信息科学与技术学院
School of Information Science and Technology

CS 110
Computer Architecture
Datapath

Instructors: Siting Liu & Yuan Xiao
Course website: https://faculty.sist.shanghaitech.edu.cn/liust/courses/CS110.html
School of Information Science and Technology (SIST)
ShanghaiTech University

2026/04/07

Administratives

- Lab 6 available, please prepare in advance, to check next week!
- HW 3 ddl April 9th!
- Proj 1.1 ddl TODAY!
- Proj 1.2 released, ddl Apr. 27th.
- HW 4 to be release today, ddl Apr. 21st.
- Discussion this week on digital circuits.

Outline

- Useful building blocks
  - ALU design
  - Register file
  - Memory considerations
- Datapath
- Design of the controller

Warm-up

- A classic problem: sequence detection for “010” (non-overlapping)

```
Input: 0 1 0 0 1 0 1 0 1 1 0
Output: 0 0 0 1 0 0 1 0 0 0 0
```

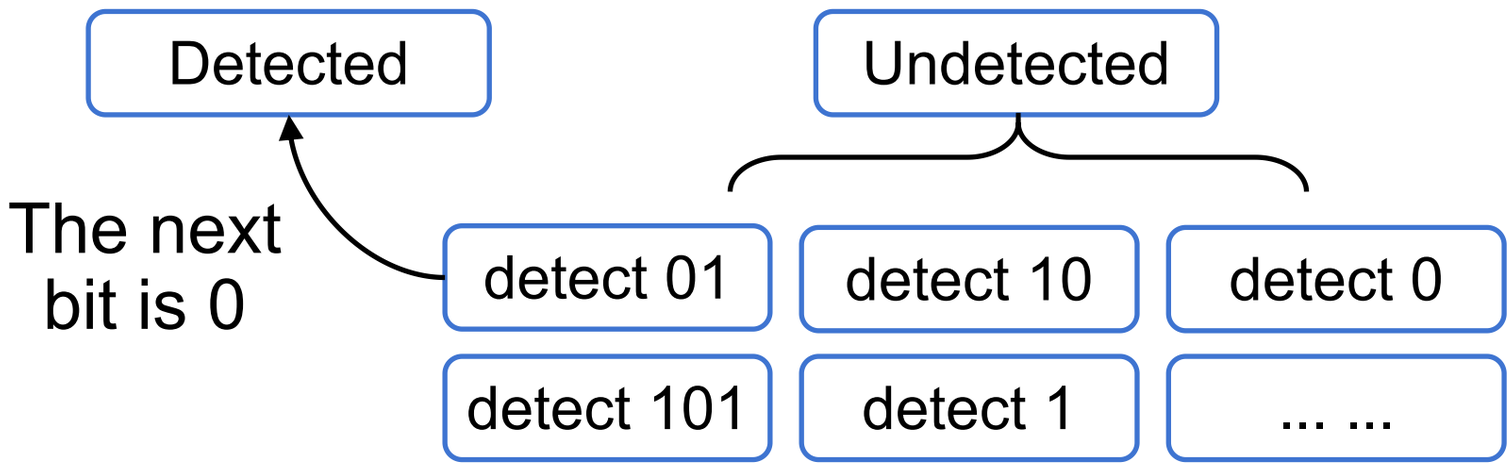
- Step 1: Draw finite state machine of the desired function (we ignore the initialization)
- Step 2: Define/assign binary numbers to represent the states, the inputs and the outputs
- Step 3: Write down the truth table (enumerate input/previous state (and current state) and their corresponding current state (and output))
- Step 4: Use template and decide the combinational block for state transition and output logic

Warm-up

- A classic problem: sequence detection for “010” (non-overlapping)

```
Input: 0 1 0 0 1 0 1 0 1 1 0
Output: 0 0 0 1 0 0 1 0 0 0 0
```

- Step 1: Draw finite state machine of the desired function

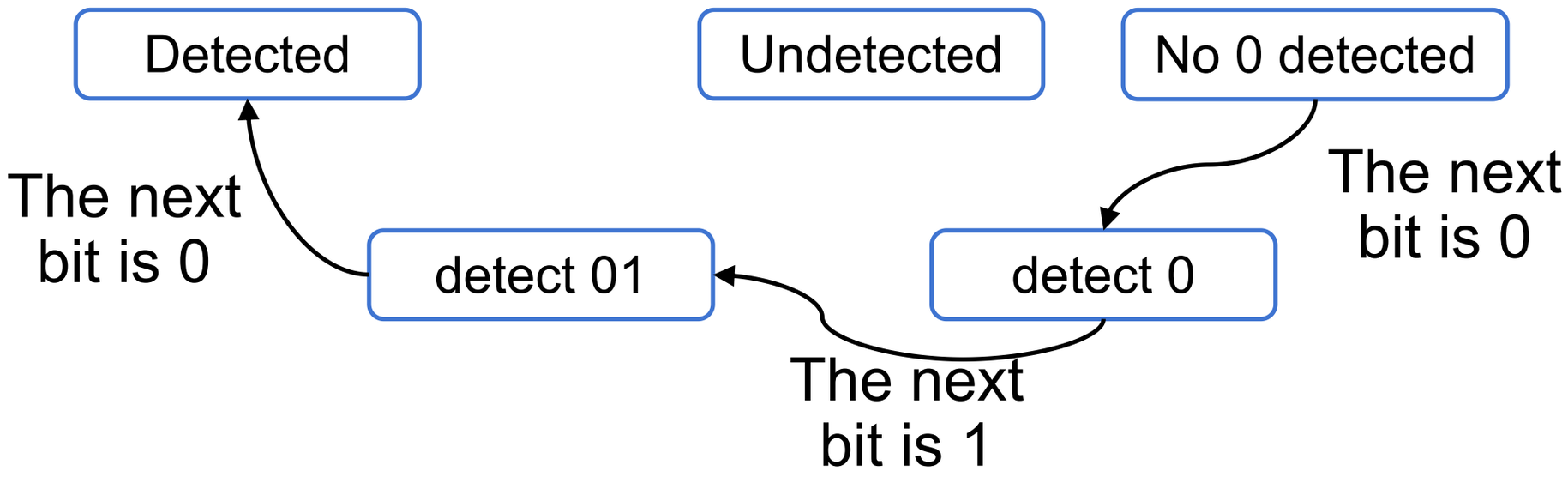


Warm-up

- A classic problem: sequence detection for “010” (non-overlapping)

```
Input: 0 1 0 0 1 0 1 0 1 1 0
Output: 0 0 0 1 0 0 1 0 0 0 0
```

- Step 1: Draw finite state machine of the desired function

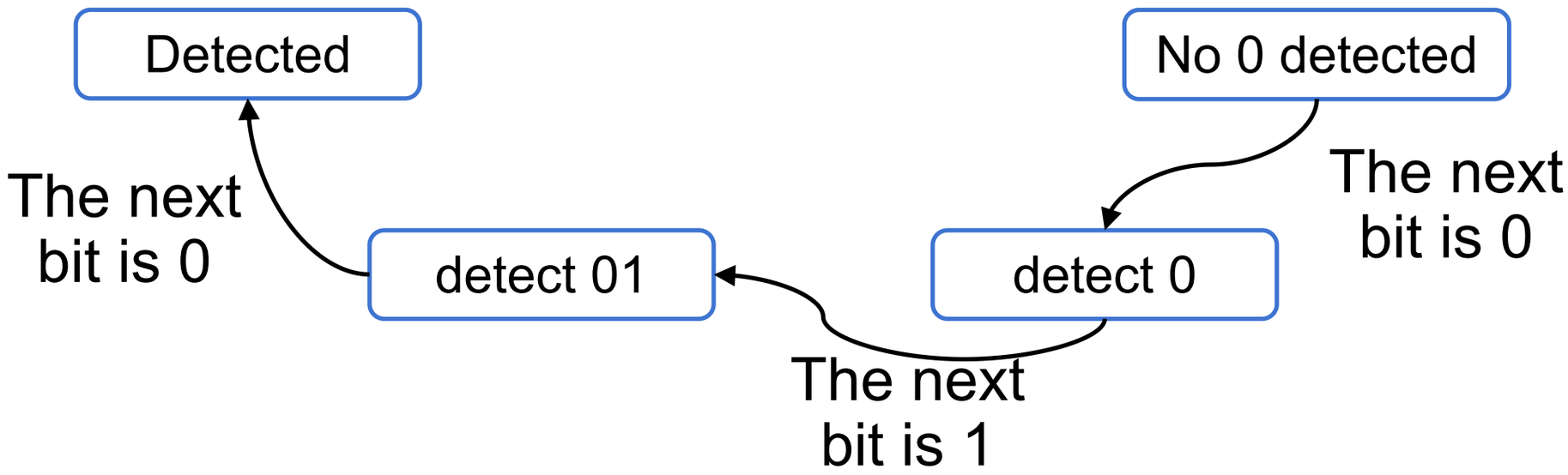


Warm-up

- A classic problem: sequence detection for “010” (non-overlapping)

```
Input: 0 1 0 0 1 0 1 0 1 1 0
Output: 0 0 0 1 0 0 1 0 0 0 0
```

- Step 1: Draw finite state machine of the desired function

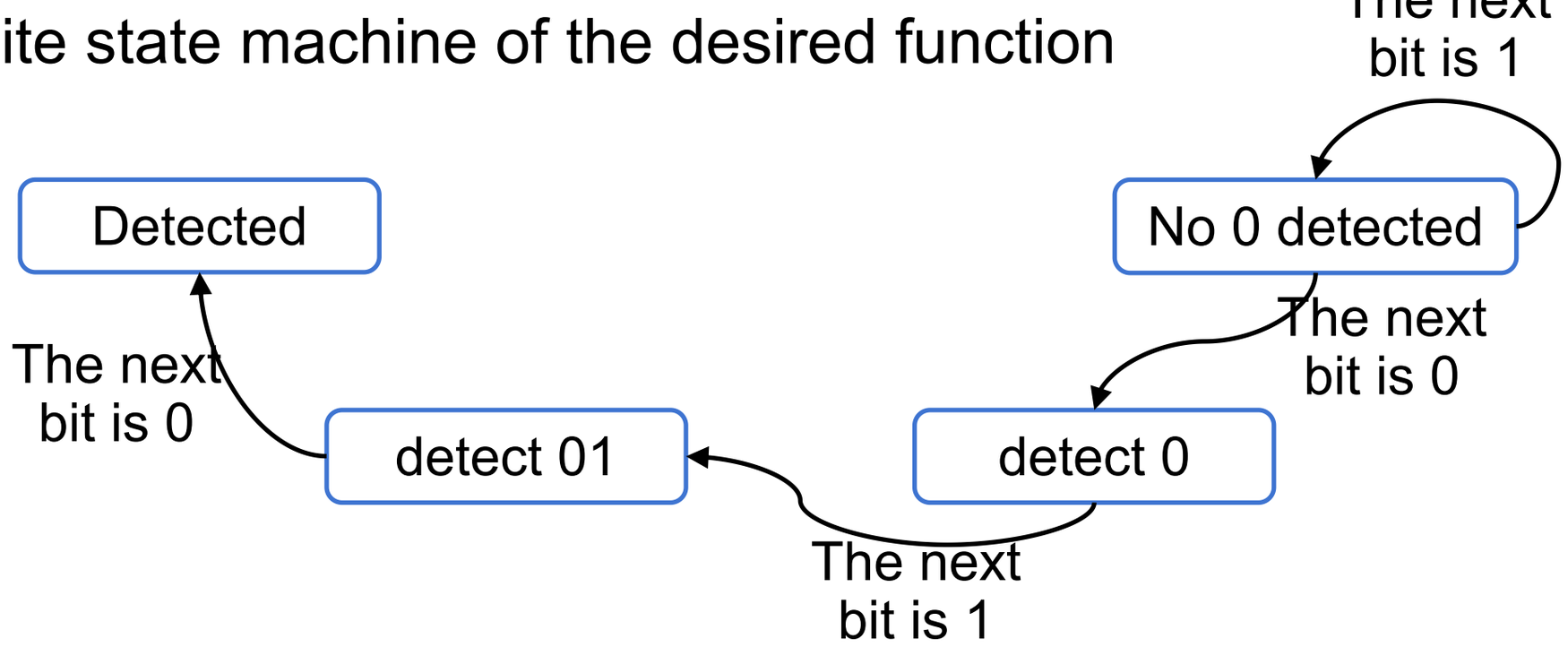


Warm-up

- A classic problem: sequence detection for “010” (non-overlapping)

```
Input: 0 1 0 0 1 0 1 0 1 1 0
Output: 0 0 0 1 0 0 1 0 0 0 0
```

- Step 1: Draw finite state machine of the desired function

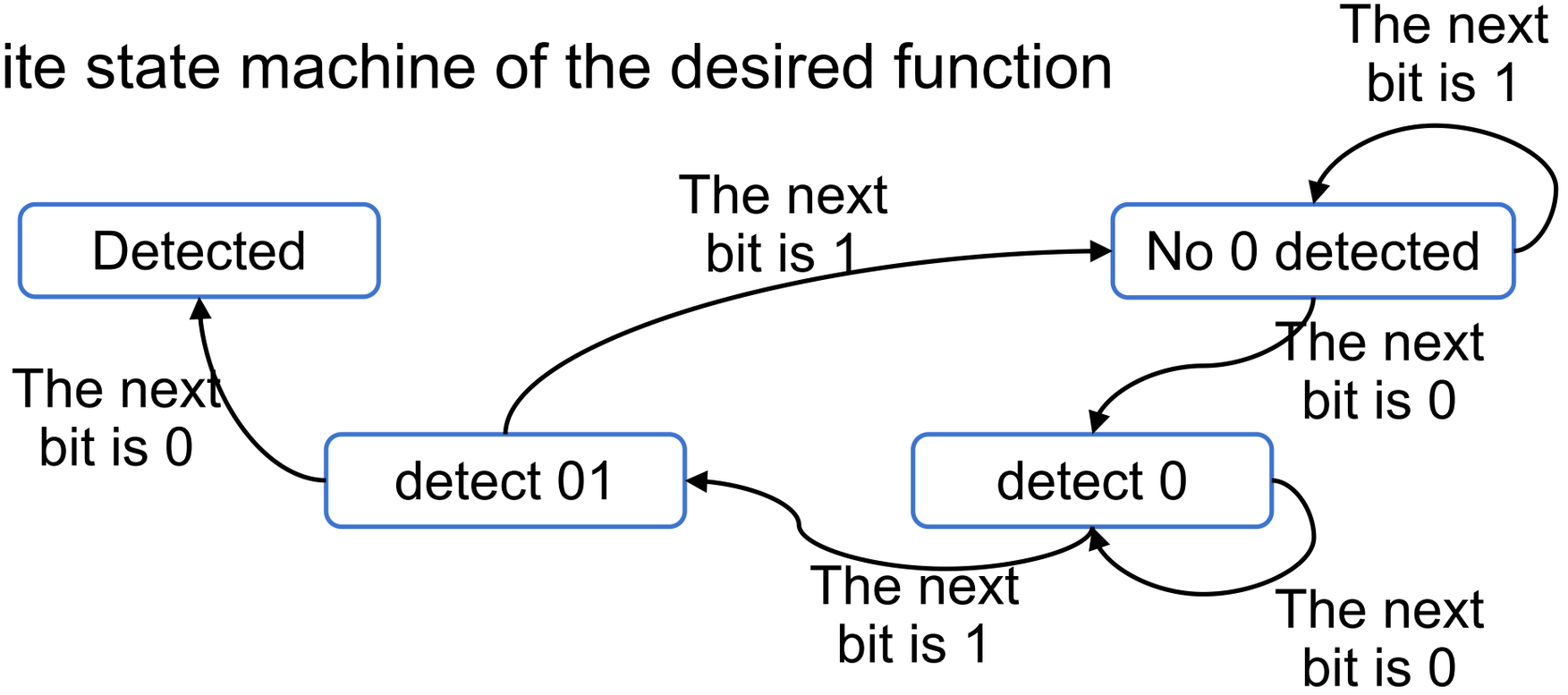


Warm-up

- A classic problem: sequence detection for “010” (non-overlapping)

```
Input: 0 1 0 0 1 0 1 0 1 1 0
Output: 0 0 0 1 0 0 1 0 0 0 0
```

