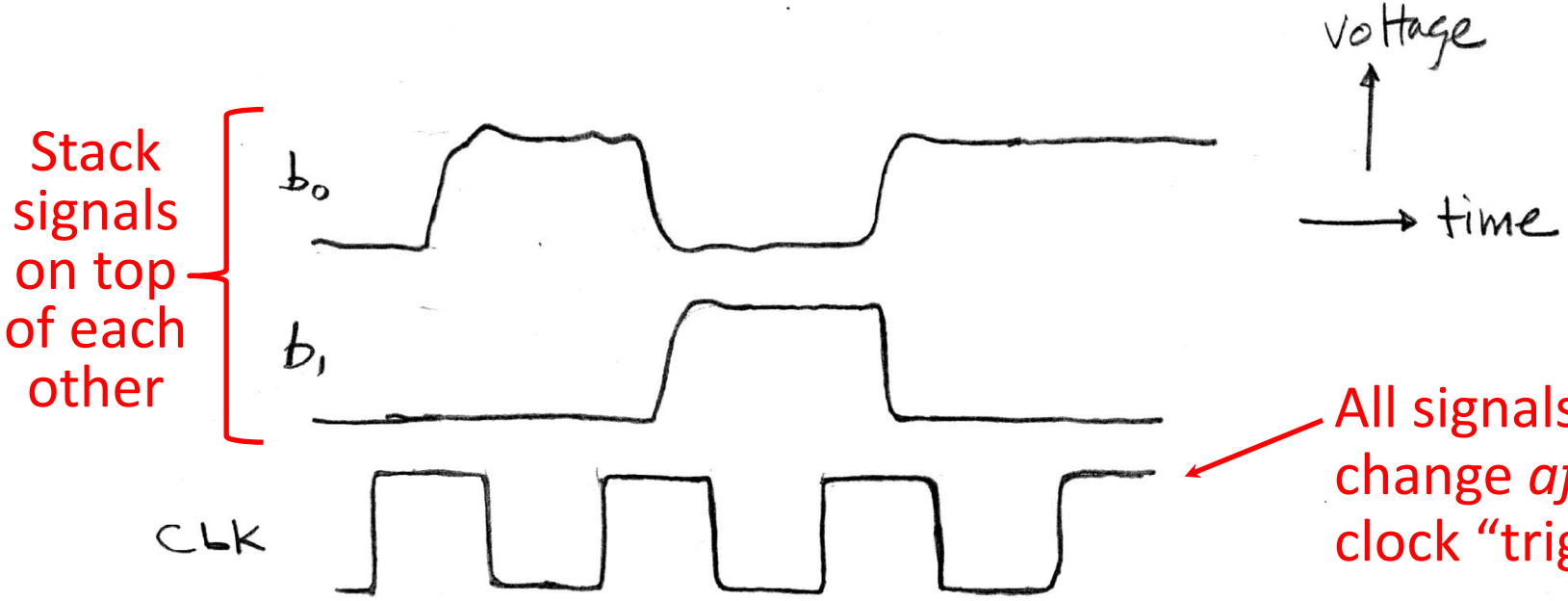
Signals and Waveforms: Clocks



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

Signals and Waveforms





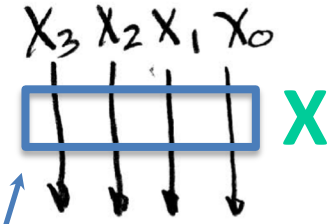
Stack signals on top of each other

All signals change *after* clock "triggers"

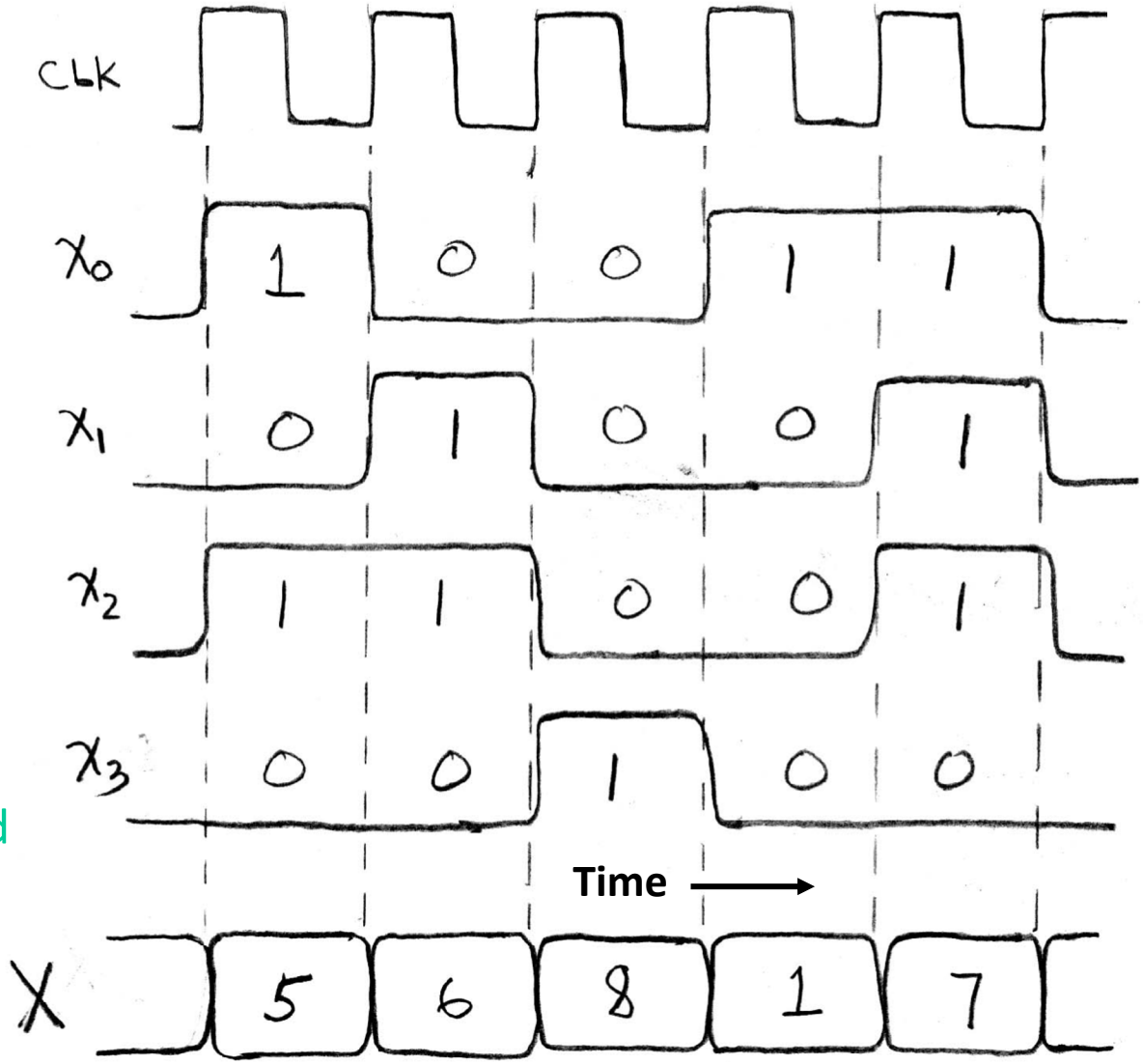
Dealing with Waveform Diagrams

- 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

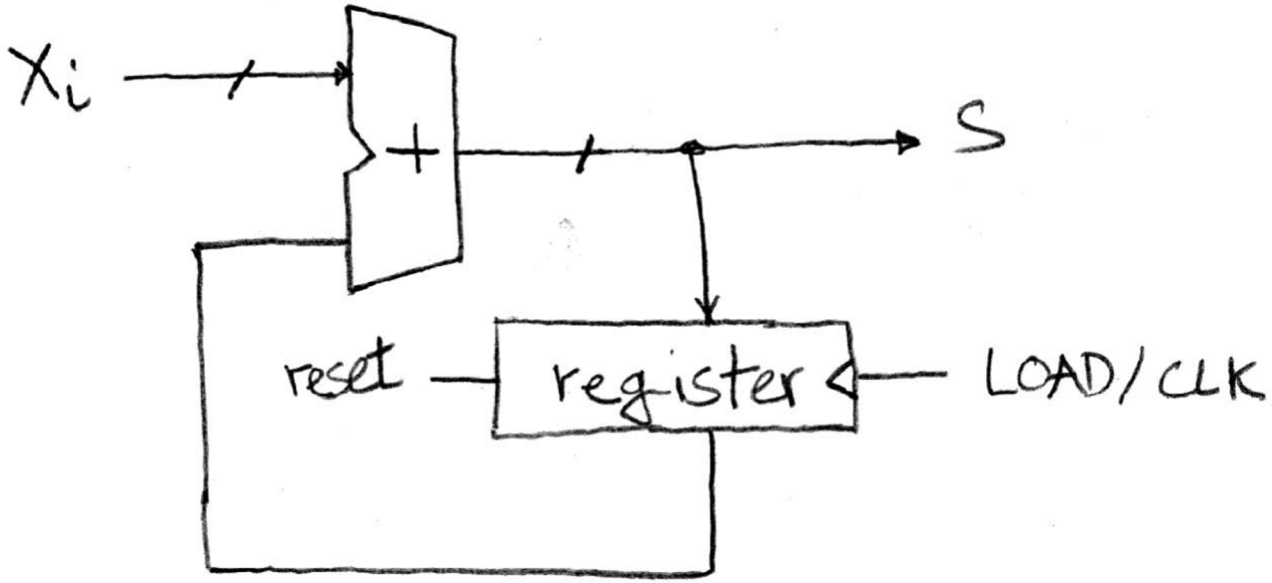
Signals and Waveforms: Grouping



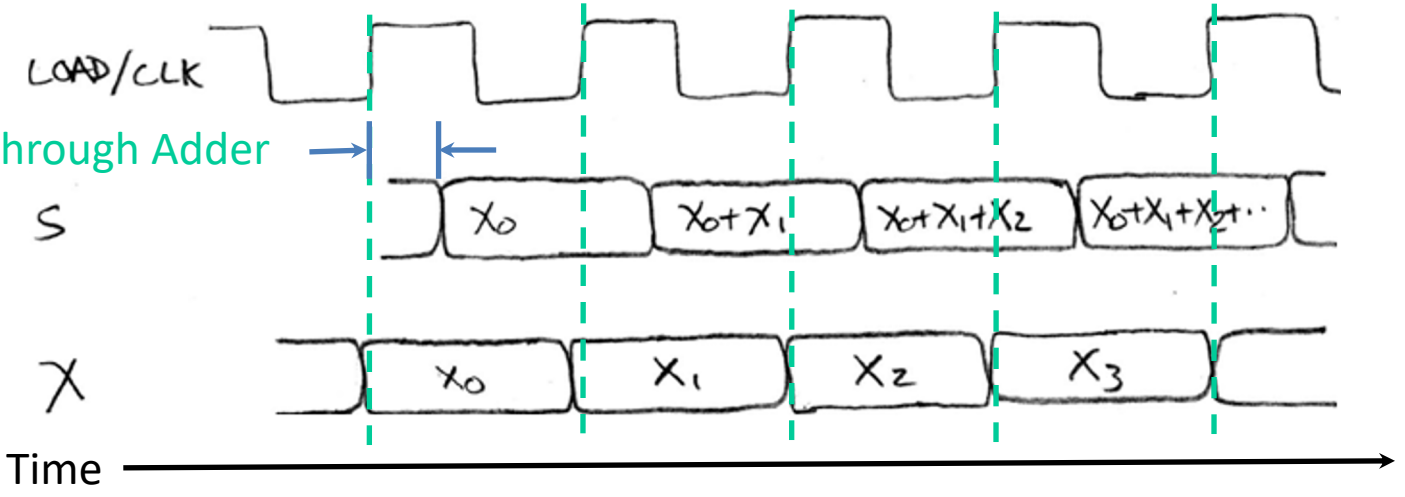
A group of wires when interpreted as a bit field is called a *bus*



Second Try: How About This?