- Step 1: Draw finite state machine of the desired function

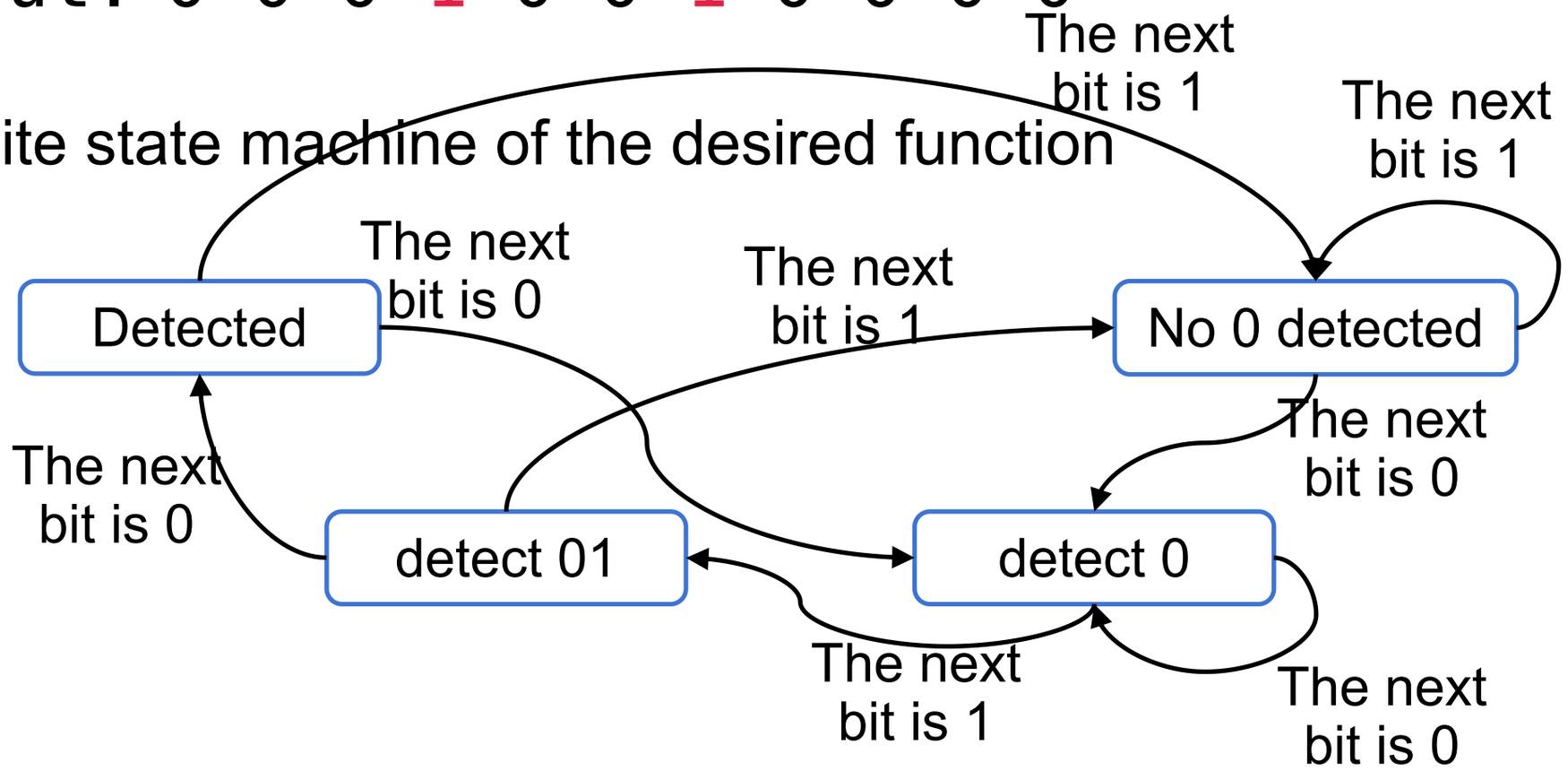


Warm-up

- A classic problem: sequence detection for “010” (non-overlapping)

```
Input: 0 1 0 0 1 0 1 0 1 1 0
Output: 0 0 0 1 0 0 1 0 0 0 0
```

- Step 1: Draw finite state machine of the desired function

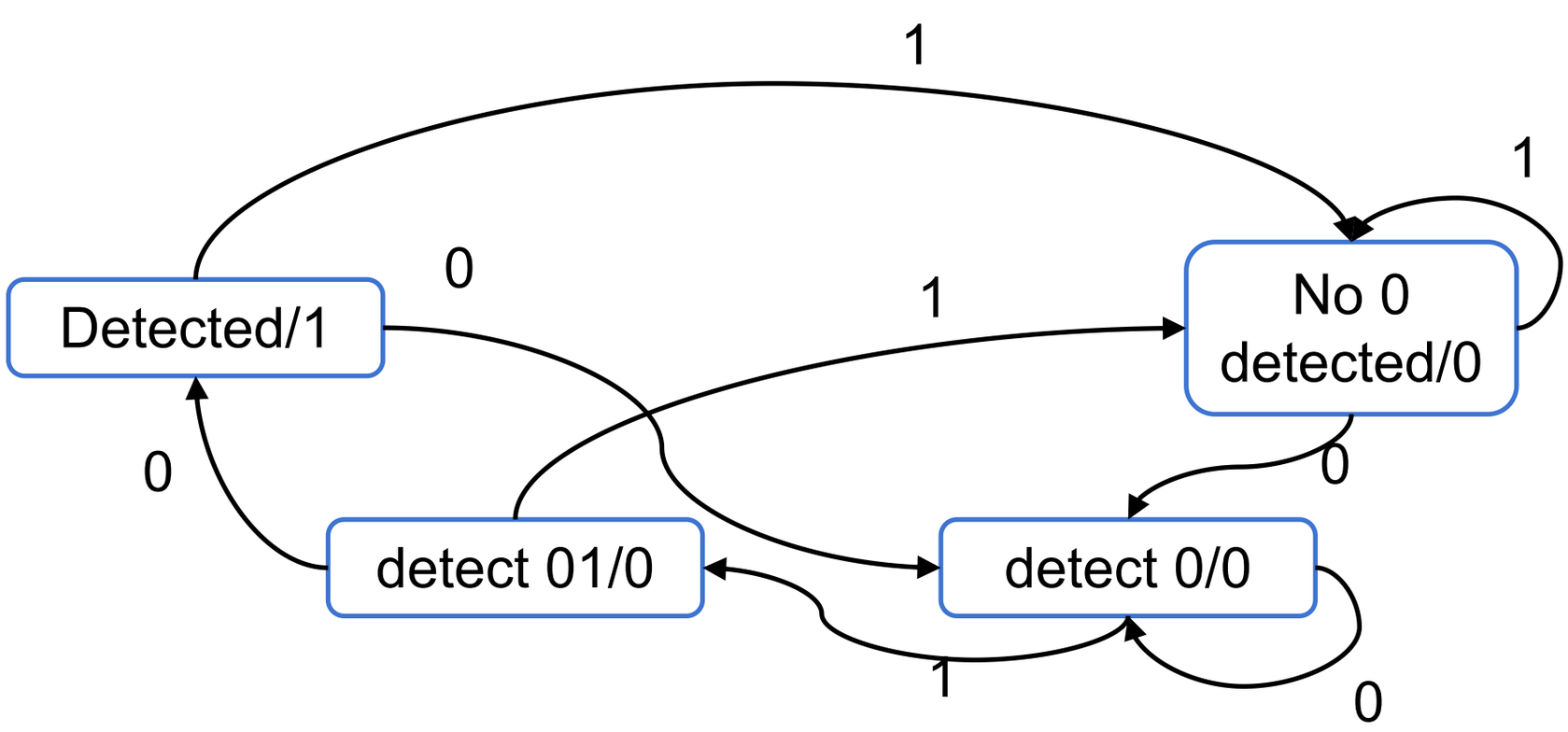


Warm-up

- A classic problem: sequence detection for “010” (non-overlapping)

```
Input: 0 1 0 0 1 0 1 0 1 1 0
Output: 0 0 0 1 0 0 1 0 0 0 0
```

- Step 1: Draw finite state machine of the desired function

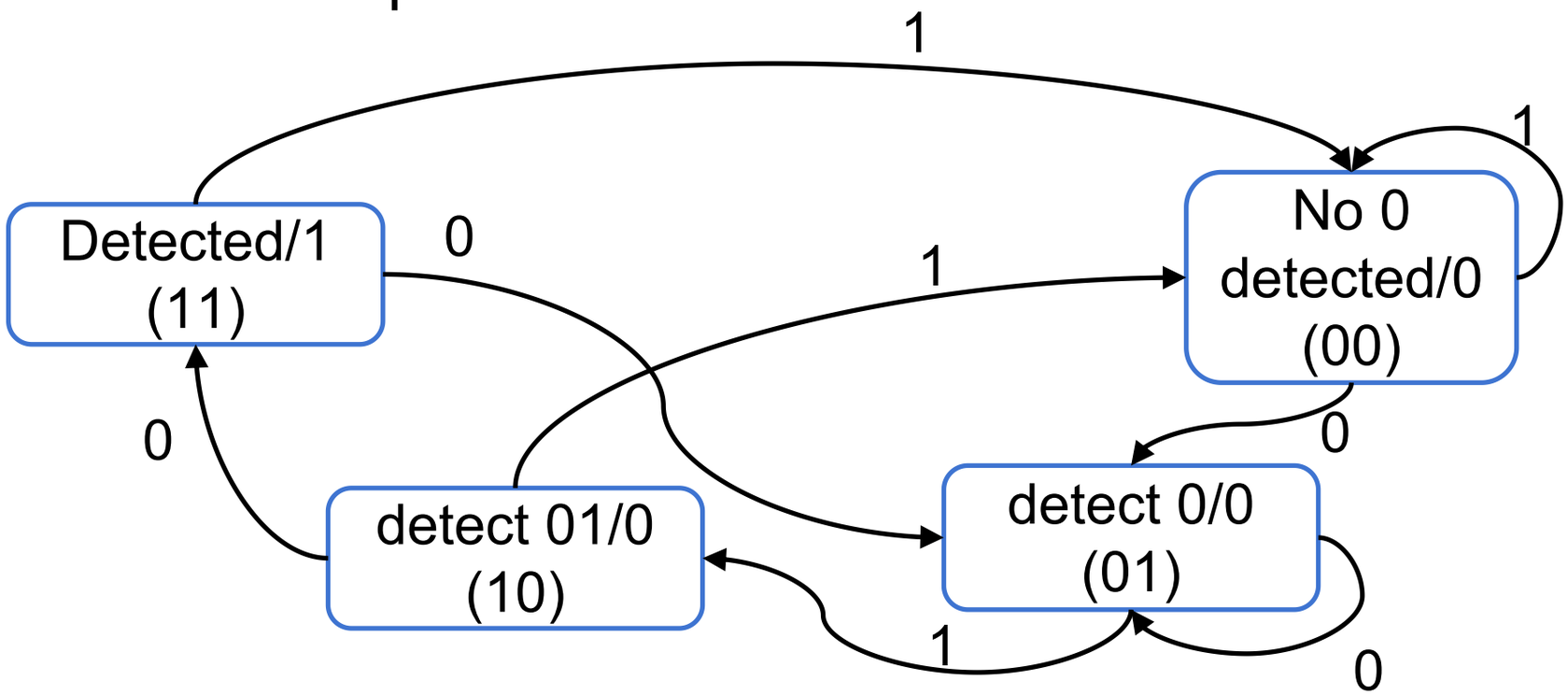


Warm-up

- A classic problem: sequence detection for “010” (non-overlapping)

```
Input:  0 1 0 0 1 0 1 0 1 1 0
Output: 0 0 0 1 0 0 1 0 0 0 0
```

- Step 2: Define/assign binary numbers to represent the states, the inputs and the outputs

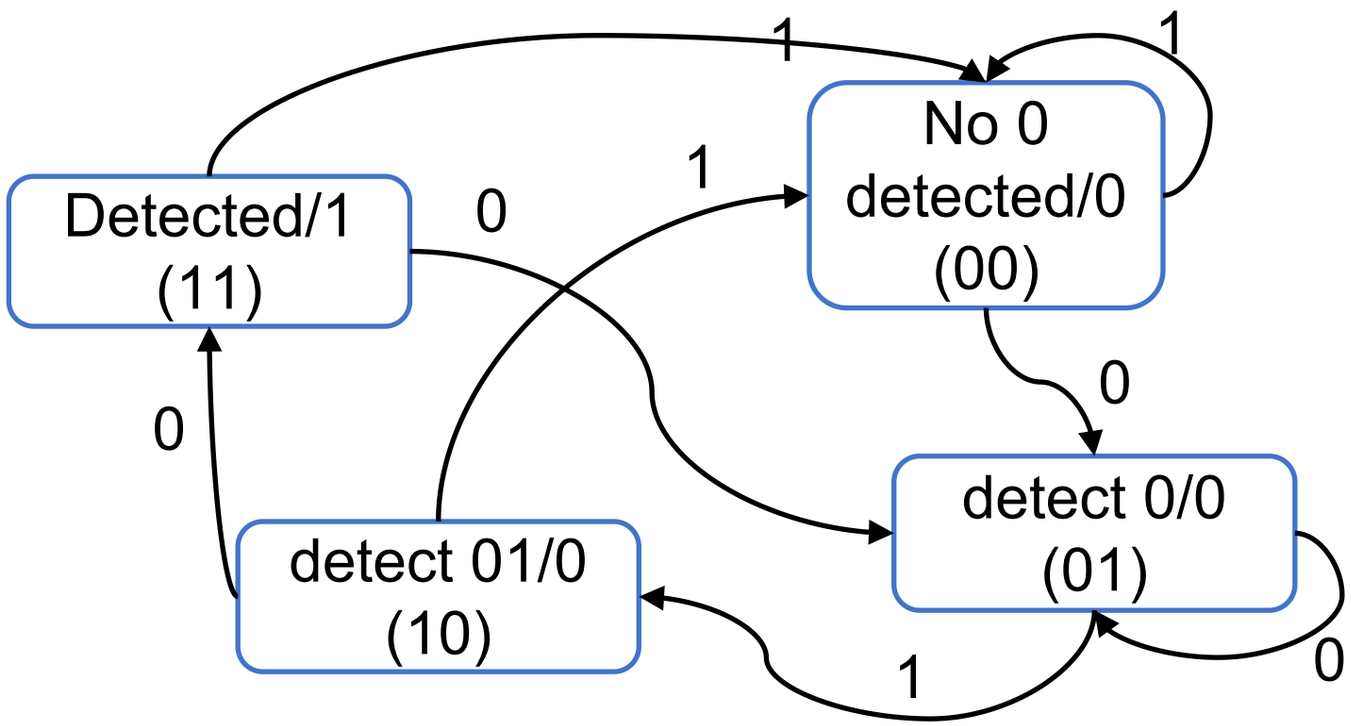


Warm-up

- A classic problem: sequence detection for “010” (non-overlapping)

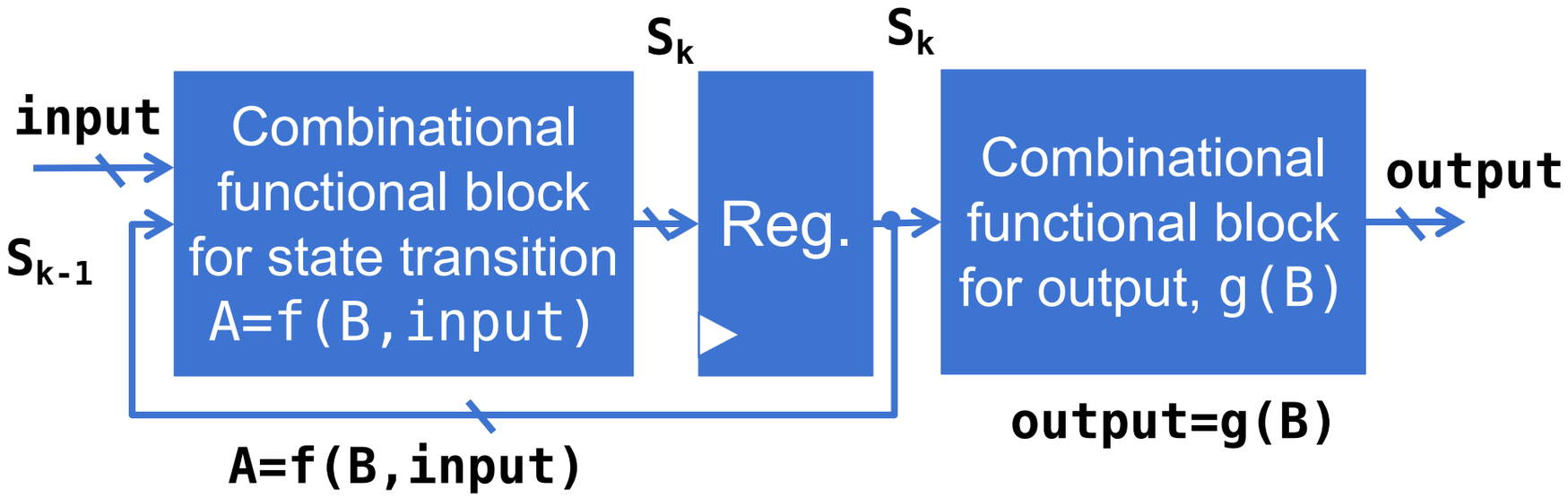
```
Input: 0 1 0 0 1 0 1 0 1 1 0
Output: 0 0 0 1 0 0 1 0 0 0 0
```

- Step 3: Write down the truth table (enumerate input/previous state (and current state) and their corresponding current state (and output))

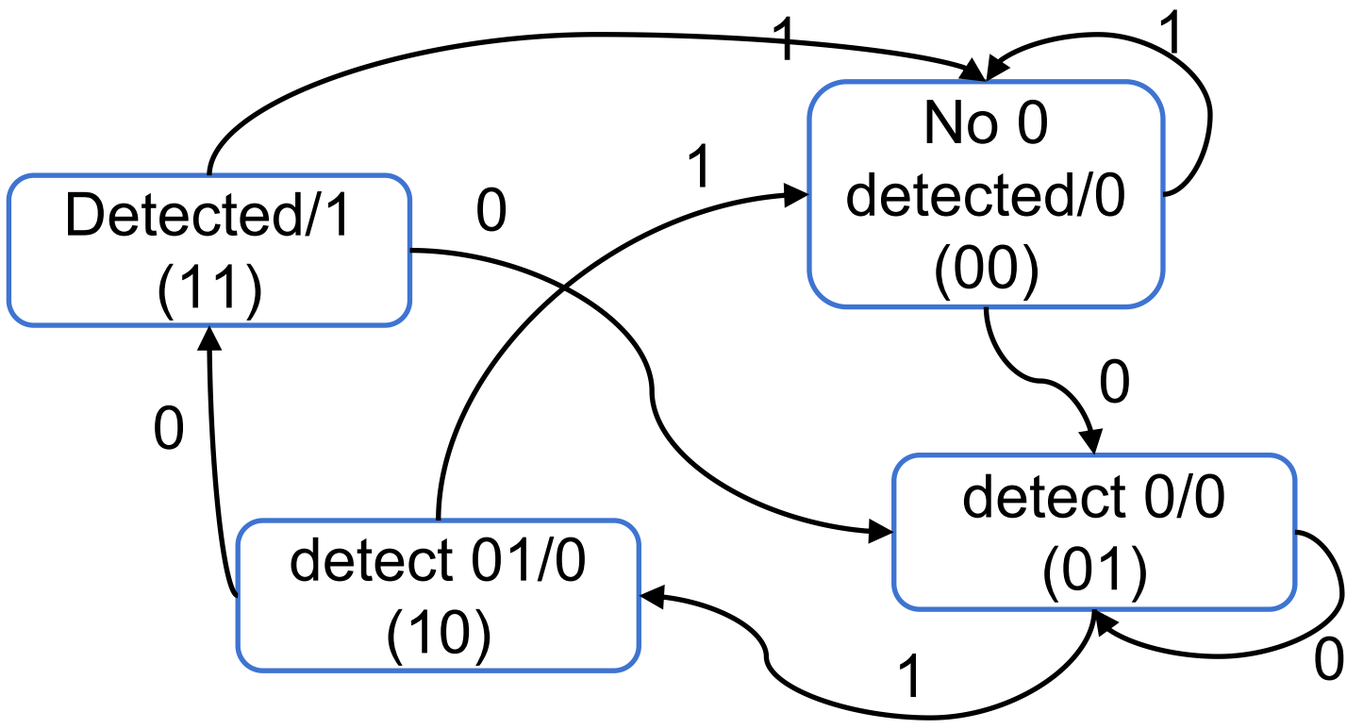


	Previous state		Current state		
input	S[1] k-1	S[0] k-1	S[1] k	S[0] k	output
0	0	0	0	1	0
0	0	1	0	1	0
0	1	0	1	1	1
0	1	1	0	1	0
1	0	0	0	0	0
1	0	1	1	0	0
1	1	0	0	0	0
1	1	1	0	0	0

Warm-up

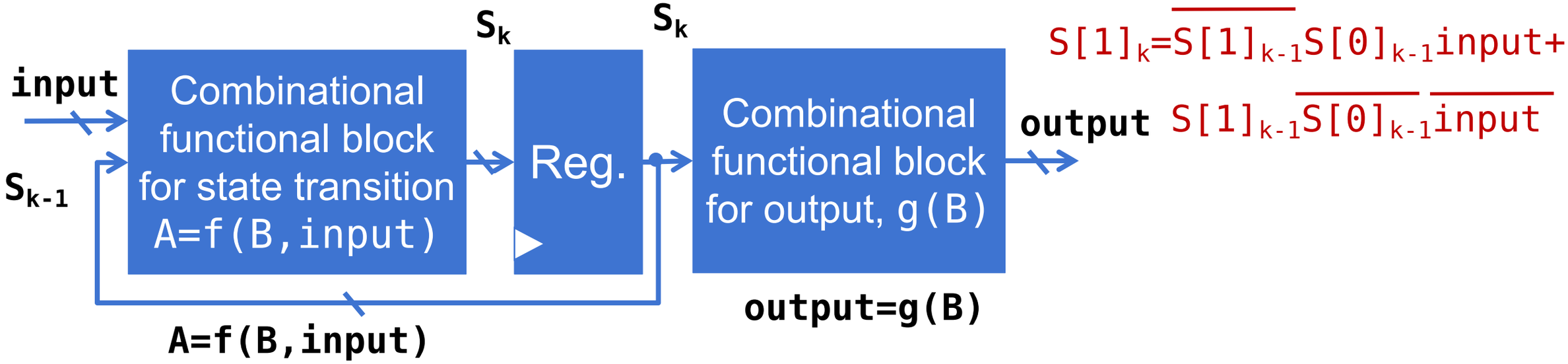


- Step 4: Use template and decide the combinational block for state transition and output logic

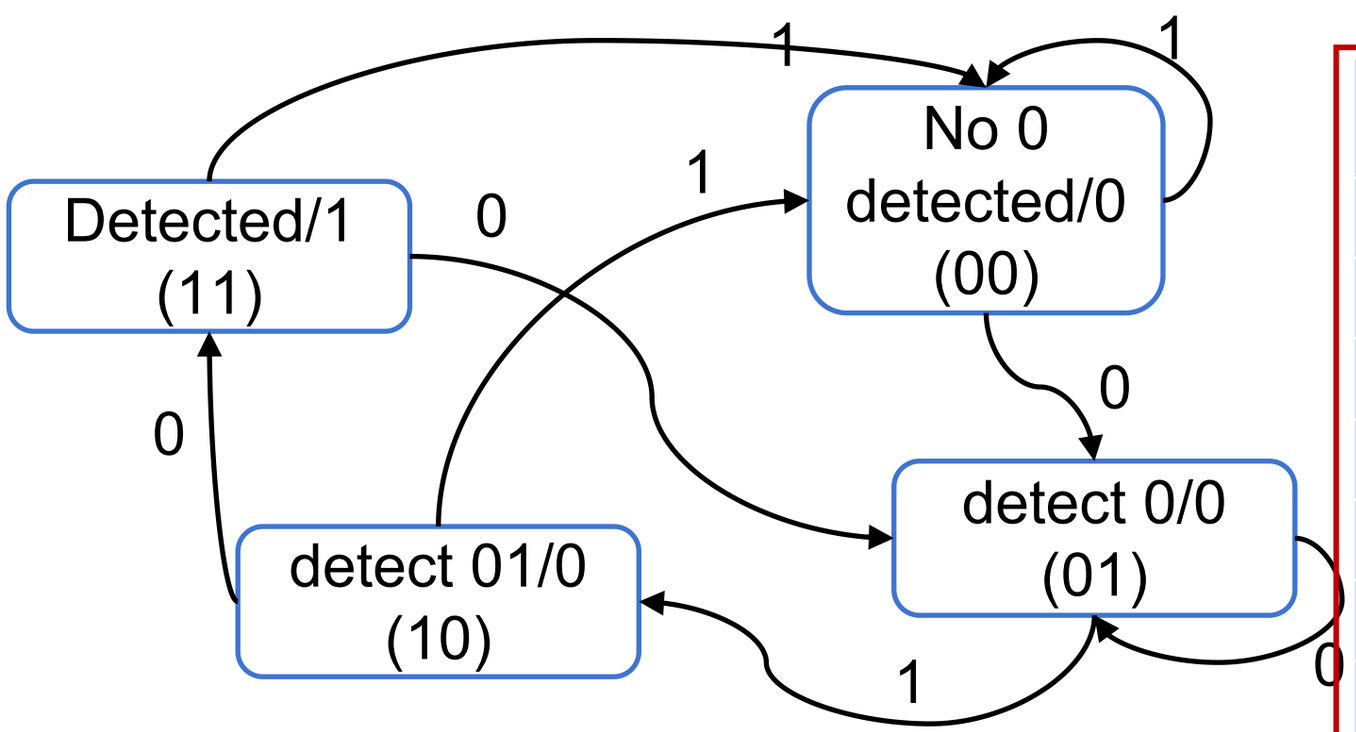


	Previous state		Current state		
input	S[1]$_{k-1}$	S[0]$_{k-1}$	S[1]$_k$	S[0]$_k$	output
0	0	0	0	1	0
0	0	1	0	1	0
0	1	0	1	1	1
0	1	1	0	1	0
1	0	0	0	0	0
1	0	1	1	0	0
1	1	0	0	0	0
1	1	1	0	0	0

Warm-up

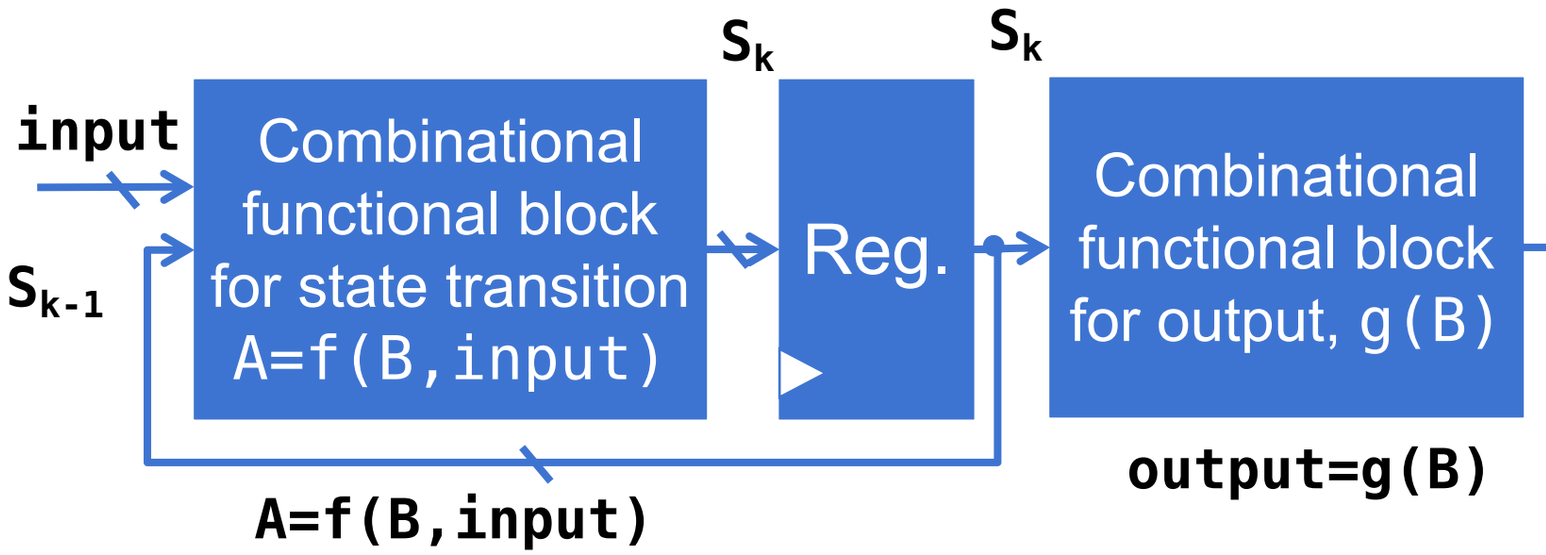


- Step 4: Use template and decide the combinational block for state transition and output logic



	Previous state		Current state		
input	S[1] k-1	S[0] k-1	S[1] k	S[0] k	output
0	0	0	0	1	0
0	0	1	0	1	0
0	1	0	1	1	1
0	1	1	0	1	0
1	0	0	0	0	0
1	0	1	1	0	0
1	1	0	0	0	0
1	1	1	0	0	0

Warm-up

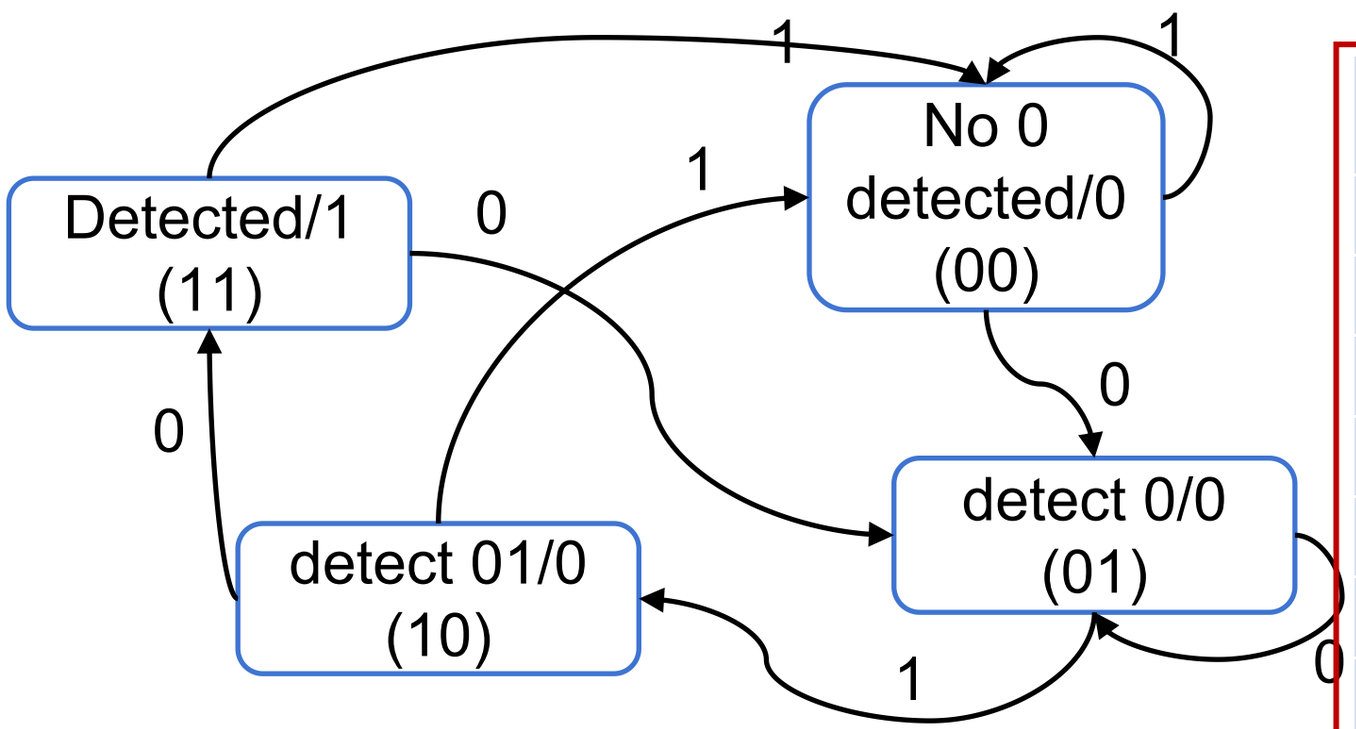


$output=S[1]_kS[0]_k$

$S[1]_k=\overline{S[1]_{k-1}}S[0]_{k-1}input+S[1]_{k-1}\overline{S[0]_{k-1}}\,\overline{input}$