Rough timing …



Review of 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

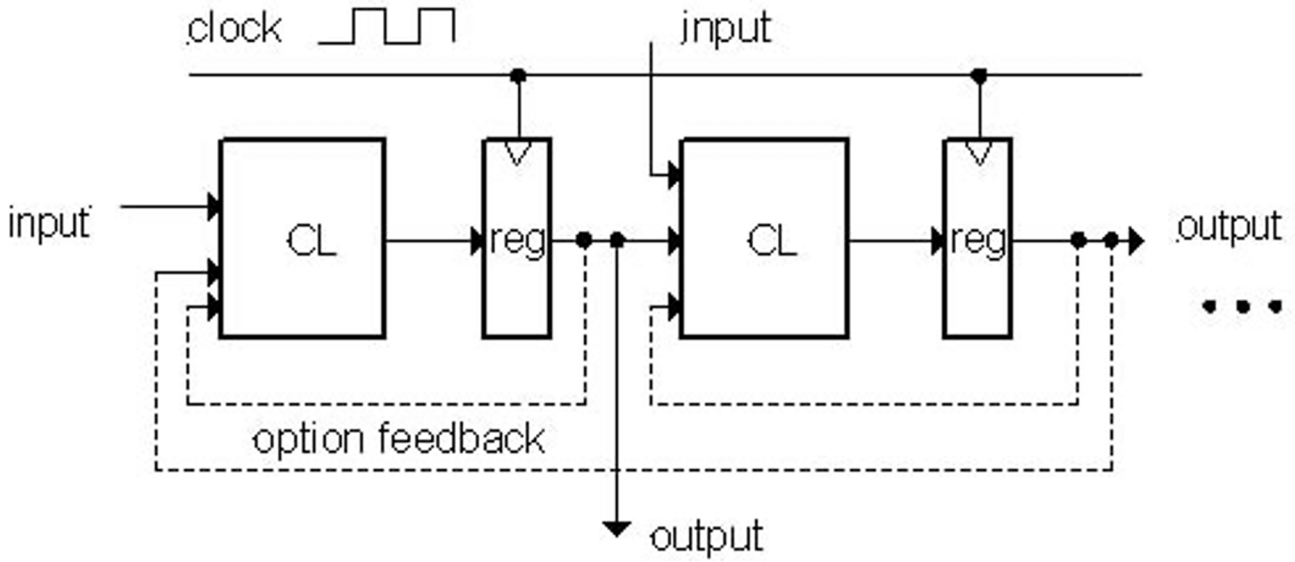
Agenda

- Muxes
- Sequential Logic Timing
- **Maximum Clock Frequency**
- Finite State Machines
- Functional Units
- Summary

Bonus Slides

- Logisim Intro

Model for Synchronous Systems

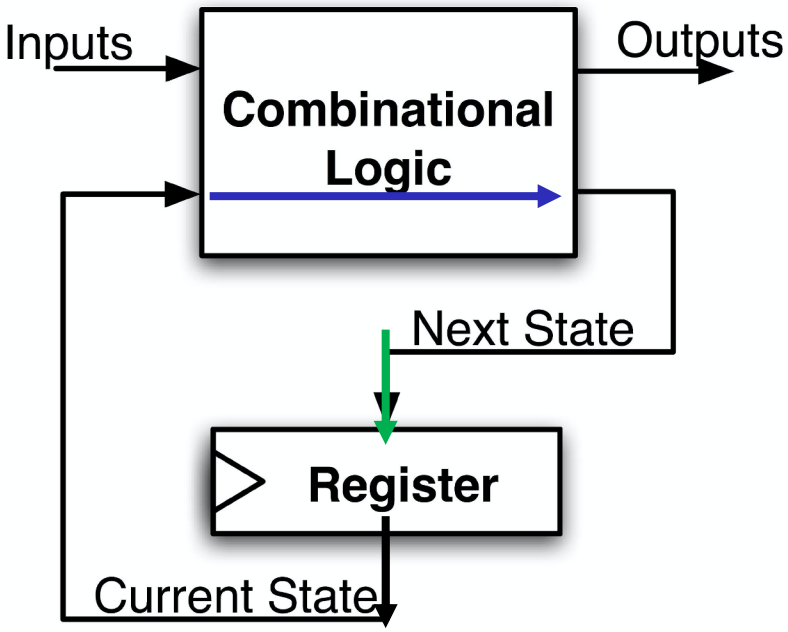


- Combinational logic blocks separated by registers
  - Clock signal connects only to sequential logic elements
  - Feedback is optional depending on application
- How do we ensure proper behavior?
  - How fast can we run our clock?

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



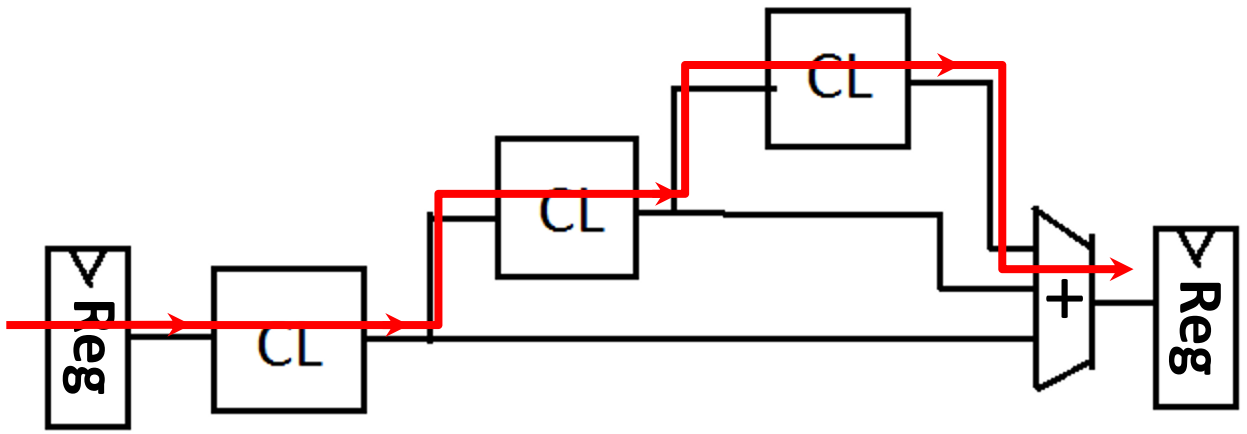
Max Delay = CLK-to-Q Delay
+ CL Delay
+ Setup Time

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



Critical Path =

CLK-to-Q Delay

+ CL Delay 1

+ CL Delay 2

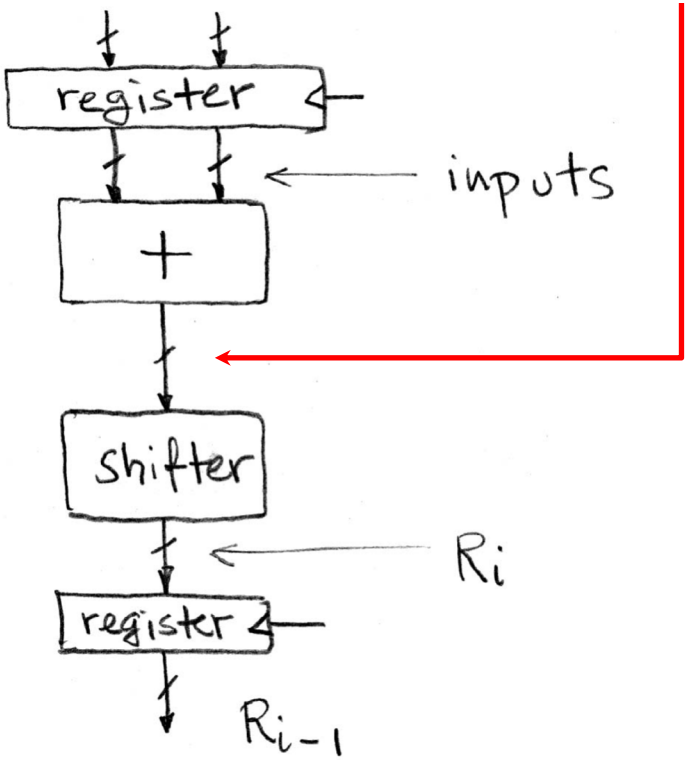
How do we go faster?

Pipelining!

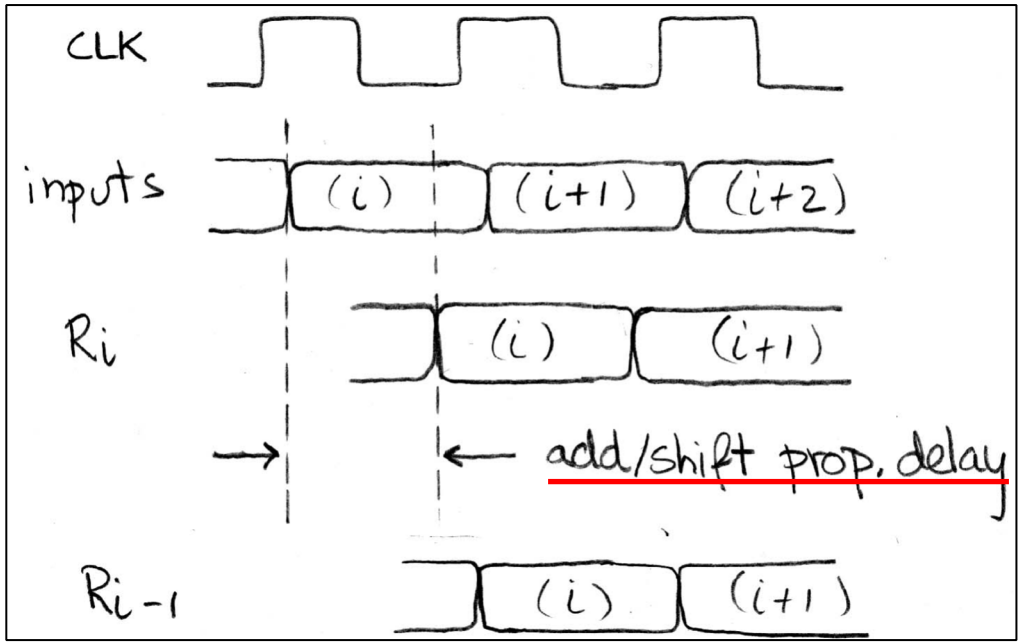
- Split operation into smaller parts and add a register between each one.

Pipelining and Clock Frequency (1/2)

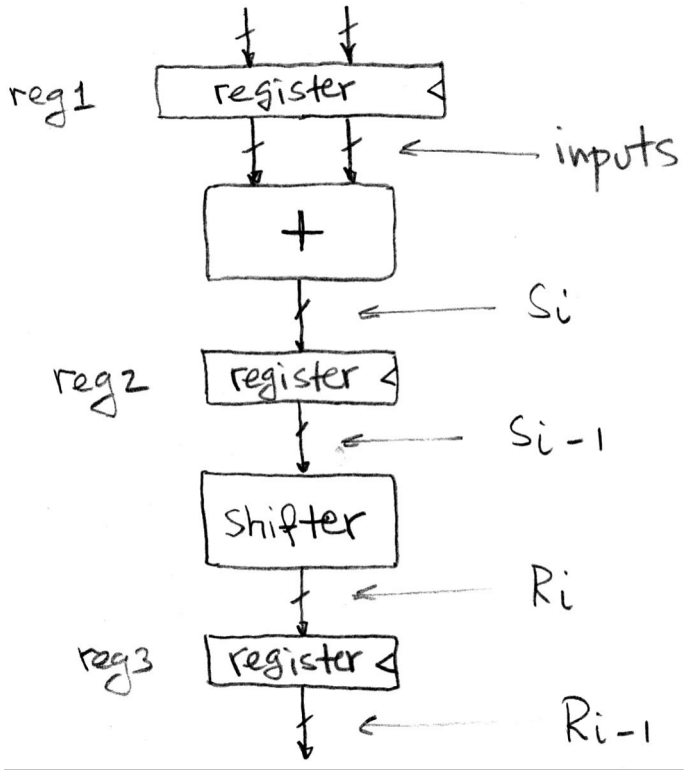
- Clock period limited by propagation delay of adder and shifter
  - Add an extra register to reduce the critical path!