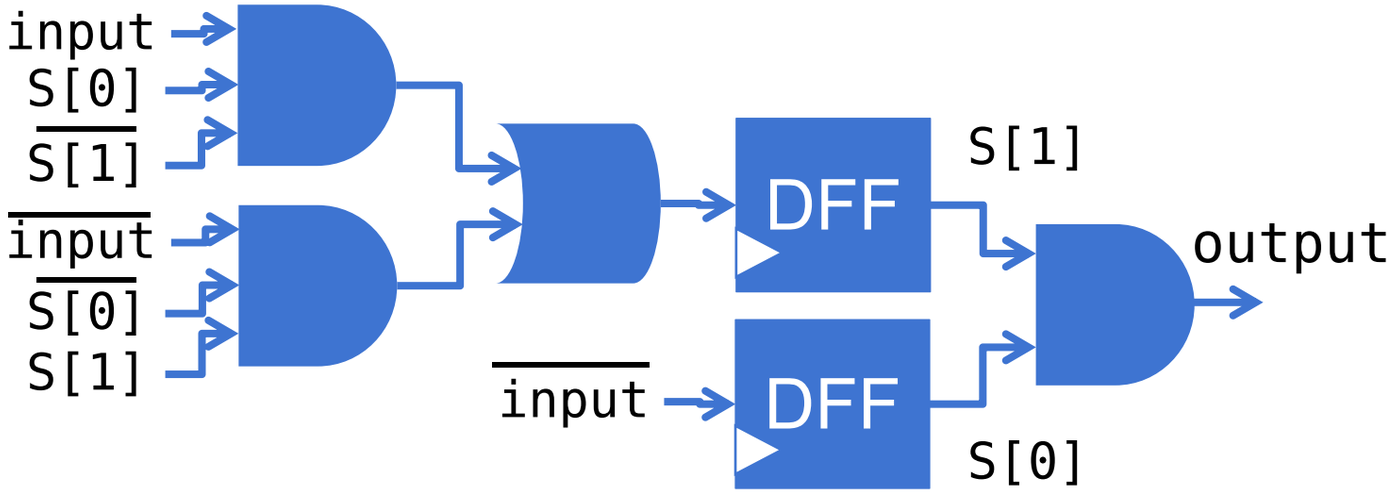
$S[0]_k=\overline{input}$

- Step 4: Use template and decide the combinational block for state transition and output logic



	Previous state		Current state		
input	S[1] k-1	S[0] k-1	S[1] k	S[0] k	output
0	0	0	0	1	0
0	0	1	0	1	0
0	1	0	1	1	1
0	1	1	0	1	0
1	0	0	0	0	0
1	0	1	1	0	0
1	1	0	0	0	0
1	1	1	0	0	0

Warm-up

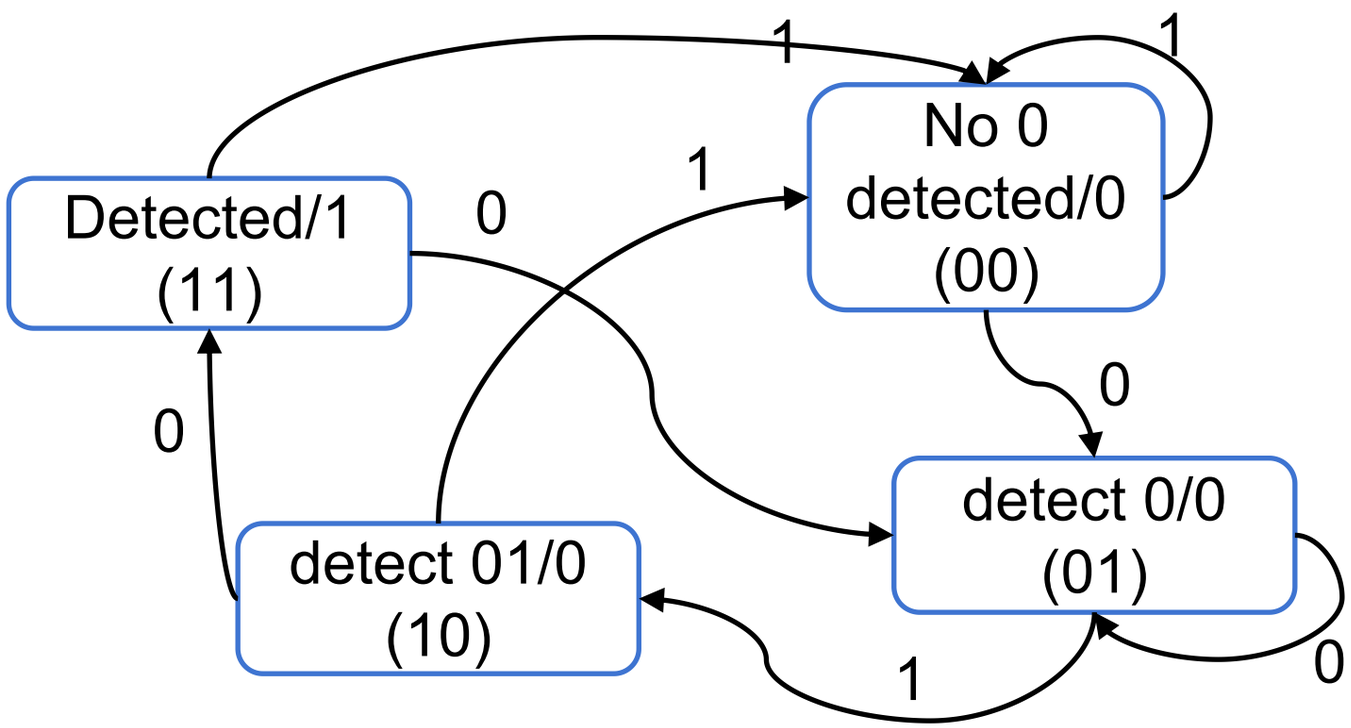


$$output = S[1]_k S[0]_k$$

$$S[1]_k = \overline{S[1]}_{k-1} S[0]_{k-1} input + S[1]_{k-1} \overline{S[0]}_{k-1} \overline{input}$$

$$S[0]_k = \overline{input}$$

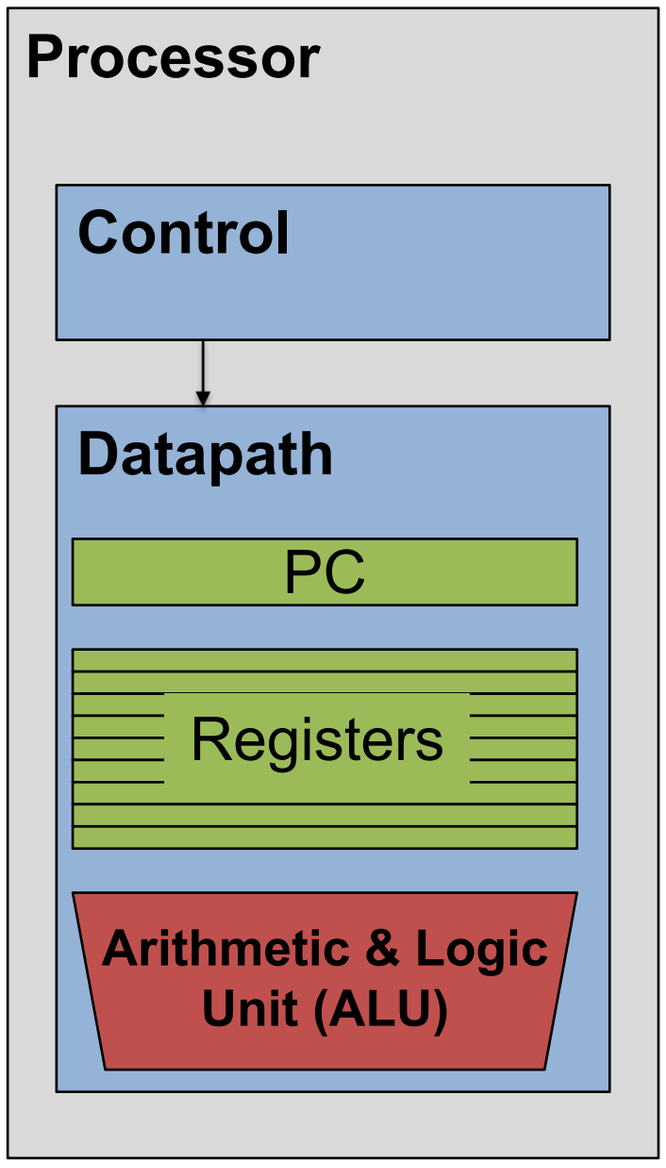
- Step 4: Use template and decide the combinational block for state transition and output logic



	Previous state		Current state		
input	S[1] k-1	S[0] k-1	S[1] k	S[0] k	output
0	0	0	0	1	0
0	0	1	0	1	0
0	1	0	1	1	1
0	1	1	0	1	0
1	0	0	0	0	0
1	0	1	1	0	0
1	1	0	0	0	0
1	1	1	0	0	0

Controller & Datapath

- A CPU that support RV32I can have so many states.

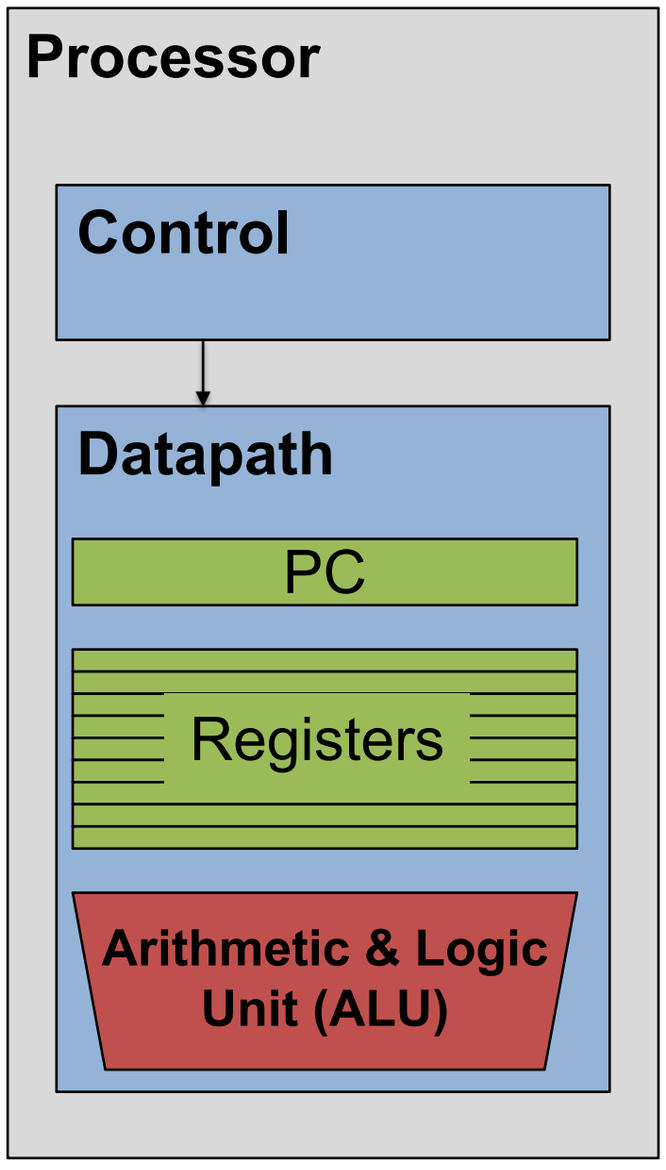


- Consider the 32 registers alone
  - `x0` always 0;
  - Each bit in the other registers can be 0 or 1;
- Not practical to enumerate all the state transitions;
- Top-down design: build small modules and then connect them as required;
- Most digital systems can be divided into datapth and controller:
  - Datapath contains data processing and storage;
  - Controller controls data access (can be modeled as FSM);
- Recall the execution of an instruction

- Our Goal: Implement a RISC-V processor as a synchronous digital system (SDS).
- Each RV32I instruction can be done within 1 clock cycle (single-cycle CPU).

Controller & Datapath

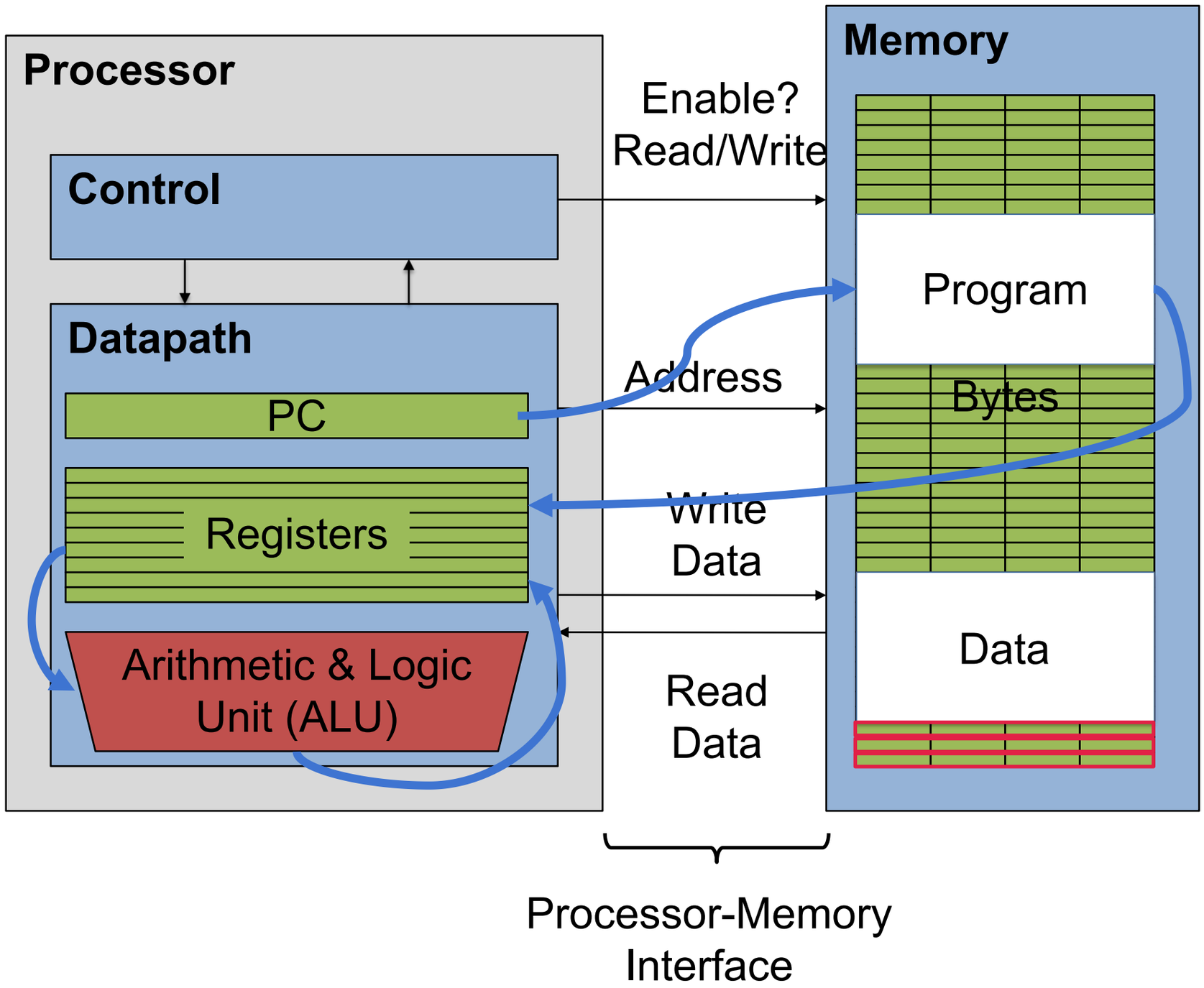
- A CPU that support RV32I can have so many states



- **Datapath**
  - Start with basic building blocks
  - Add building blocks to the digital system with added supported instructions
- **Controller**
  - Can be considered as an FSM

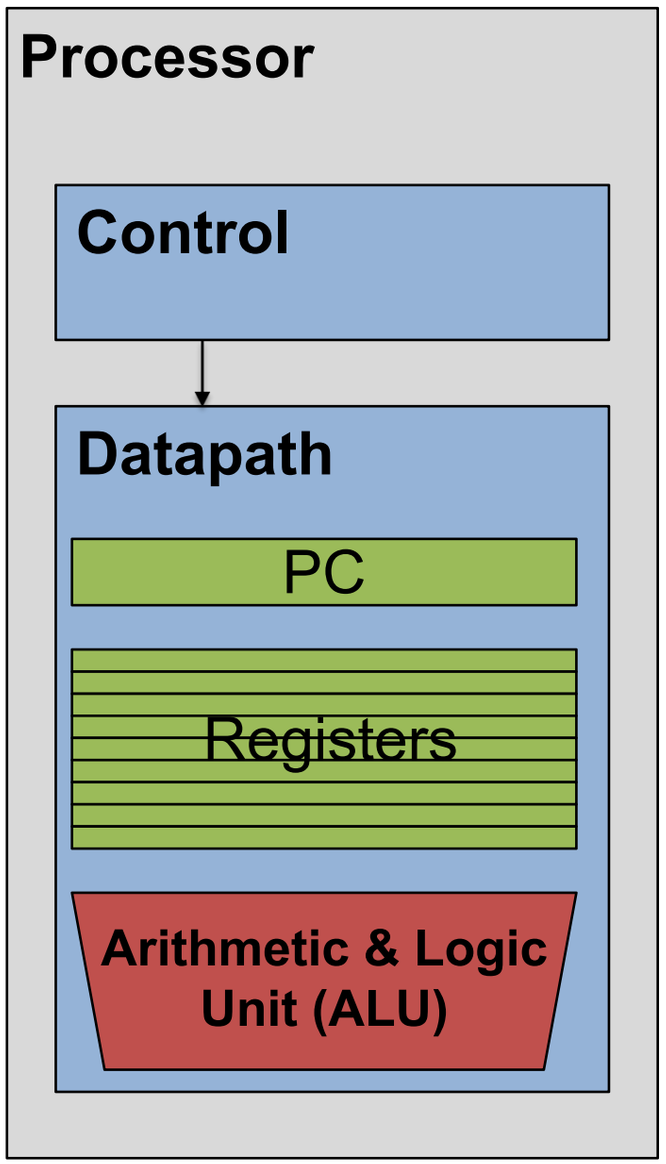
- Our Goal: Implement a RISC-V processor as a synchronous digital system (SDS).
- Each RV32I instruction can be done within 1 clock cycle (single-cycle CPU).

Datapath


Processor
Control
Datapath
PC
Registers
Arithmetic & Logic Unit (ALU)
Memory
Program
Bytes
Data
Enable?
Read/Write
Address
Write Data
Read Data
Processor-Memory Interface

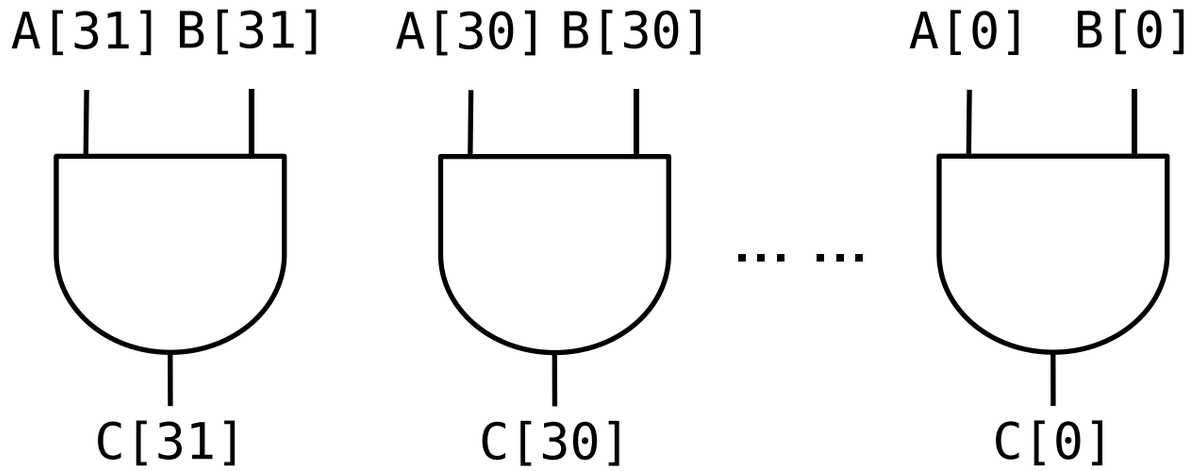
Useful building blocks

- An ALU should be able to execute all the arithmetic and logic operations



ADD	ADDI
SUB	SLTI
SLL	SLTIU
SLT	XORI
SLTU	ORI
XOR	ANDI
SRL	
SRA	
OR	
AND	

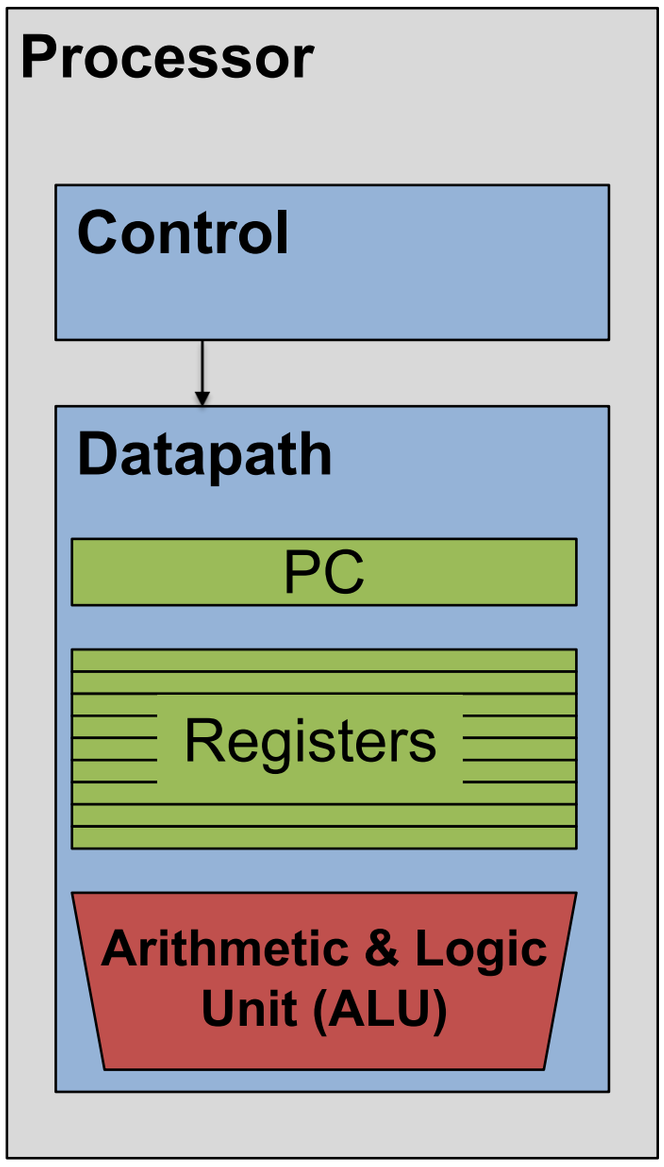
- AND as an example
  - 2 32-bit inputs A and B
  - 1 32-bit output C



- Our Goal: Implement a RISC-V processor as a synchronous digital system.
- Each RV32I instruction can be done within 1 clock cycle.

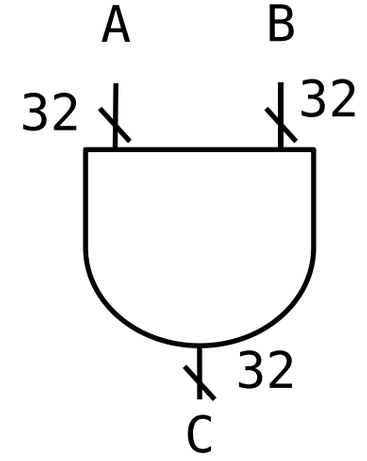
Useful building blocks

- An ALU should be able to execute all the arithmetic and logic operations

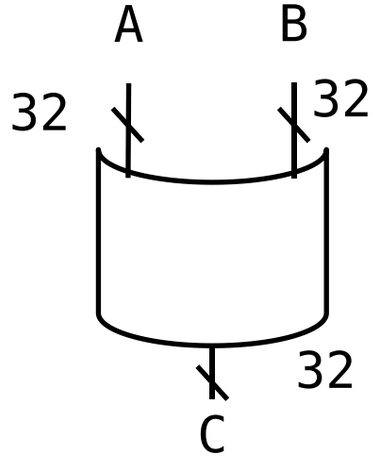


ADD	ADDI
SUB	SLTI
SLL	SLTIU
SLT	XORI
SLTU	ORI
XOR	ANDI
SRL	
SRA	
OR	
AND	

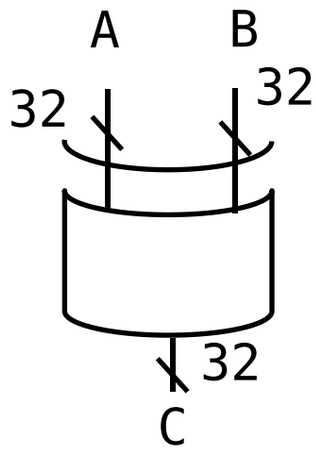
- AND as an example
  - 2 32-bit inputs A and B
  - 1 32-bit output C



A simplified AND gate array symbol



OR

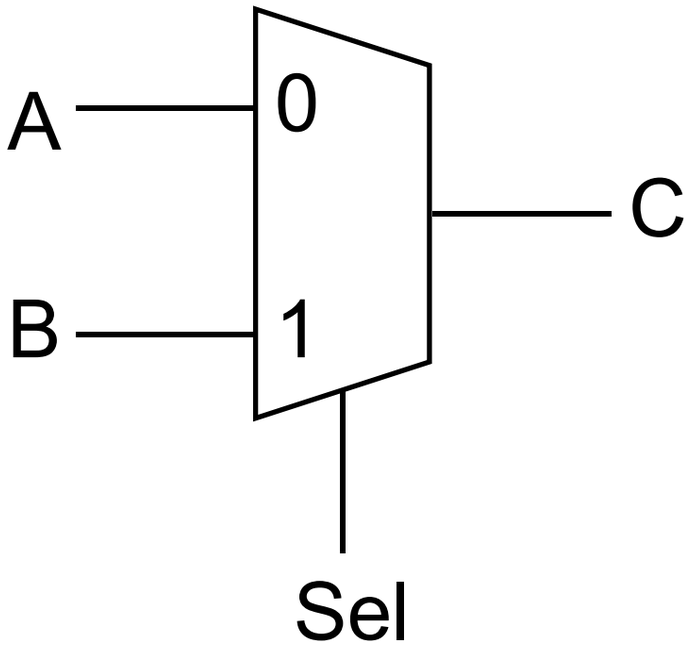


XOR

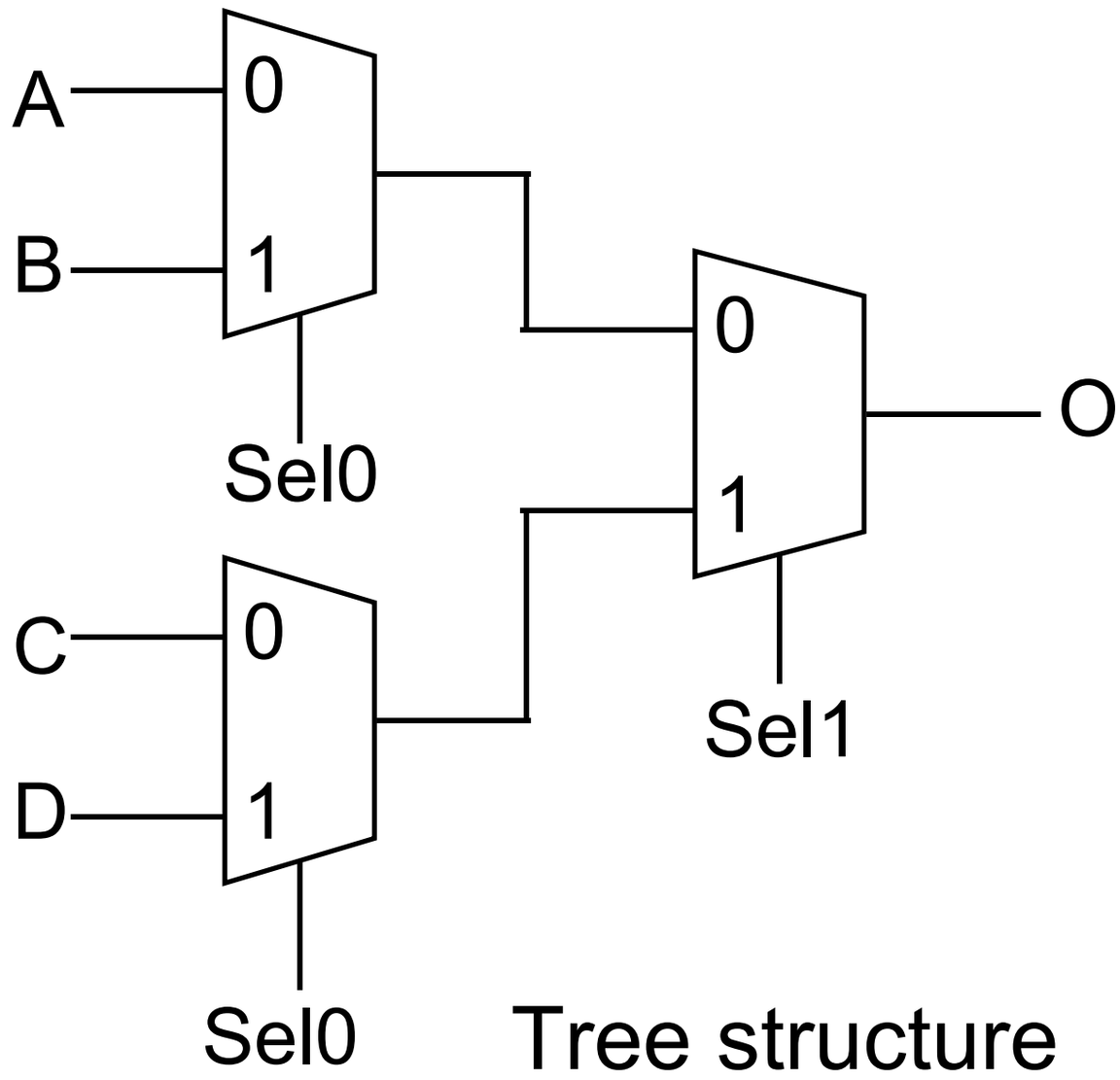
- Our Goal: Implement a RISC-V processor as a synchronous digital system.
- Each RV32I instruction can be done within 1 clock cycle.