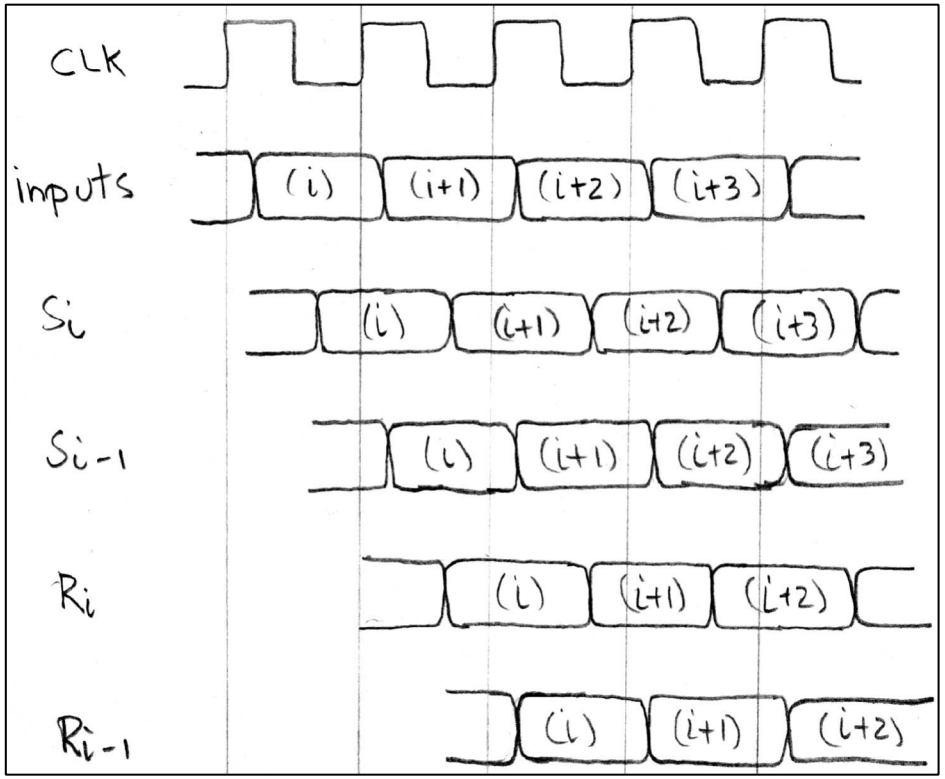


Timing:



Pipelining and Clock Frequency (2/2)





- Reduced critical path → allows higher clock freq.
- Extra register → extra (shorter) cycle to produce first output

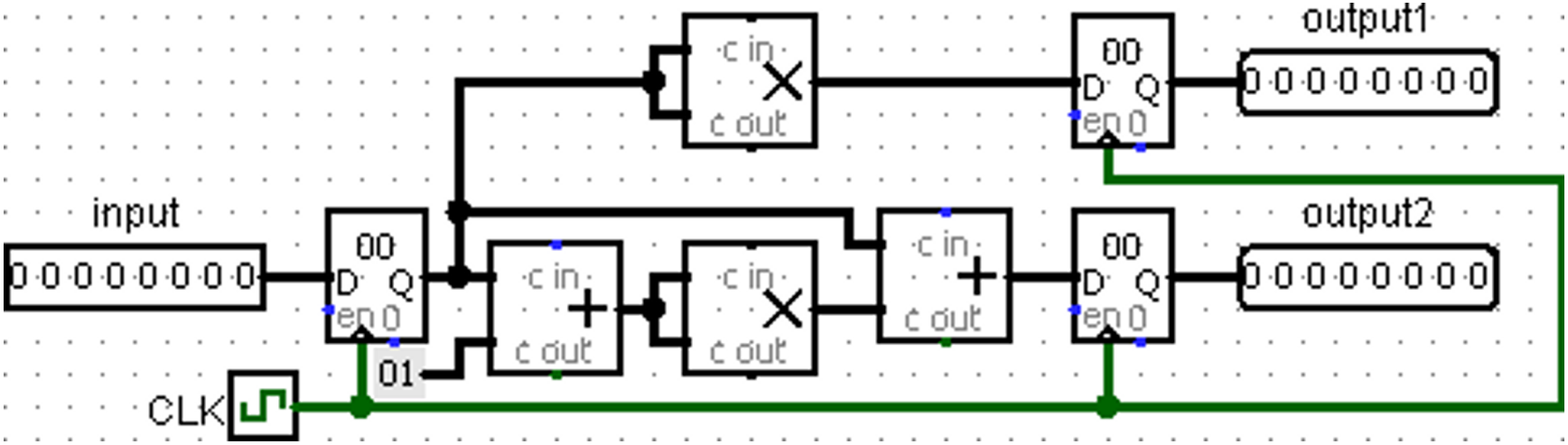
Pipelining Basics

- By adding more registers, break path into shorter "stages"
  - Aim is to reduce critical path
  - Signals take an additional clock cycle to propagate through *each* stage
- New critical path must be calculated
  - Affected by placement of new pipelining registers
  - Faster clock rate → higher throughput (outputs)
  - More stages → higher startup latency
- Pipelining tends to improve performance
  - More on this (and application to CPUs) later

Question: Want to run on 1 GHz processor.

t_{add} = 100 ps. t_{mult} = 200 ps. t_{setup} = t_{hold} = 50 ps.

What is the maximum clock-to-q time?

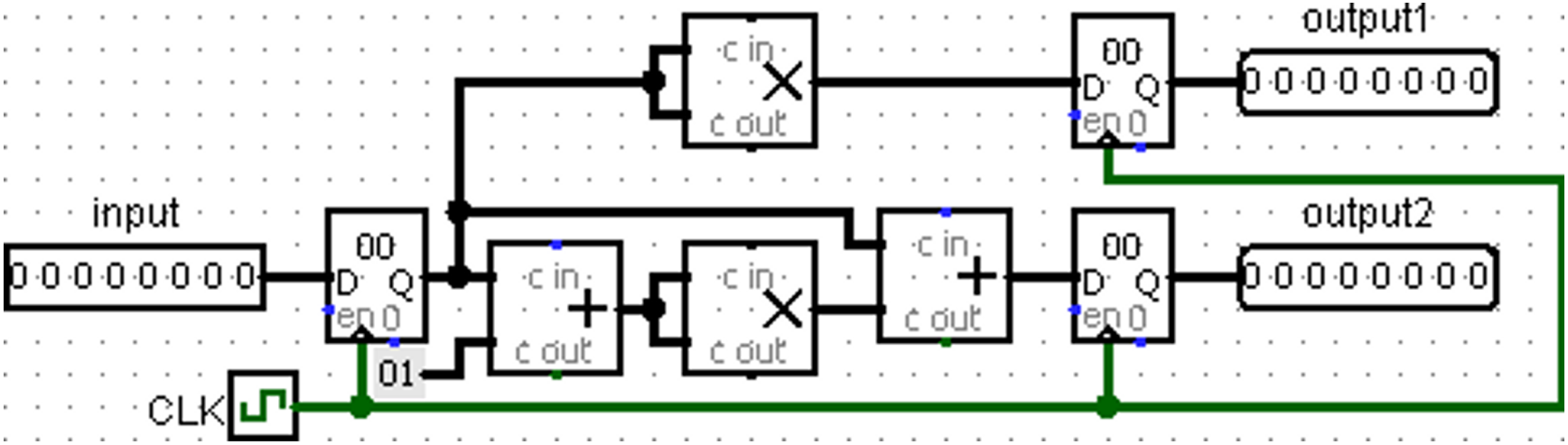


(A)	550 ps
(B)	750 ps
(C)	500 ps
(D)	700 ps

Question: Want to run on 1 GHz processor.

t_{add} = 100 ps. t_{mult} = 200 ps. t_{setup} = t_{hold} = 50 ps.

What is the maximum clock-to-q time?



(A)	550 ps
(B)	750 ps
(C)	500 ps
(D)	700 ps

Bottom path is critical path:

clock-to-q + 100 + 200 + 100 + 50 < 1000 ps = 1ns

clock-to-q + 450 < 1000 ps

clock-to-q < 550

Agenda

- Muxes
- Sequential Logic Timing
- Maximum Clock Frequency
- **Functional Units**
- Summary

Bonus Slides

- FSMs

Functional Units (a.k.a. Execution Unit)

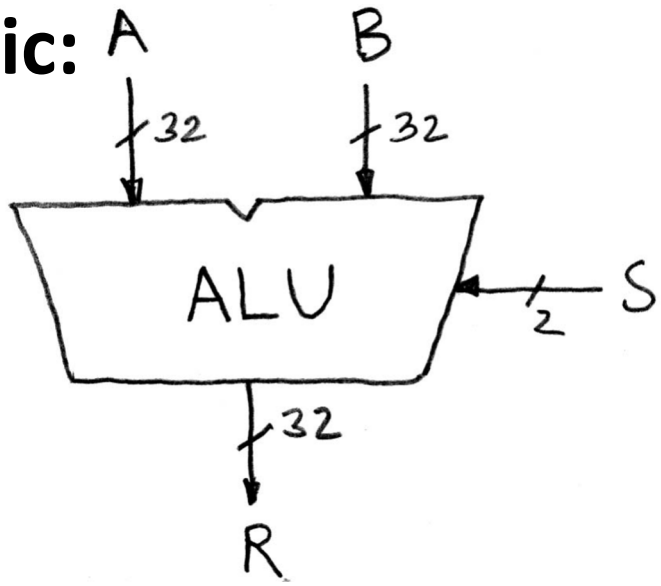
- Now that we know sequential logic, we can explore some pieces of a processor!

- Functional Units are a part of the processor that perform operations and calculations based on the running program
  - Arithmetic Logic Unit
  - Floating Point Unit
  - Load/Store Unit
  - and several more...

Invented by *John Von Neumann*
He also invented
- Stored Program Concept
- Mergesort
- Mutually Assured Destruction

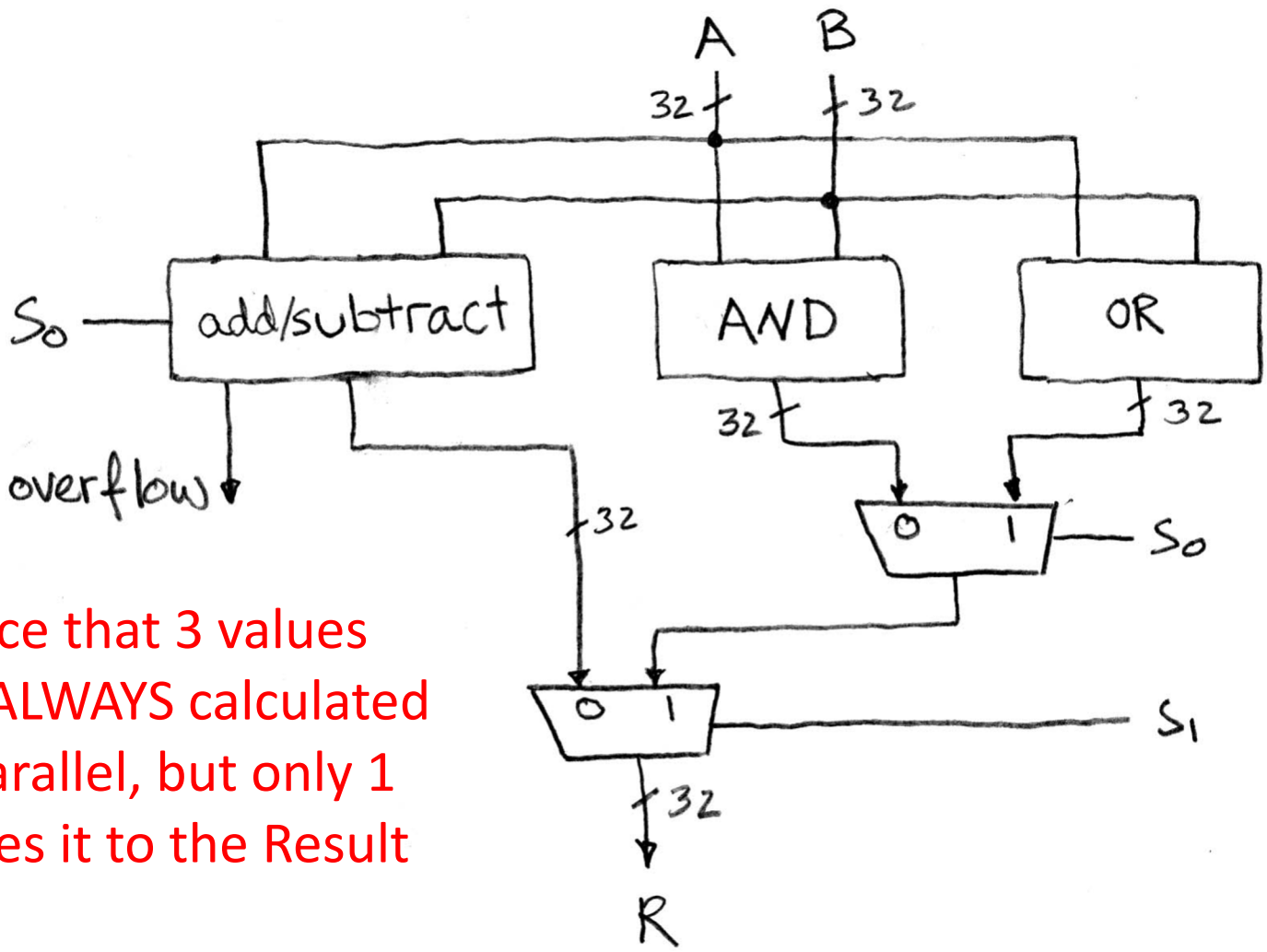
Arithmetic Logic Unit (ALU)