Useful Combinational Circuits

- Multiplexer (2-to-1)



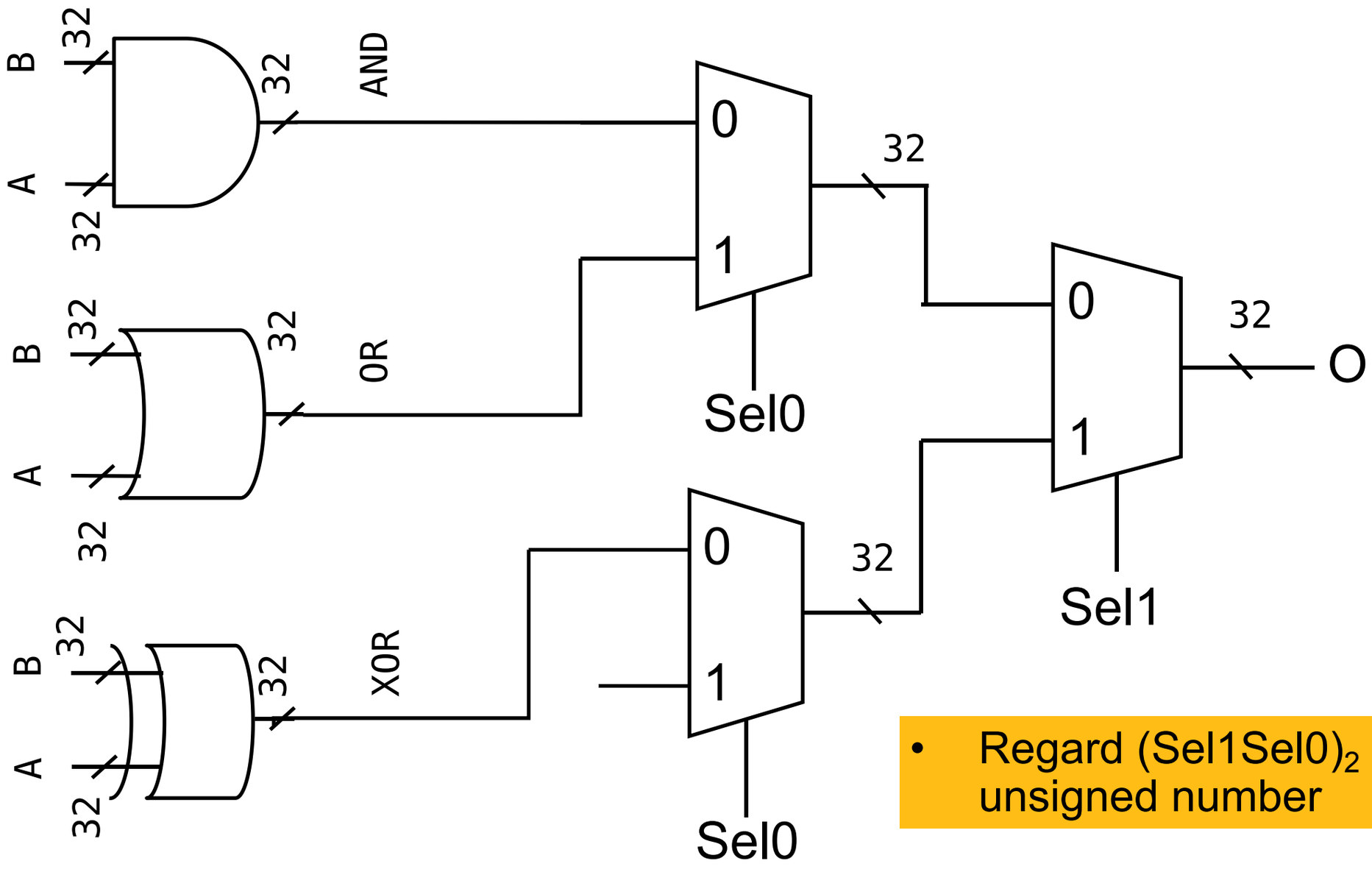
- Multiplexer (2^n-to-1)



Tree structure

Control through selection

- 32-bit Multiplexer and logic gates to support some logic instructions

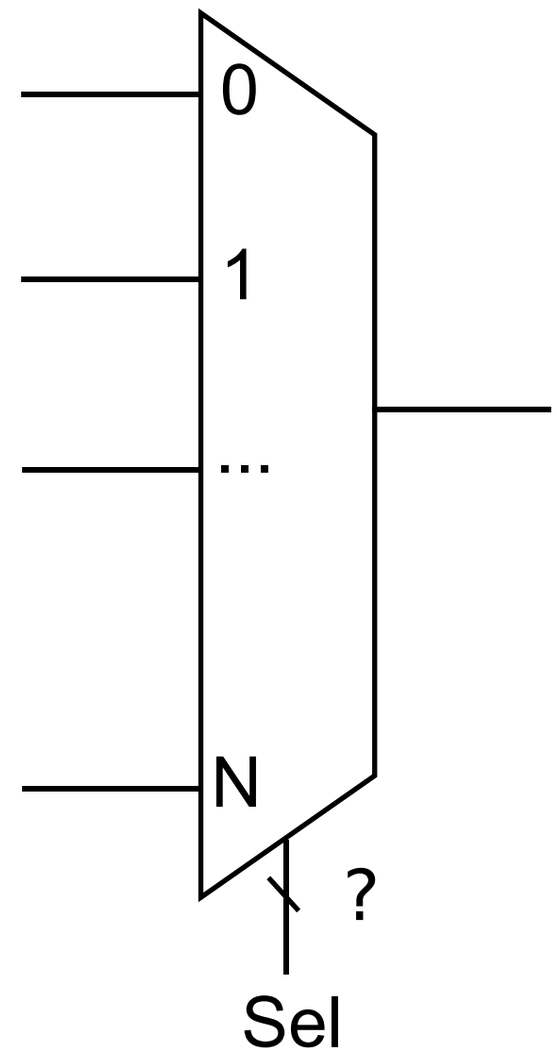


- Regard $(Sel1Sel0)_2$ as an unsigned number

- More layers of multiplexer to select from more inputs

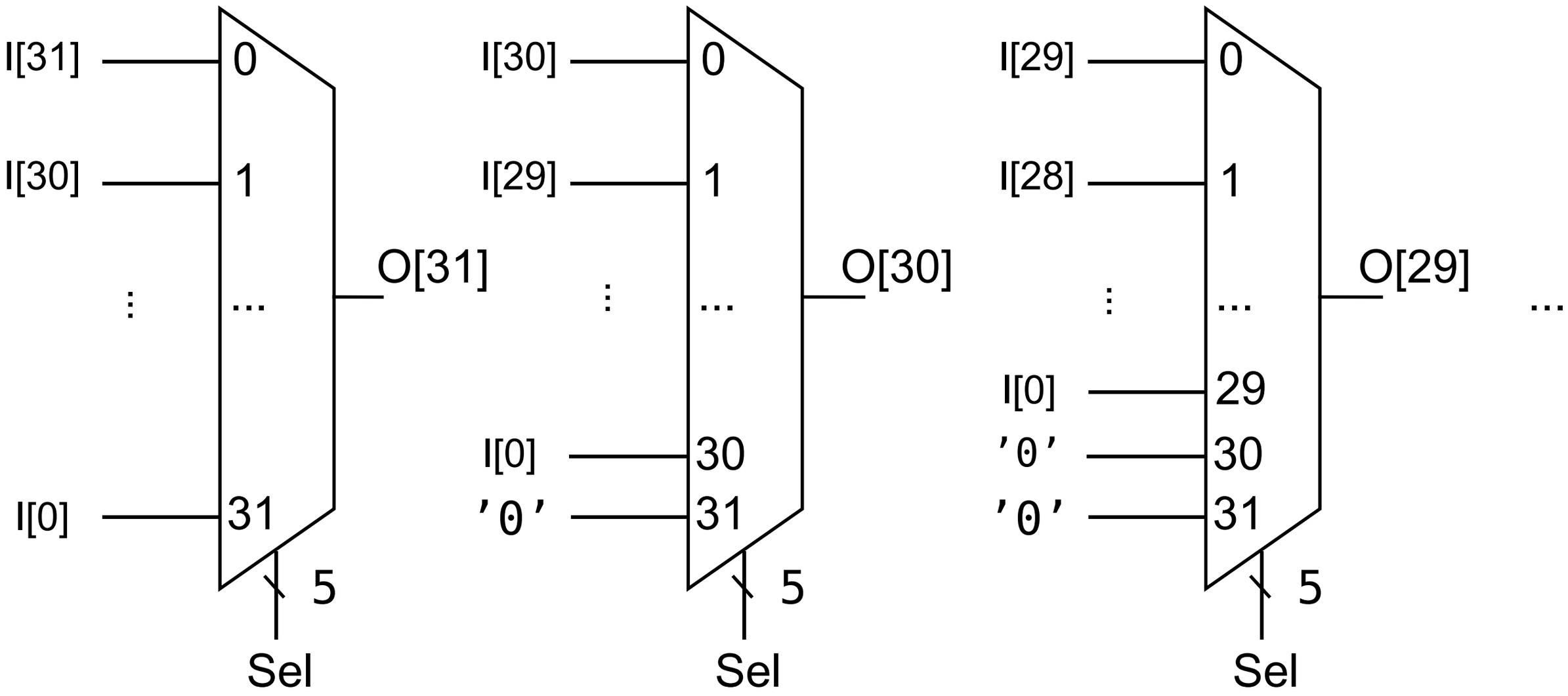
Multiplexer

- N-to-1 multiplexer symbol



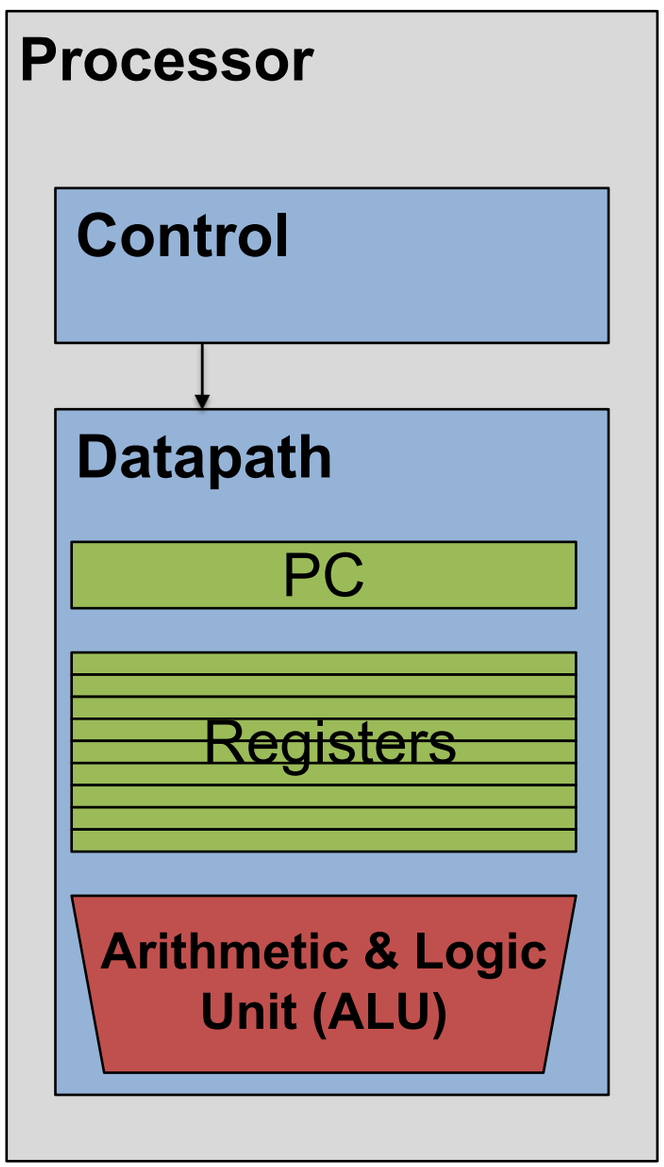
Multiplexers used for shifter

- Left shift a single bit -> left shift multiple single bits
- Other shifter designs such as barrel shifter



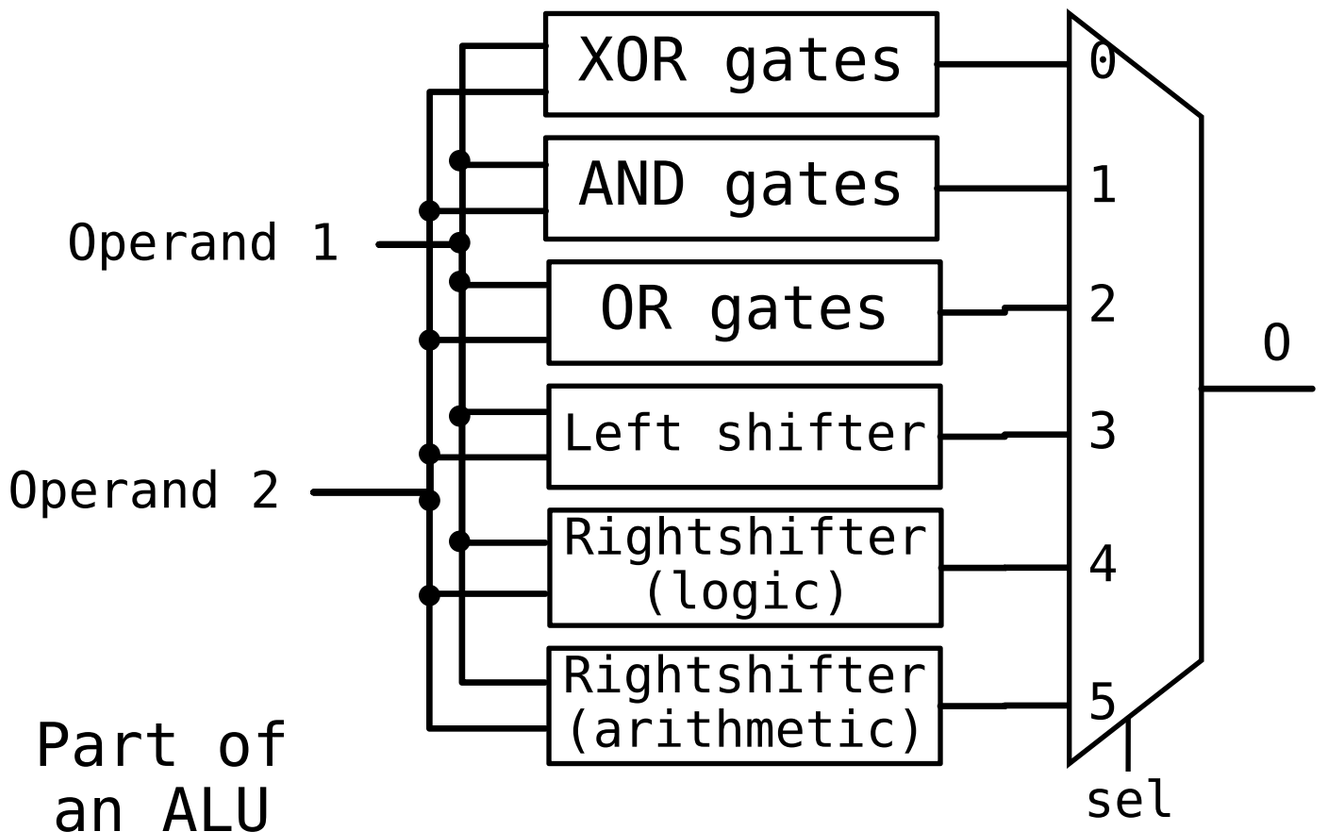
Useful building blocks

- An ALU should be able to execute all the arithmetic and logic operations



ADD
SUB
SLL
SLT
SLTU
XOR
SRL
SRA
OR
AND

ADDI
SLTI
SLTIU
XORI
ORI
ANDI

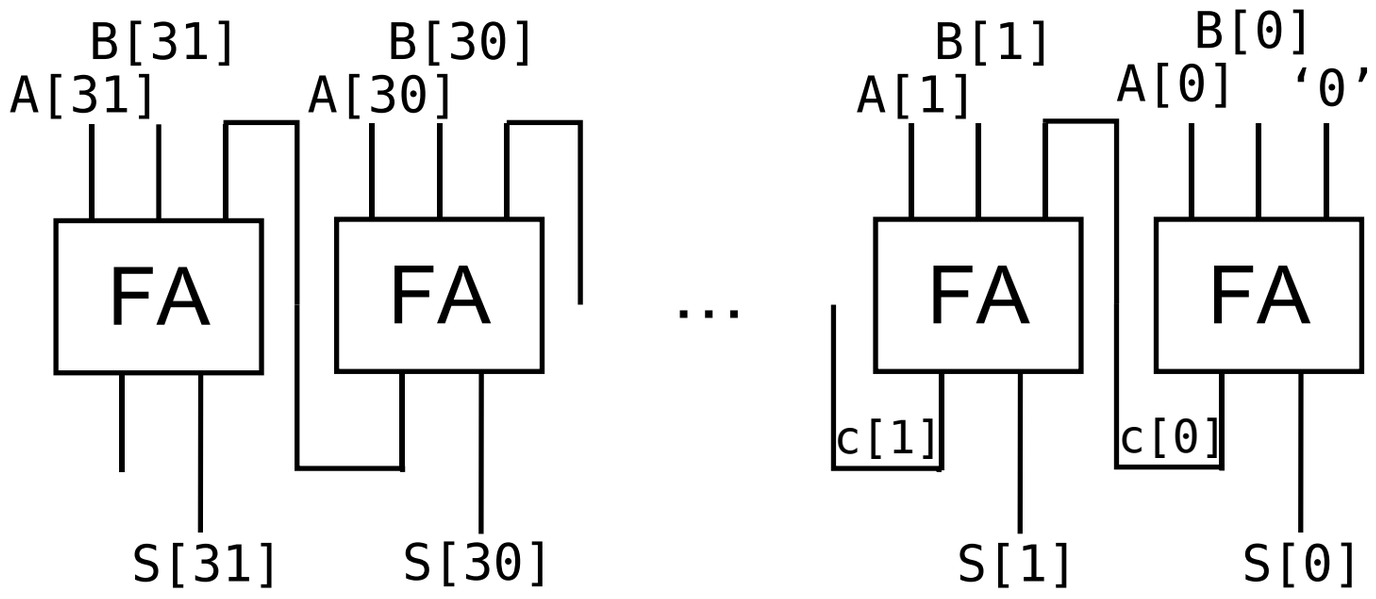


Note that all the signals expect the selection signals are 32-bit.

- Our Goal: Implement a RISC-V processor as a synchronous digital system.
- Each RV32I instruction can be done within 1 clock cycle.

Adder & subtractor

- An adder design

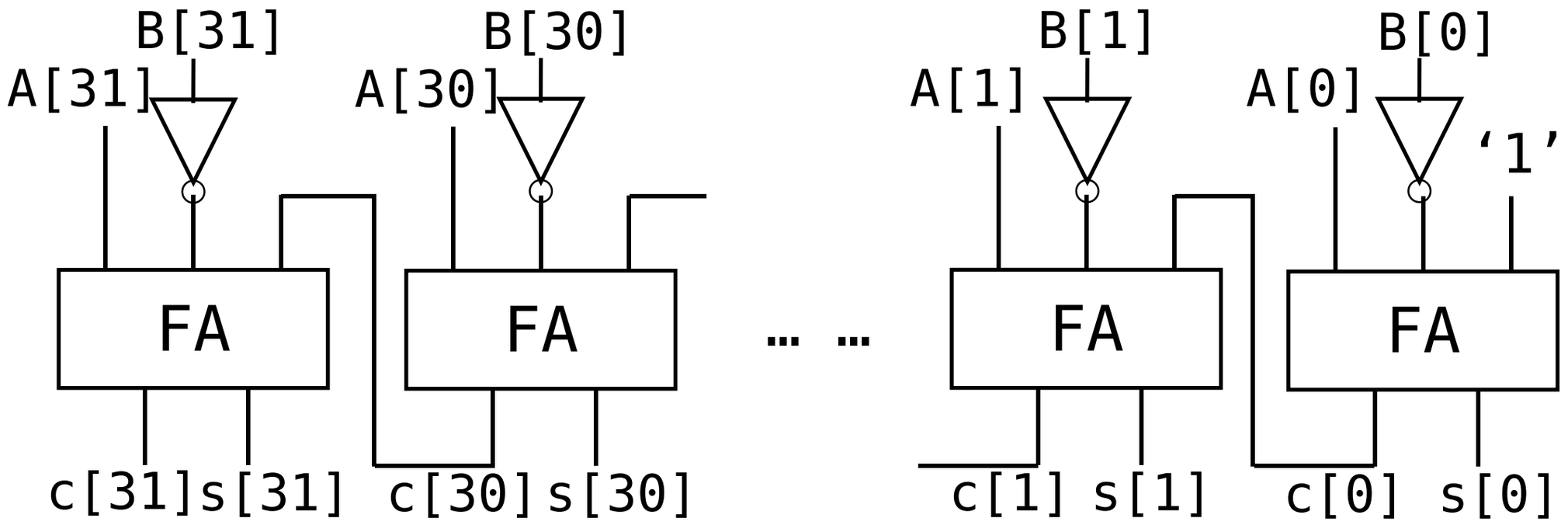


A 32-bit adder

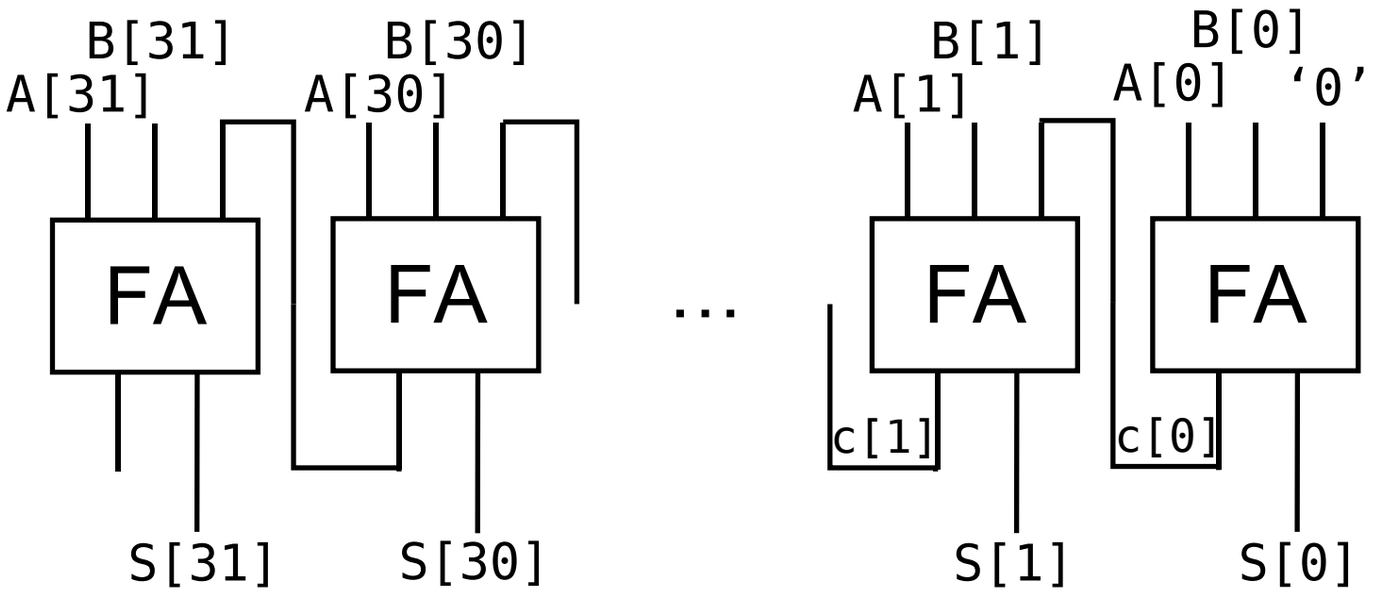
- A smart subtractor design
  - Recall that subtracting a number is equivalent to adding its negative version

A smart subtractor design

$$A - B = A + (-B) = A + \bar{B} + 1 \pmod{2^{N-1}}$$

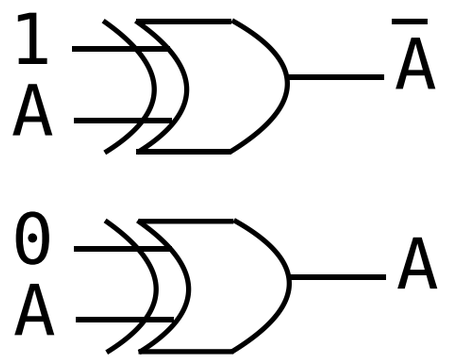


A 32-bit subtractor



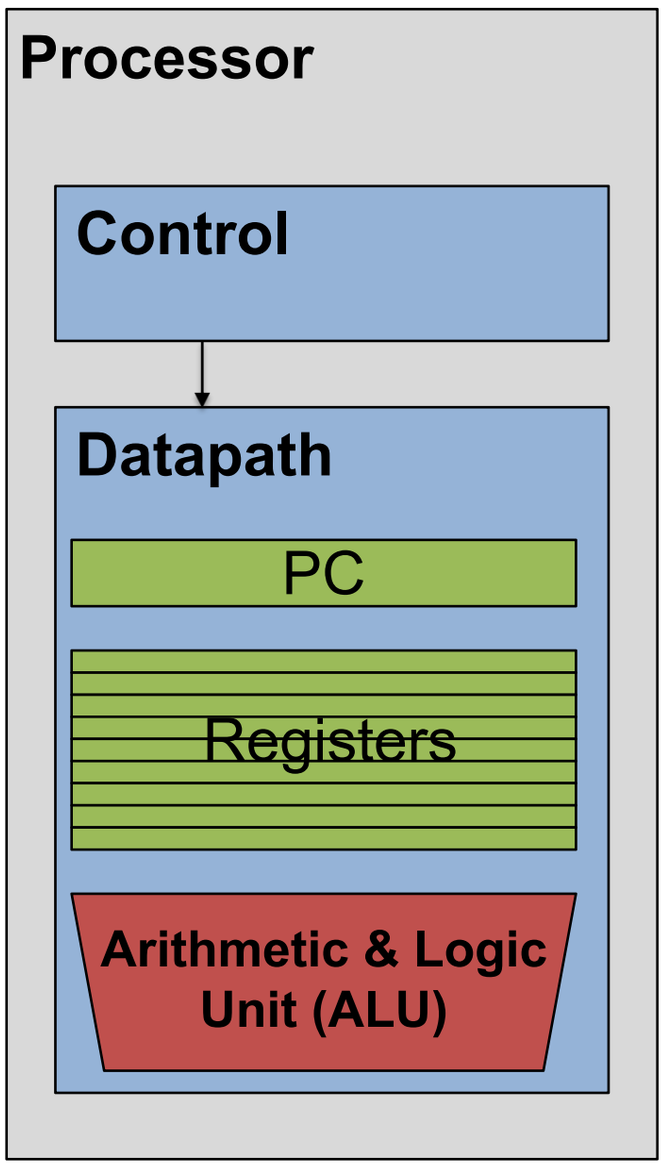
A 32-bit adder

- Recall X0R gate



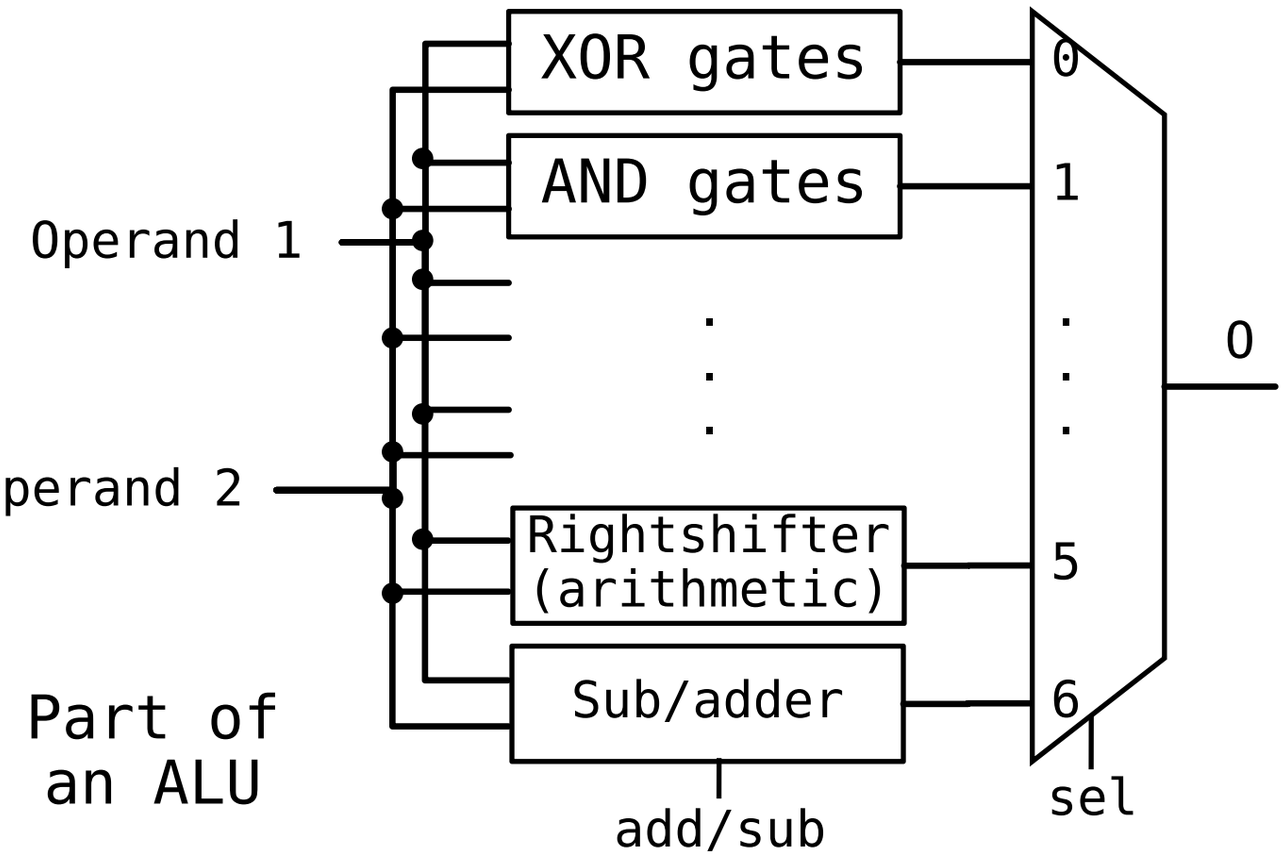
Useful building blocks

- An ALU should be able to execute all the arithmetic and logic operations



ADD
SUB
SLL
SLT
SLTU
XOR
SRL
SRA
OR
AND

ADDI
SLTI
SLTIU
XORI
ORI
ANDI

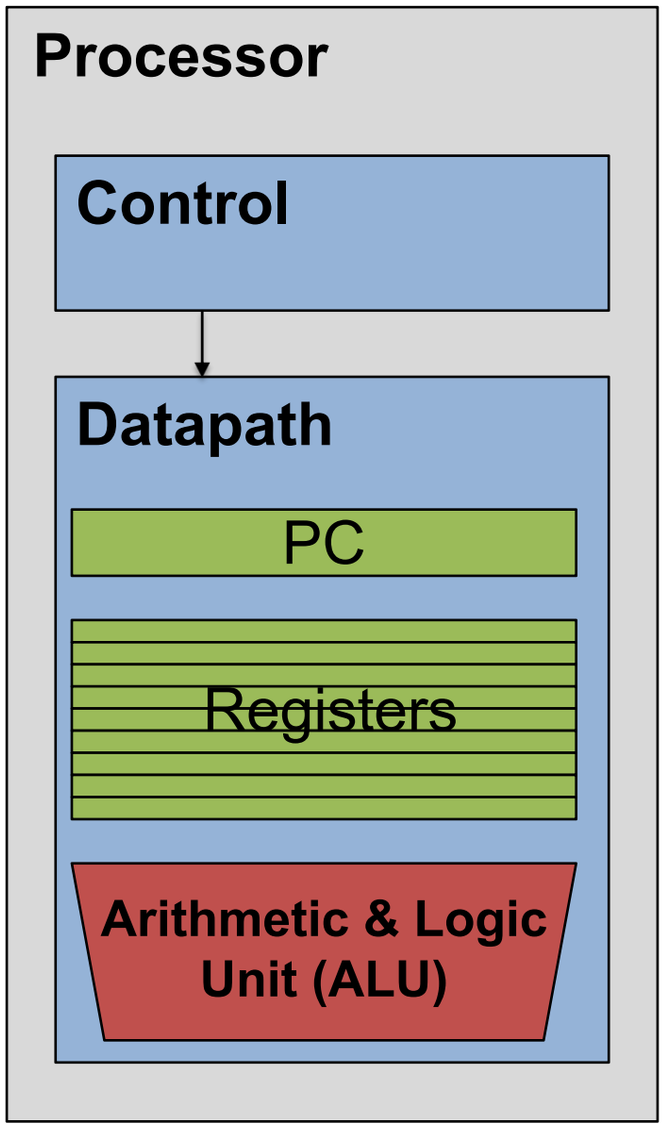


Note that all the signals expect the selection signals are 32-bit.