- Most processors contain a special logic block called the "Arithmetic Logic Unit" (ALU)
  - We'll show you an easy one that does ADD, SUB, bitwise AND, and bitwise OR
- **Schematic:**



when S=00, R = A + B
when S=01, R = A – B
when S=10, R = A AND B
when S=11, R = A OR B

Simple ALU Schematic



Notice that 3 values are ALWAYS calculated in parallel, but only 1 makes it to the Result

Adder/Subtractor Design

1) Combinational Logic design we've seen before:
   - write out truth table
   - convert to boolean logic
   - minimize logic
   - then implement

How big might truth table and/or Boolean expression get?

Adder/Subtractor Design

2) Break down the problem into smaller pieces that we can cascade or hierarchically layer
   - Let's try this approach instead

Adder/Subtractor: 1-bit LSB Adder

$$
\begin{array}{cccc|c}
 & a_3 & a_2 & a_1 & a_0 \\
+ & b_3 & b_2 & b_1 & b_0 \\
\hline
 & s_3 & s_2 & s_1 & s_0
\end{array}
$$

Carry-out bit

a_0	b_0	s_0	c_1
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

$$s_0 = a_0 \text{ XOR } b_0$$
$$c_1 = a_0 \text{ AND } b_0$$

Adder/Subtractor: 1-bit Adder

Possible carry-in c_1

$$\begin{array}{cccc} a_3 & a_2 & a_1 & a_0 \\ + \; b_3 & b_2 & b_1 & b_0 \\ \hline s_3 & s_2 & s_1 & s_0 \end{array}$$

a_i	b_i	c_i	s_i	c_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

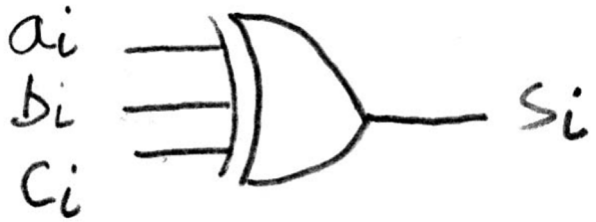
Here defining XOR of many inputs to be 1 when an *odd* number of inputs are 1

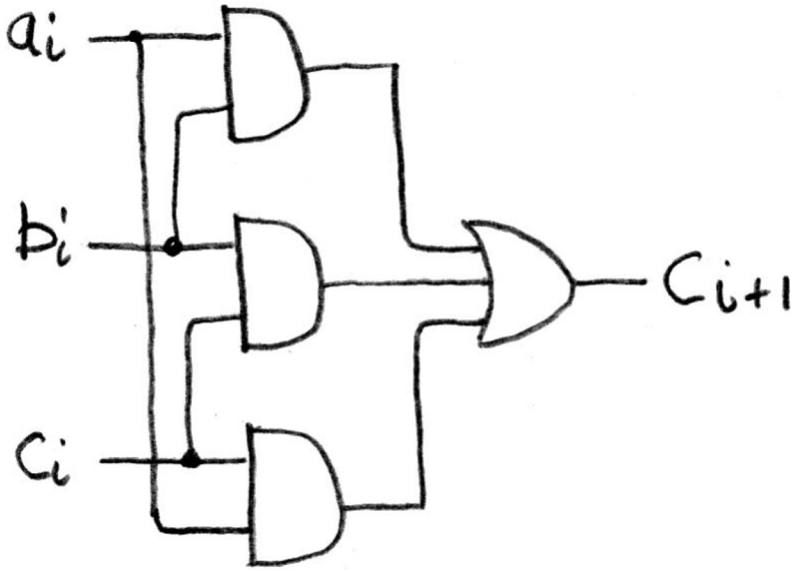
$$s_i = \mathrm{XOR}(a_i, b_i, c_i)$$

$$c_{i+1} = \mathrm{MAJ}(a_i, b_i, c_i) = a_i b_i + a_i c_i + b_i c_i$$

Adder/Subtractor: 1-bit Adder

- **Circuit Diagrams:**

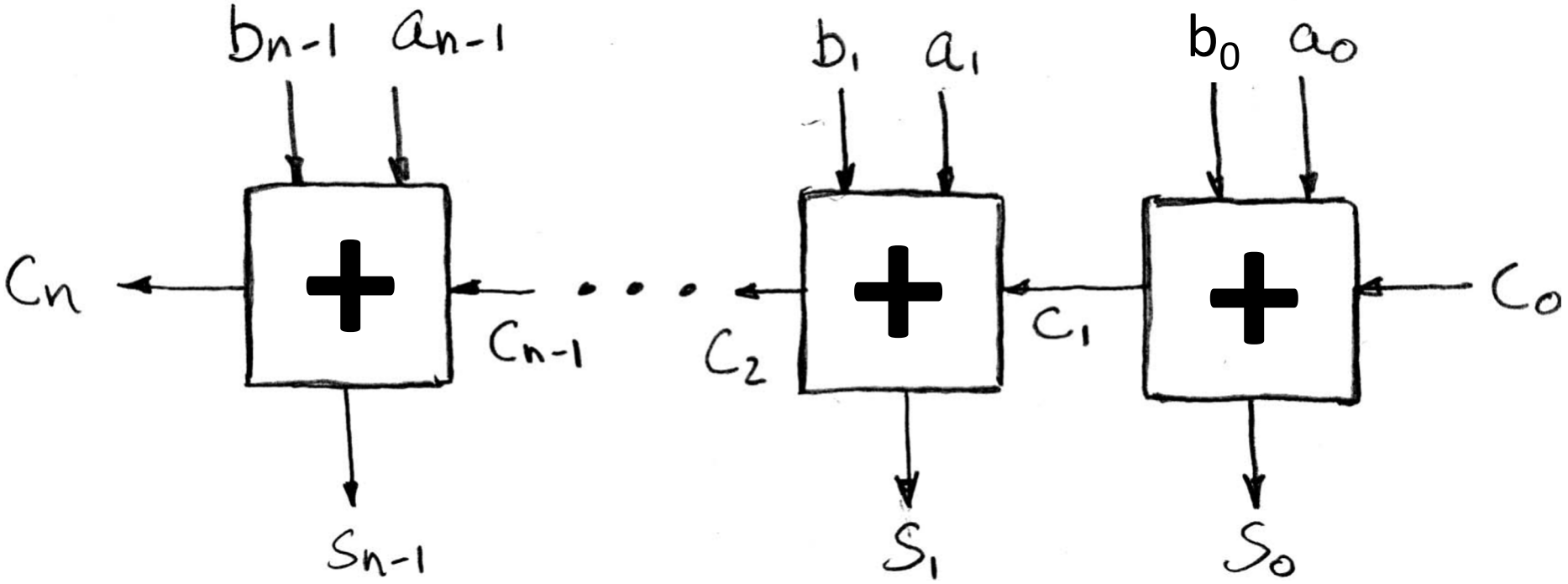




$$s_i = \mathrm{XOR}(a_i, b_i, c_i)$$
$$c_{i+1} = \mathrm{MAJ}(a_i, b_i, c_i) = a_i b_i + a_i c_i + b_i c_i$$

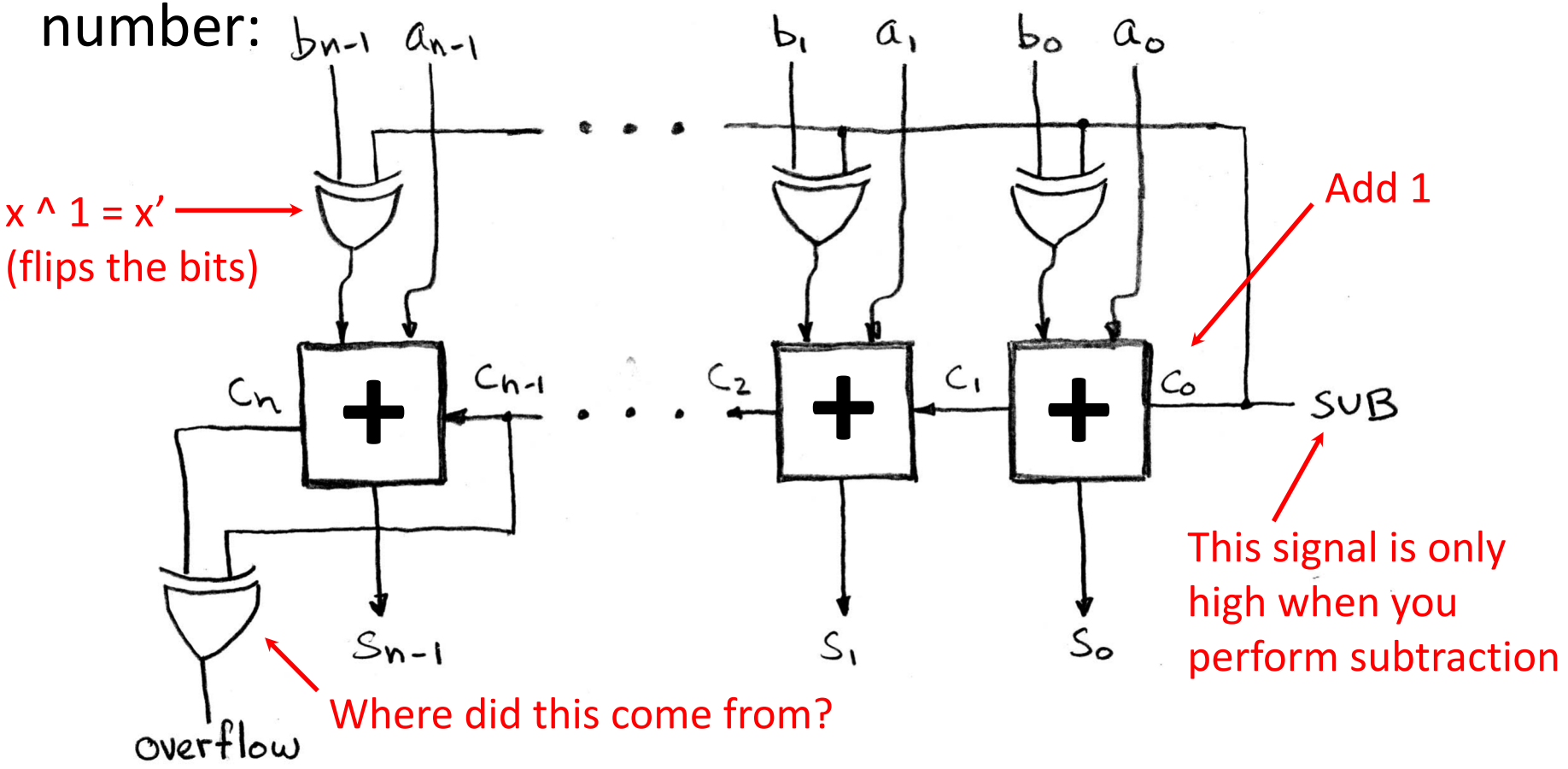
N x 1-bit Adders → N-bit Adder

- Connect $CarryOut_{i-1}$ to $CarryIn_i$ to chain adders:



Two's Complement Adder/Subtractor

- Subtraction accomplished by adding negated number:



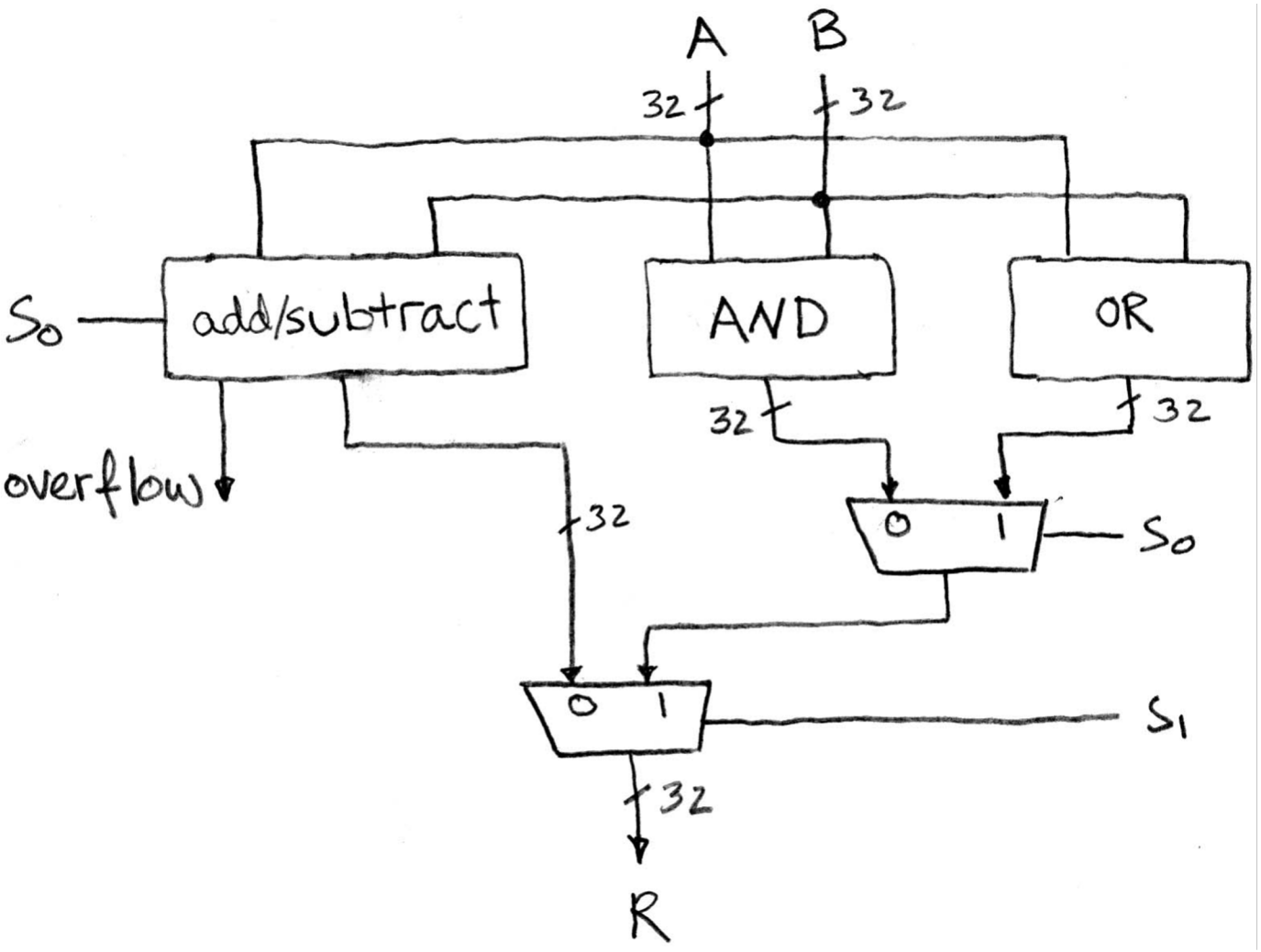
Detecting Overflow

- Unsigned overflow
  - On addition, if carry-out from MSB is 1
  - On subtraction, if carry-out from MSB is 0
    - This case is a lot harder to see than you might think
- Signed overflow
  1) Overflow from adding “large” positive numbers
  2) Overflow from adding “large” negative numbers

Signed Overflow Examples (4-bit)

- Overflow from two positive numbers:
  - 0111 + 0111, 0111 + 0001, 0100 + 0100.
  - Carry-out from the 2nd MSB (but not MSB)
    - pos + pos ≠ neg
- Overflow from two negative numbers:
  - 1000 + 1000, 1000 + 1111, 1011 + 1011.
  - Carry-out from the MSB (but not 2nd MSB)
    - neg + neg ≠ pos
- Expression for signed overflow: C_n XOR C_{n-1}

Simple ALU Schematic



Agenda

- Muxes
- Sequential Logic Timing
- Maximum Clock Frequency
- Finite State Machines
- Functional Units
- **Summary**

Bonus Slides

- Logisim Intro

Summary

- Hardware systems are constructed from *Stateless* Combinational Logic and *Stateful* Sequential Logic (includes registers)
- Circuits have a delay to them, and the critical path (longest delay between registers) determines the maximum clock frequency

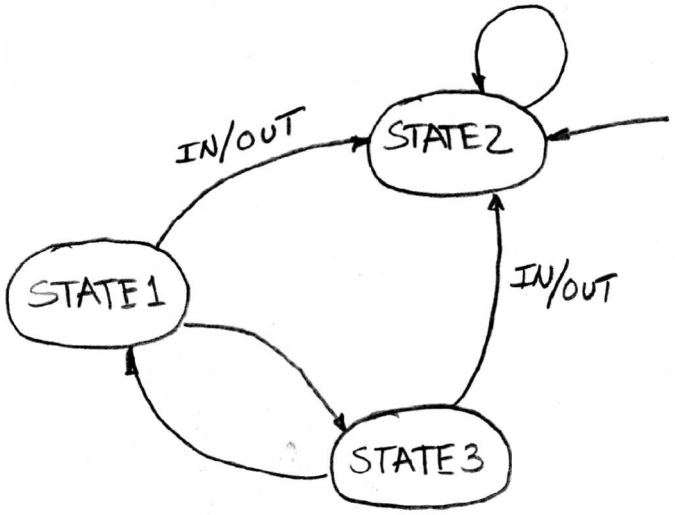
Non Testable Material

Back to representations

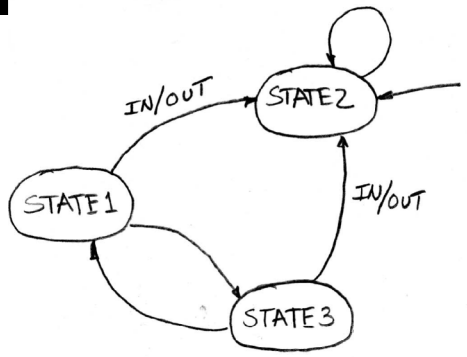
- How do we represent sequential logic?
  - Truth tables could account for history
  - We could do boolean logic with prior state as a variable
- We can also use a new representation:
  Finite State Machines

Finite State Machines (FSMs)

- A convenient way to conceptualize computation over time
- Function can be represented with a *state transition diagram*
- With combinational logic and registers, any FSM can be implemented in hardware!



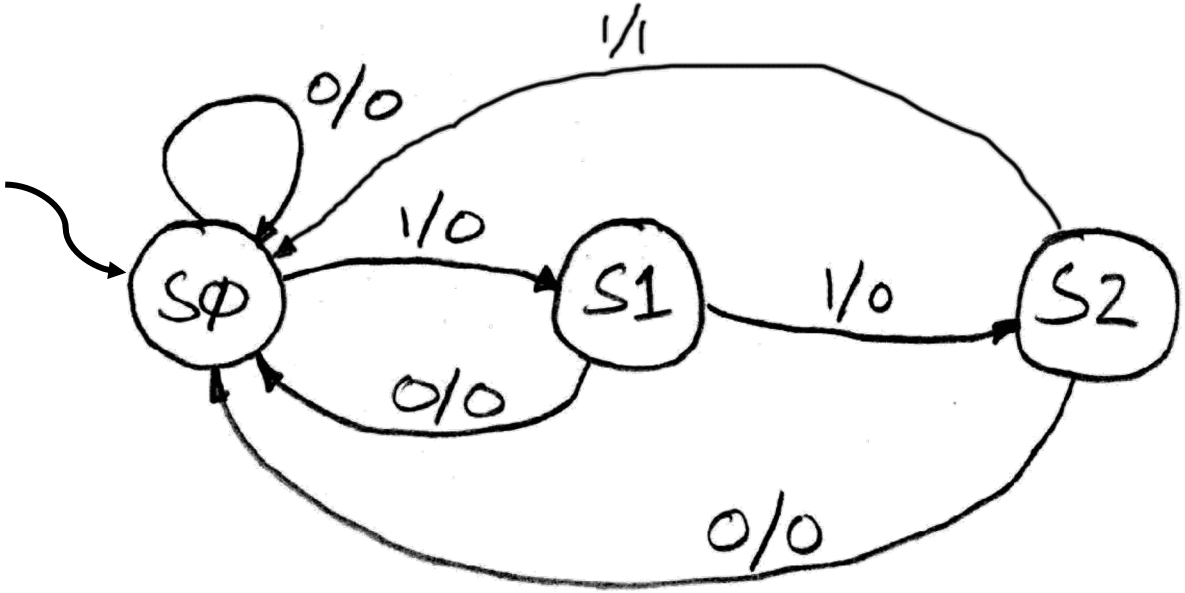
FSM Overview



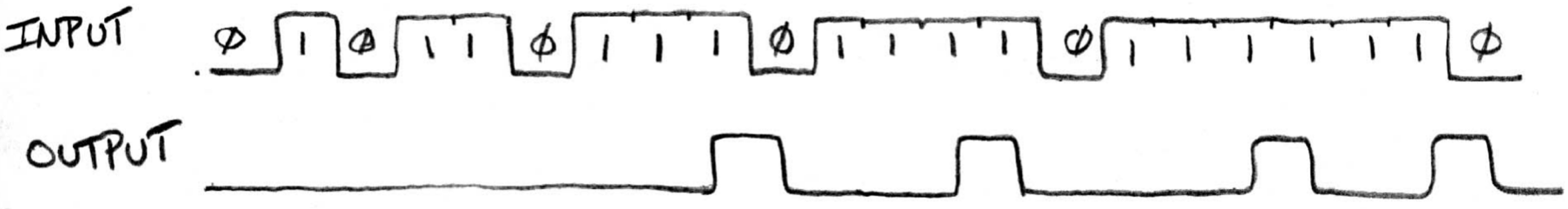
- An FSM (in this class) is defined by:
  - A set of *states* S (circles)
  - An *initial state* s_0 (only arrow not between states)
  - A *transition function* that maps from the current input and current state to the output and the next state (arrows between states)
- State transitions are controlled by the clock:
  - On each clock cycle the machine checks the inputs and generates a new state (could be same) and

Example: 3 Ones FSM

- FSM to detect 3 consecutive 1's in the Input

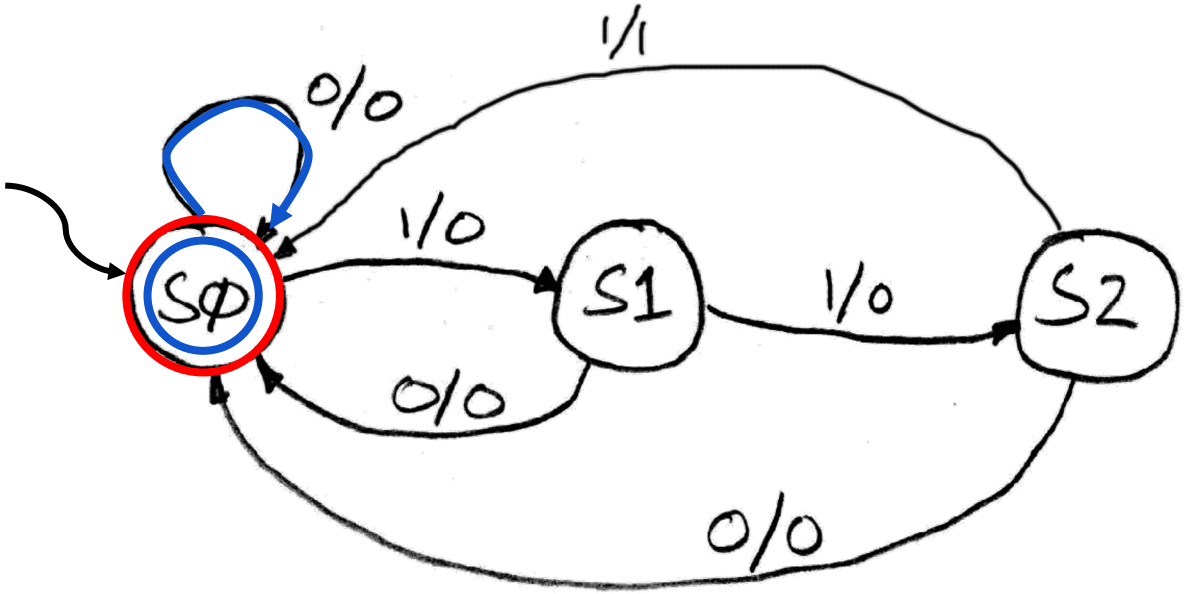


States: S0, S1, S2
Initial State: S0
Transitions of form:
input/output

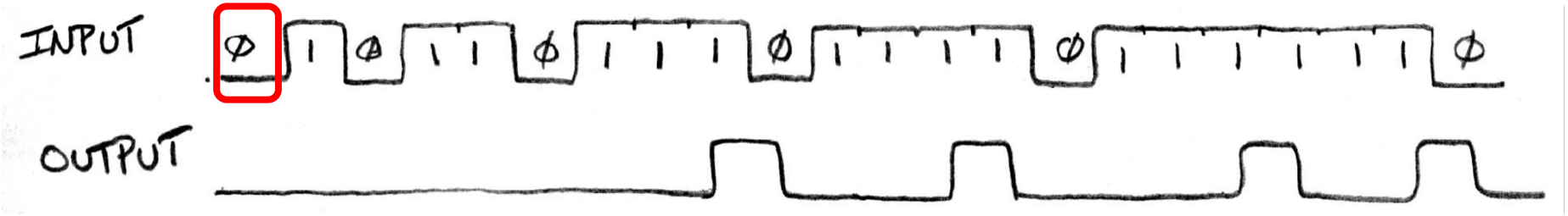


Example: 3 Ones FSM

- FSM to detect 3 consecutive 1's in the Input

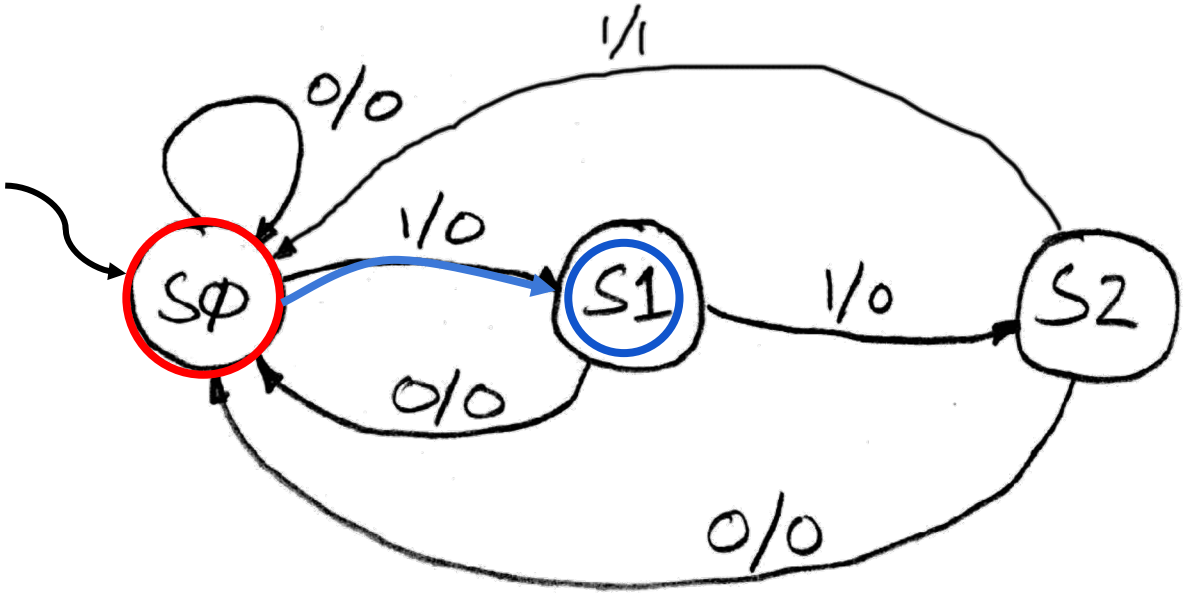


Starting state in red
Resulting state in Blue

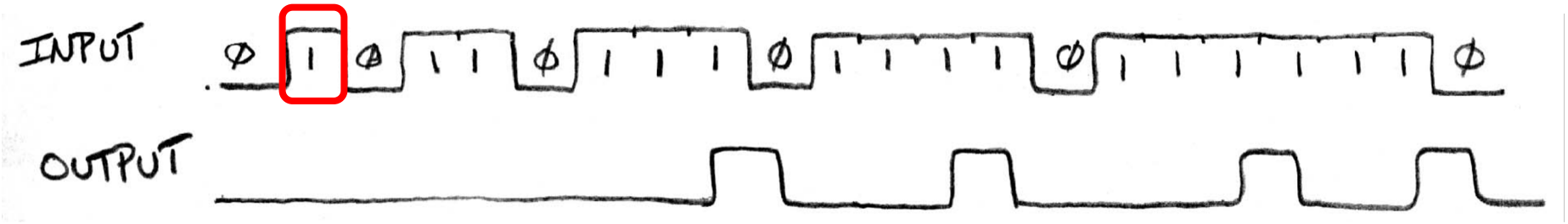


Example: 3 Ones FSM

- FSM to detect 3 consecutive 1's in the Input

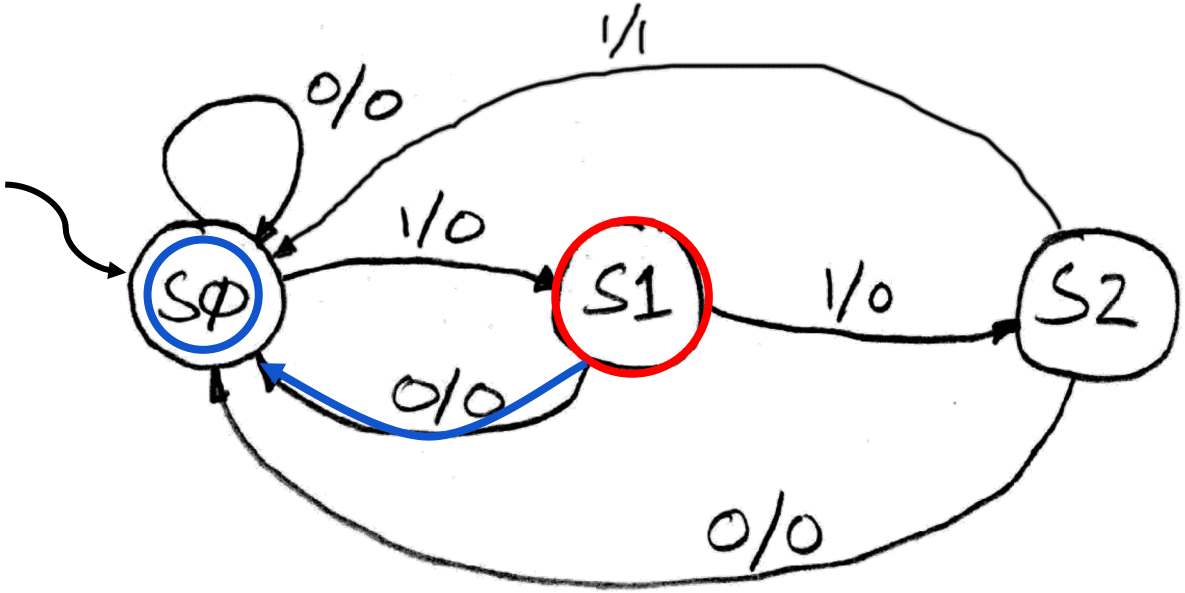


Starting state in red
Resulting state in Blue

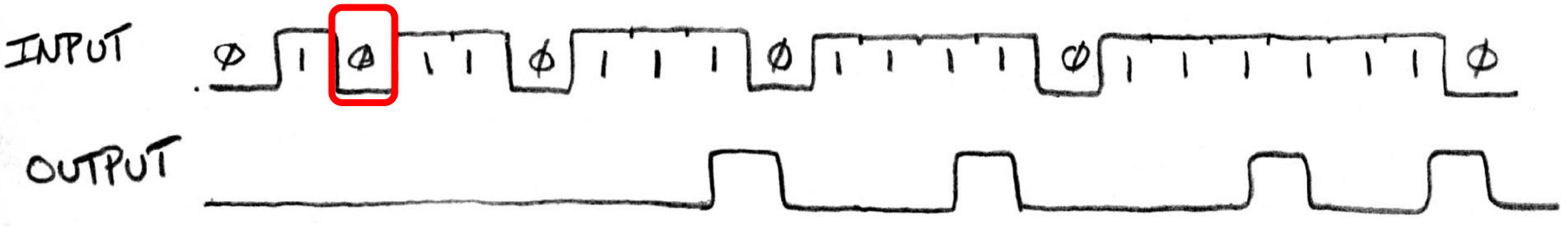


Example: 3 Ones FSM

- FSM to detect 3 consecutive 1's in the Input

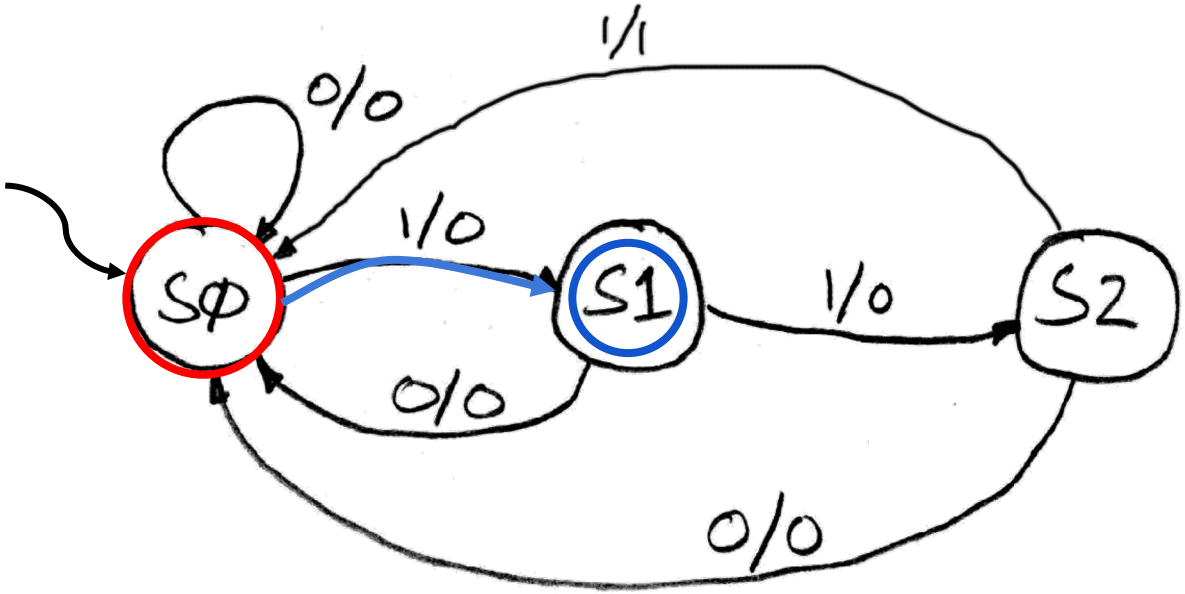


Starting state in red
Resulting state in Blue

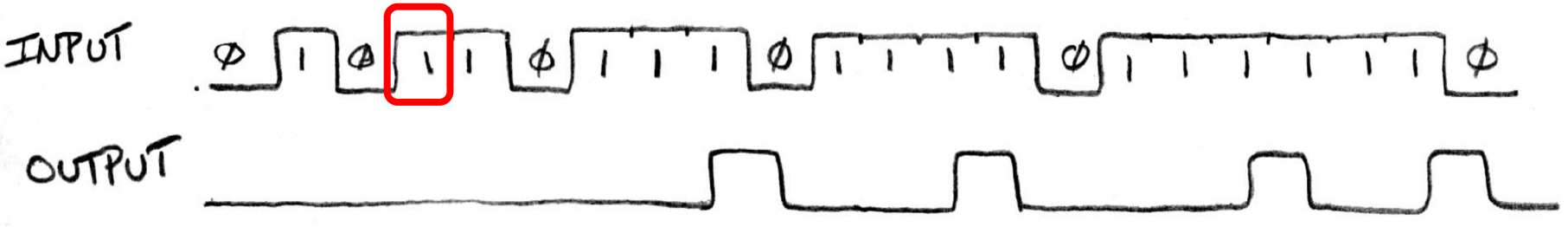


Example: 3 Ones FSM

- FSM to detect 3 consecutive 1's in the Input

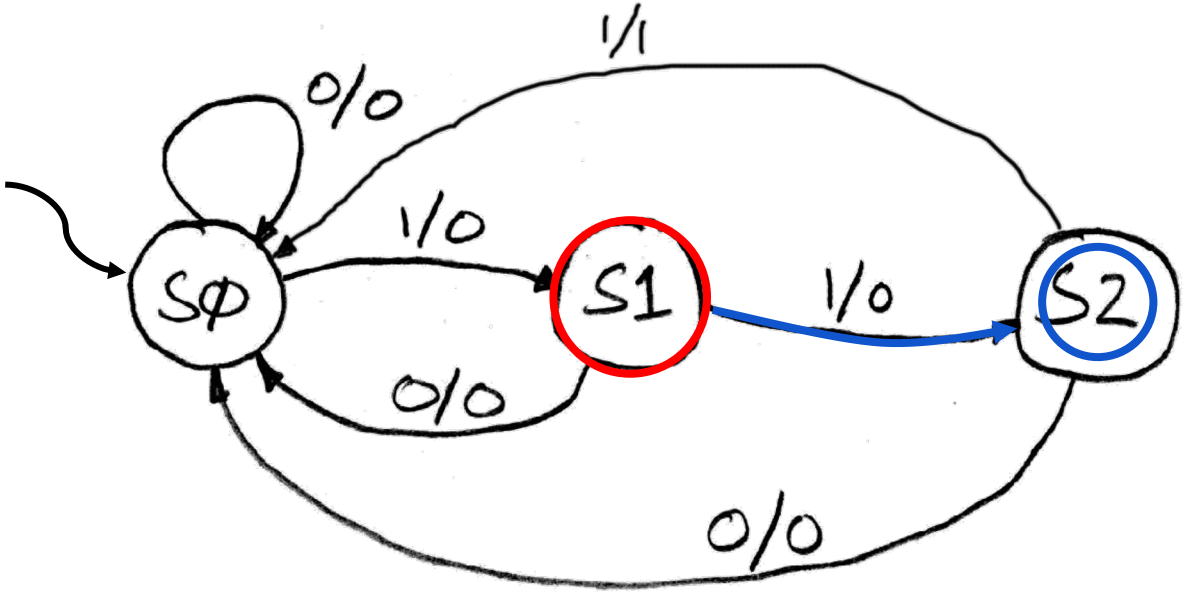


Starting state in red
Resulting state in Blue

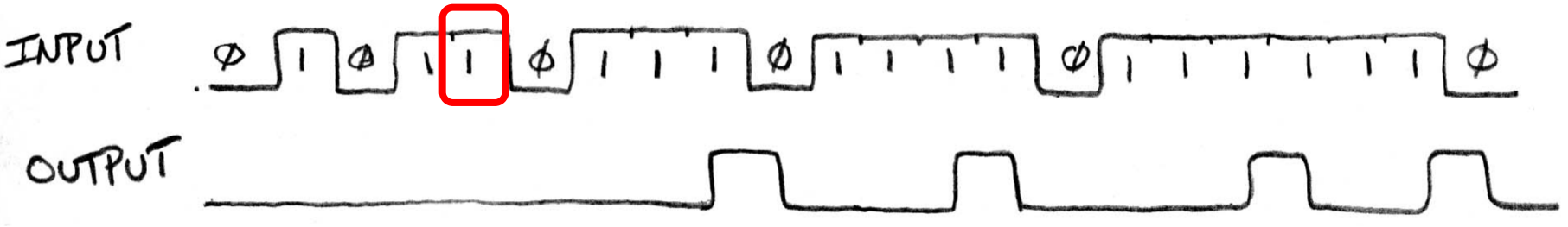


Example: 3 Ones FSM

- FSM to detect 3 consecutive 1's in the Input

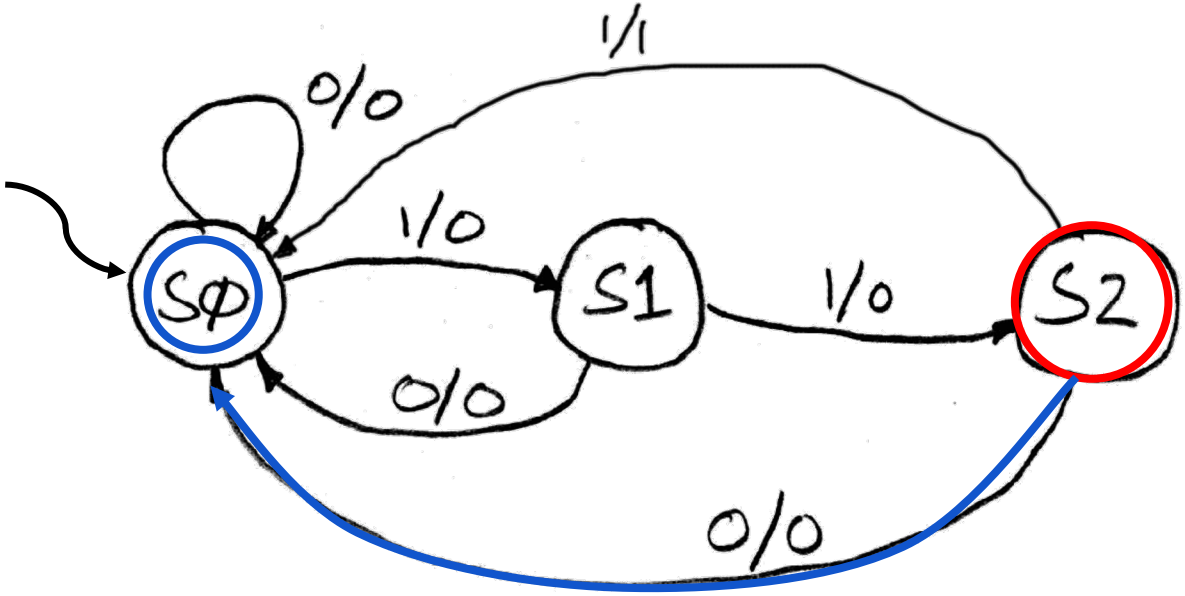


Starting state in red
Resulting state in Blue

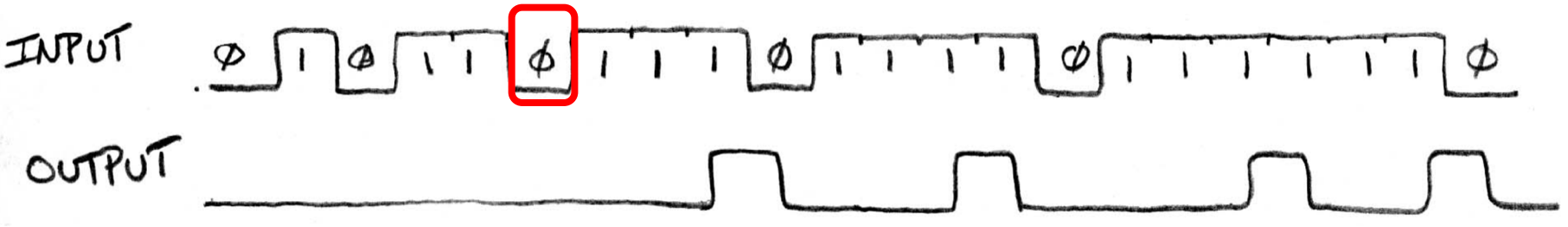


Example: 3 Ones FSM

- FSM to detect 3 consecutive 1's in the Input

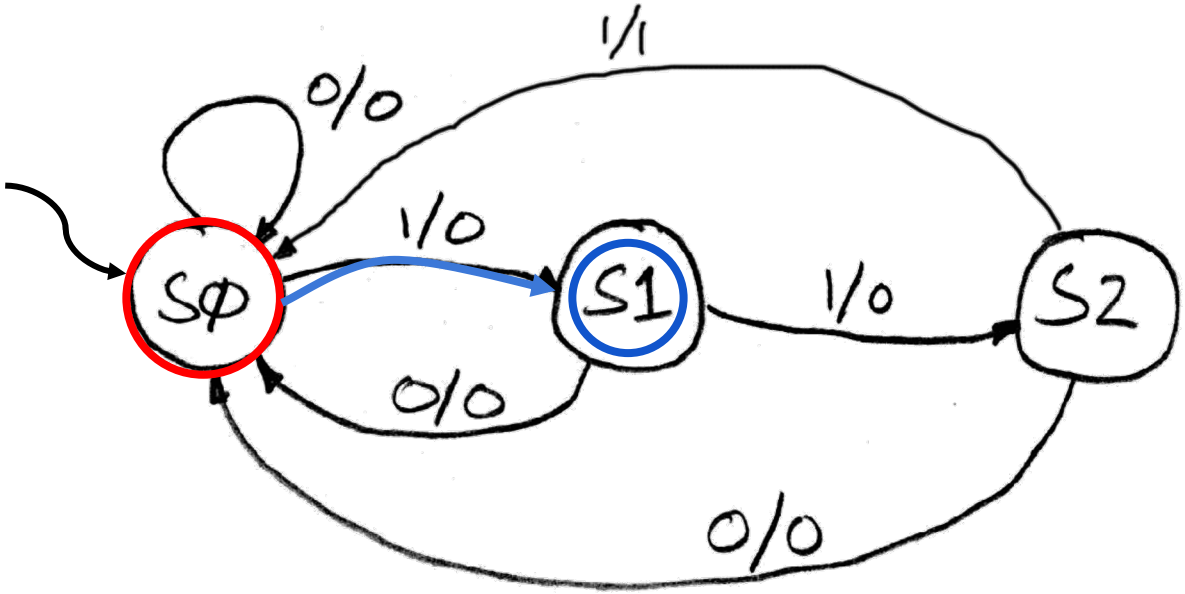


Starting state in red
Resulting state in Blue

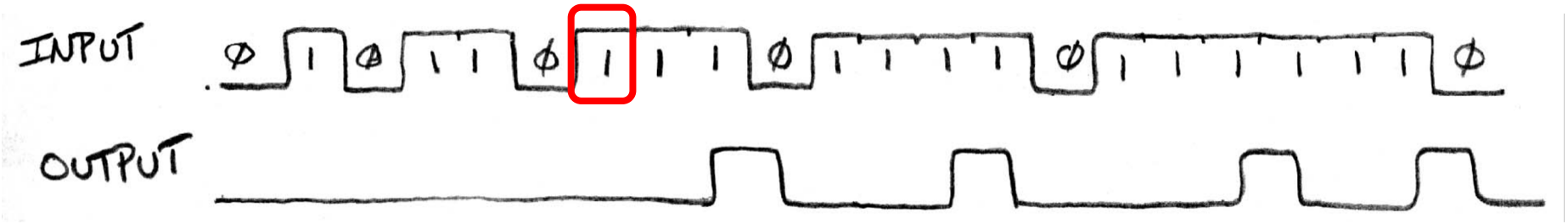


Example: 3 Ones FSM

- FSM to detect 3 consecutive 1's in the Input

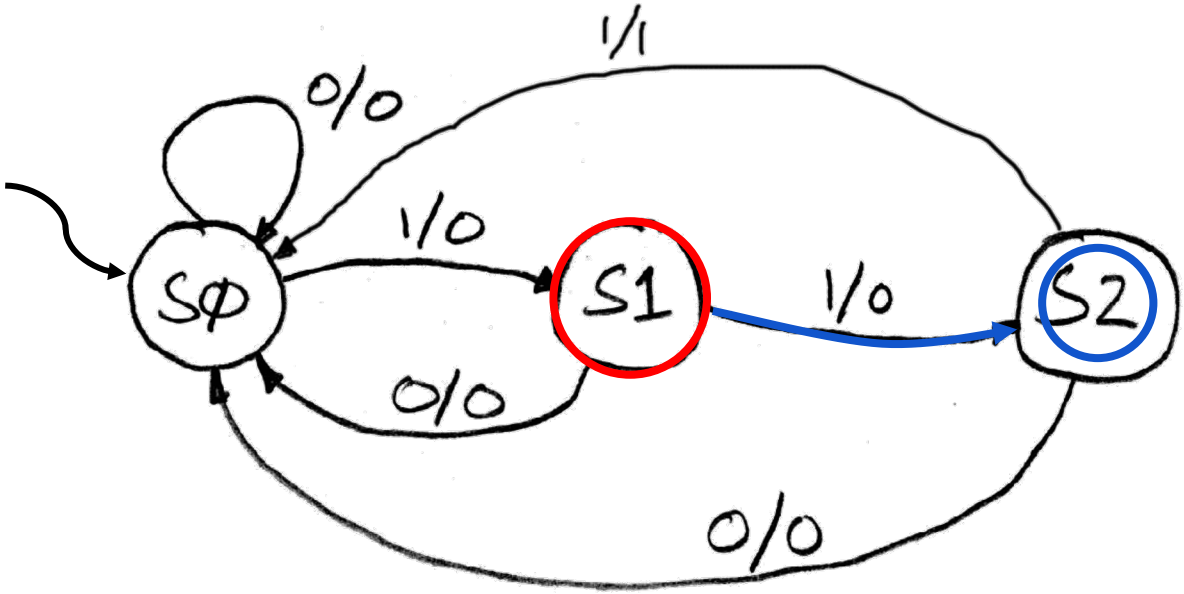


Starting state in red
Resulting state in Blue

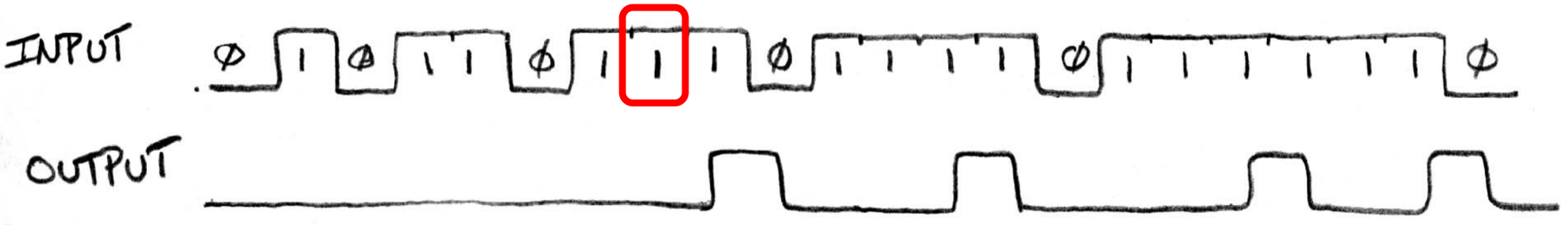


Example: 3 Ones FSM

- FSM to detect 3 consecutive 1's in the Input

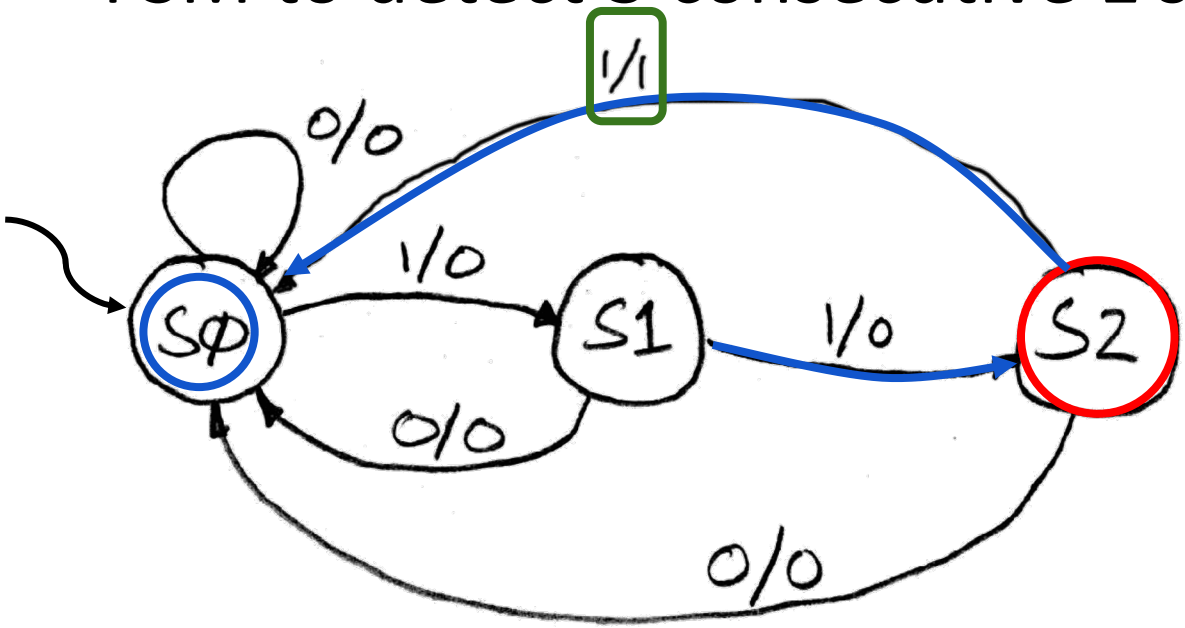


Starting state in red
Resulting state in Blue

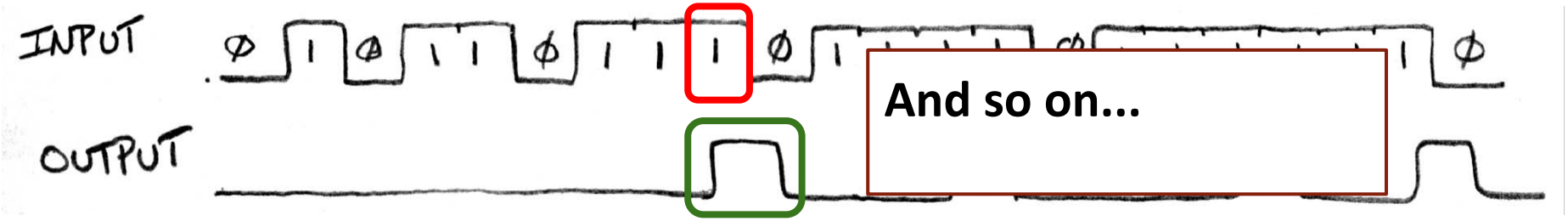


Example: 3 Ones FSM

- FSM to detect 3 consecutive 1's in the Input

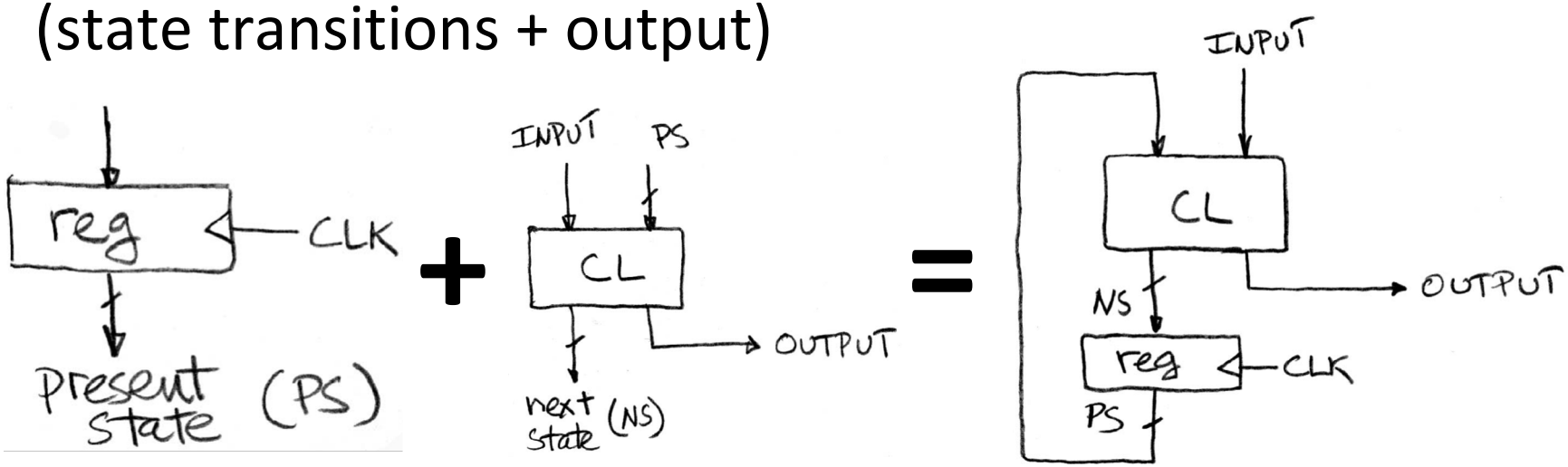


Starting state in red
Resulting state in Blue



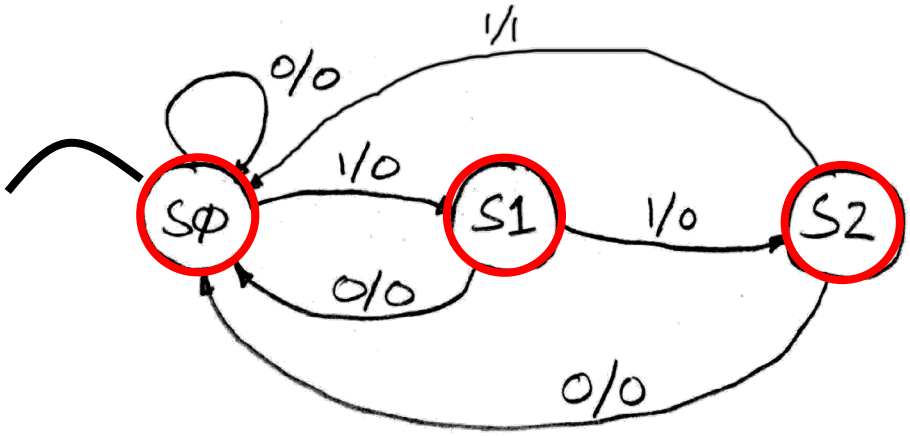
Hardware Implementation of FSM

- Register holds a representation of the FSM's state
  - Must assign a *unique* bit pattern for each state
  - Output is *present/current state* (PS/CS)
  - Input is *next state* (NS)
- Combinational Logic implements transition function (state transitions + output)



FSM: Combinational Logic

- Read off transitions into Truth Table!
  - **Inputs:** Current State (CS) and Input (In)
  - **Outputs:** Next State (NS) and Output (Out)



CS	In	NS	Out
00	0	00	0
00	1	01	0
01	0	00	0
01	1	10	0
10	0	00	0
10	1	00	1

Unspecified Output Values

- Our FSM has only 3 states
  - 2 entries in truth table are undefined/unspecified
- Use symbol 'X' to mean it can be either a 0 or 1
  - Make choice to simplify final expression
  - Any choice is correct

CS	In	NS	Out
00	0	00	0
00	1	01	0
01	0	00	0
01	1	10	0
10	0	00	0
10	1	00	1
11	0	XX	X
11	1	XX	X

3 Ones FSM in Hardware

- 2-bit **Register** needed for state
- **CL:** $NS_1 = CS_0 In$, $NS_0 = \neg CS_1 \neg CS_0 In$, $Out = CS_1 In$