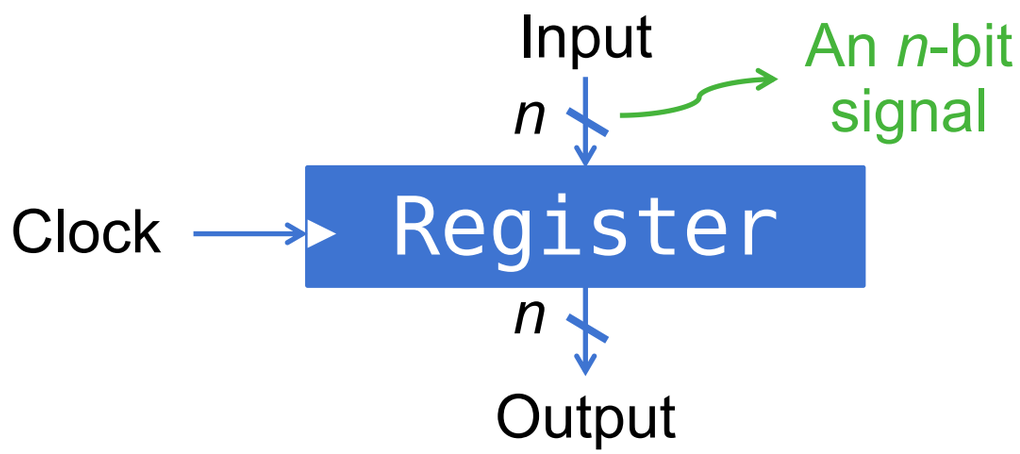
- ALU design that supports R-/I-arithmetic and logic operations completed

Useful building blocks-Register file

- The register file is the component that contains all the general purpose registers of the microprocessor
- A register file should provide data given the register numbers
- A register file should be able to change the stored value

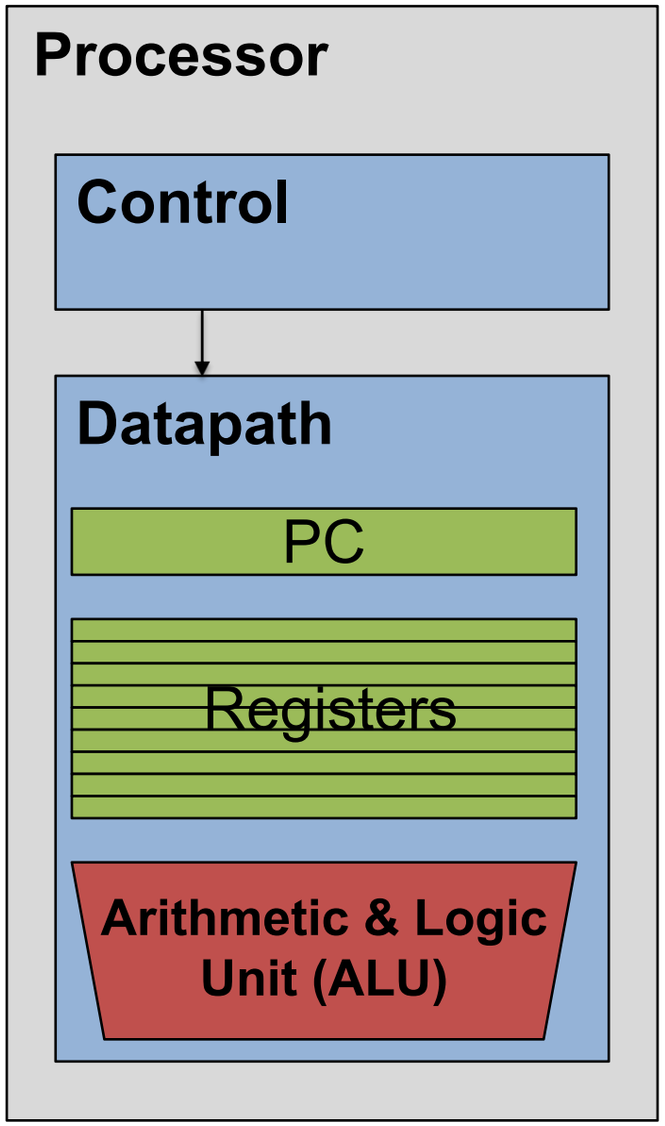


- Recall we have registers that store values

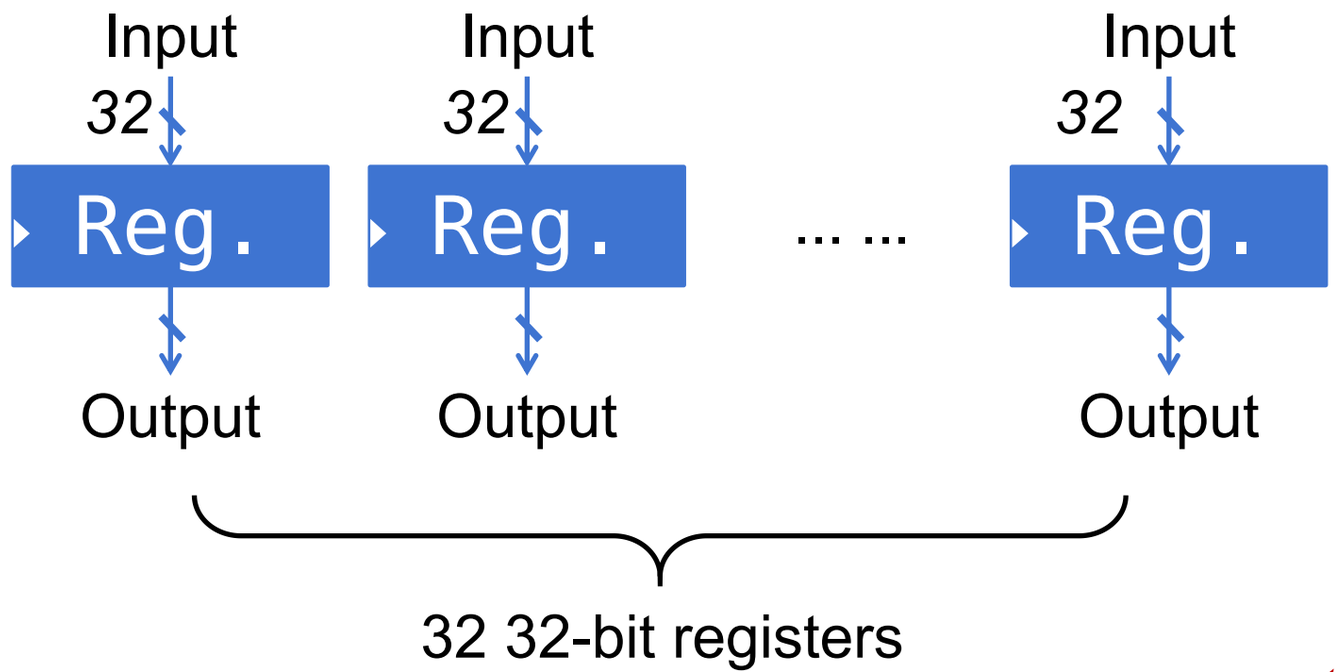


Useful building blocks-Register file

- The register file is the component that contains all the general purpose registers of the microprocessor
- A register file should provide data given the register numbers
- A register file should be able to change the stored value



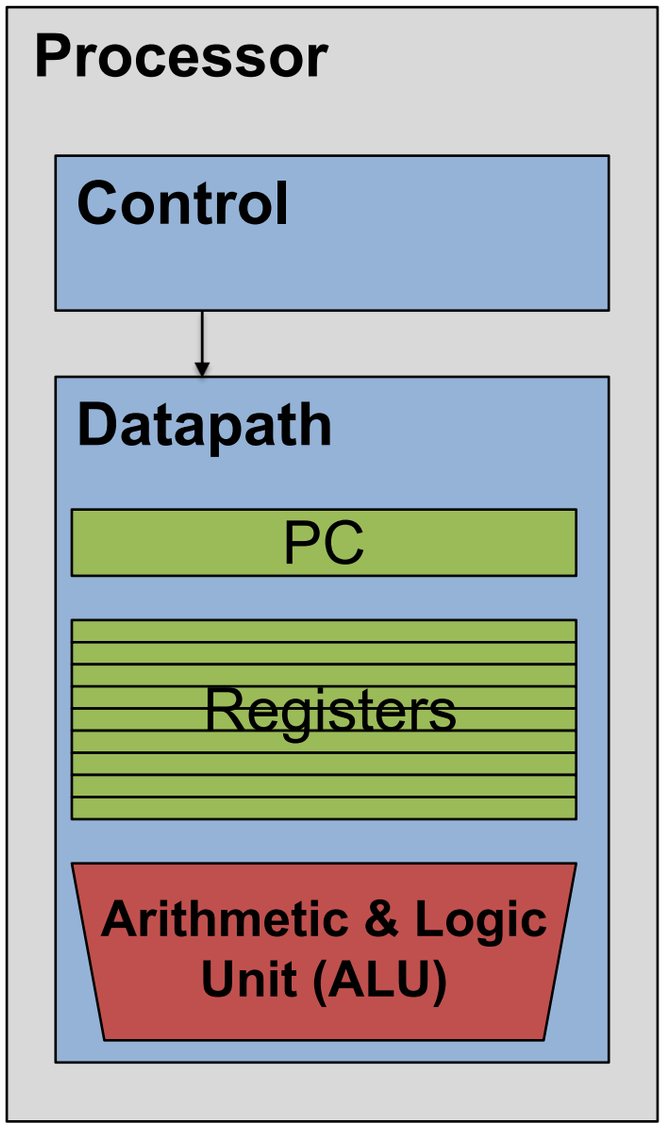
- Recall we have registers that store values



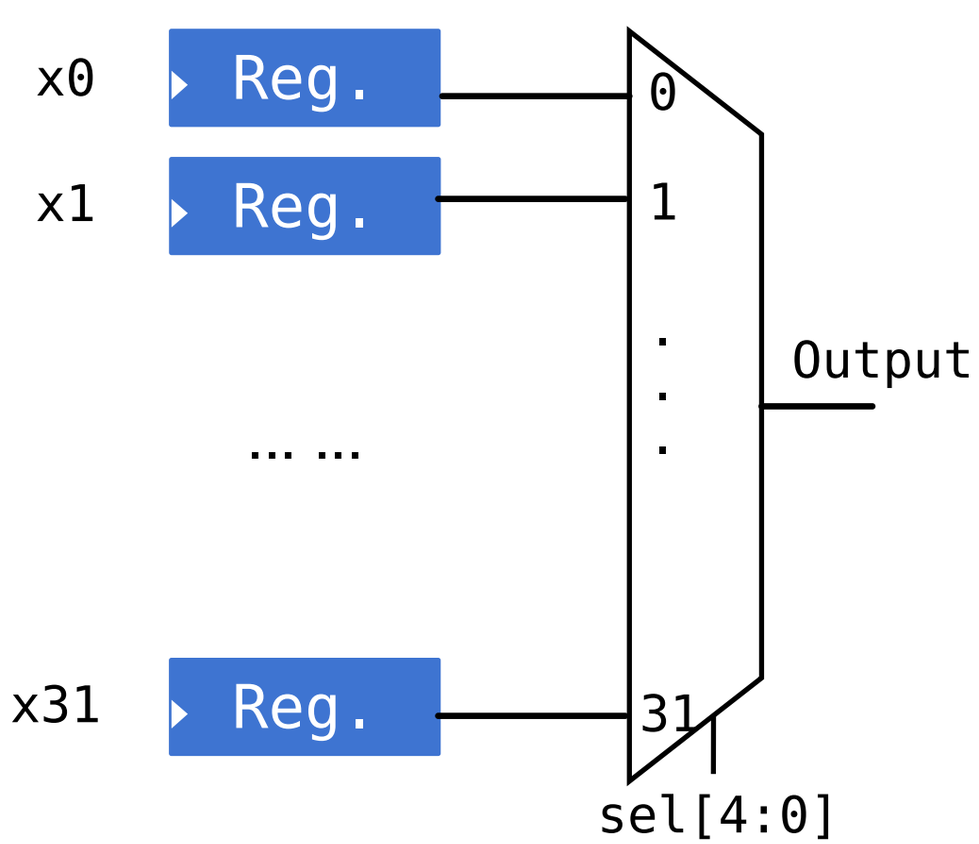
- How to select one to output? Multiplexer

Useful building blocks-Register file

- The register file is the component that contains all the general purpose registers of the microprocessor
- A register file should provide data given the register numbers
- A register file should be able to change the stored value



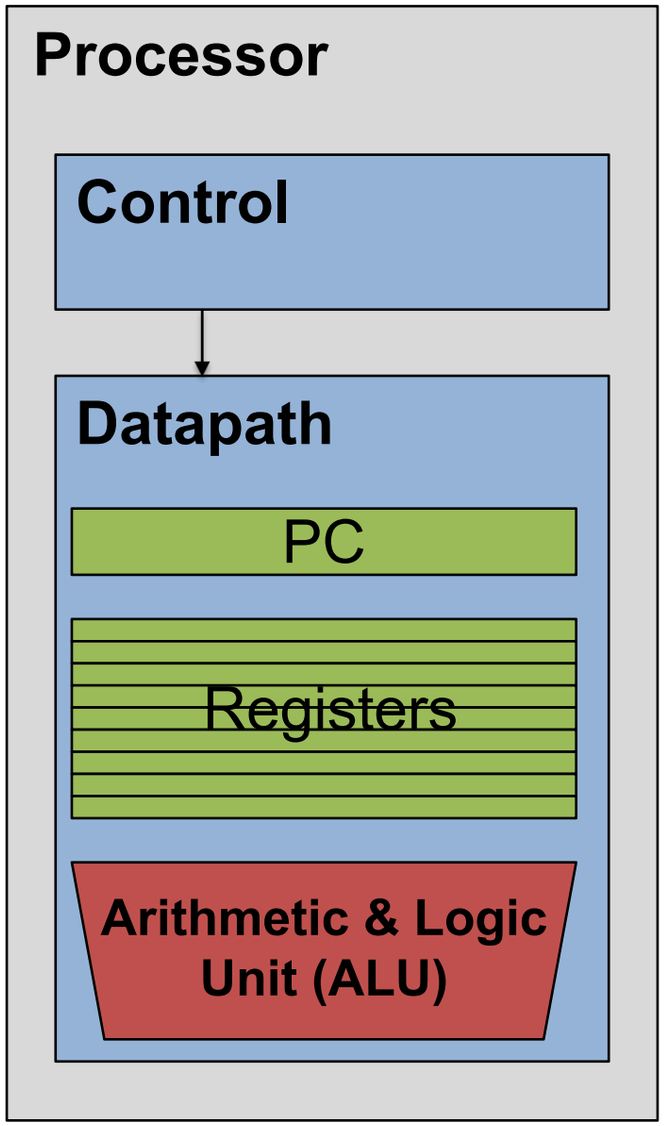
- Recall we have registers that store values



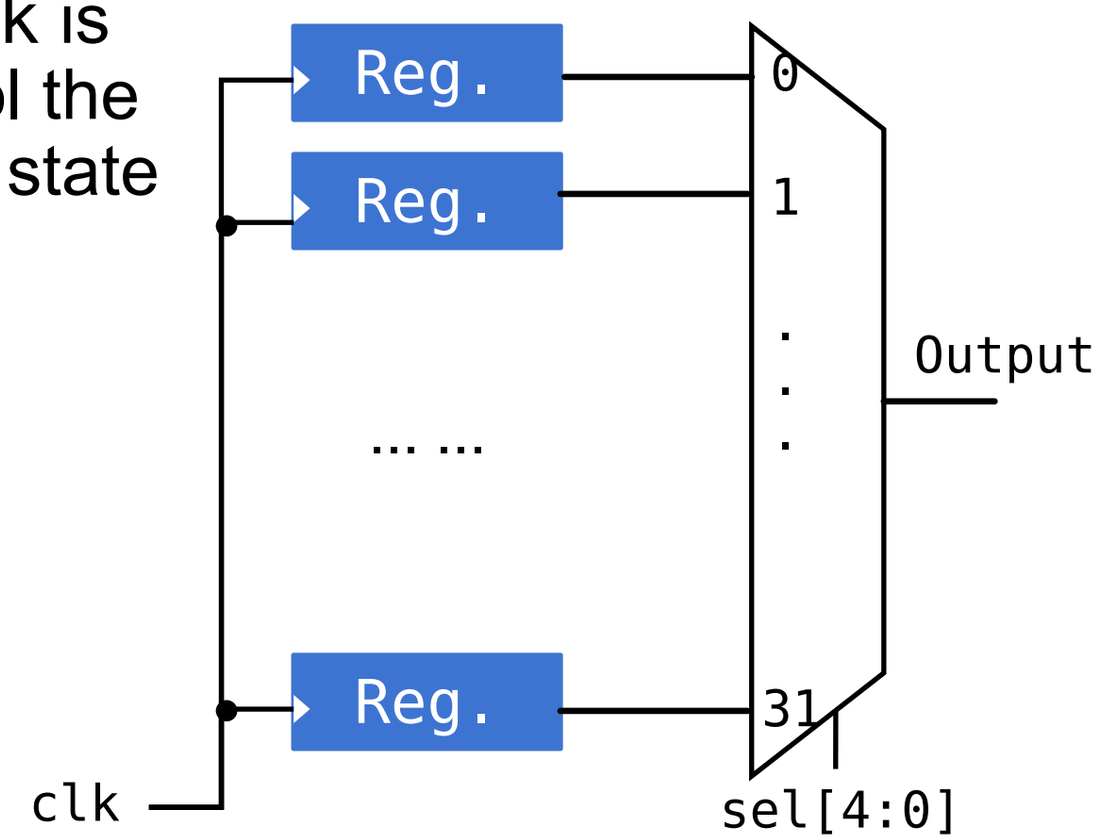
Multiplexer

Useful building blocks-Register file

- The register file is the component that contains all the general purpose registers of the microprocessor
- A register file should provide data given the register numbers
- A register file should be able to change the stored value

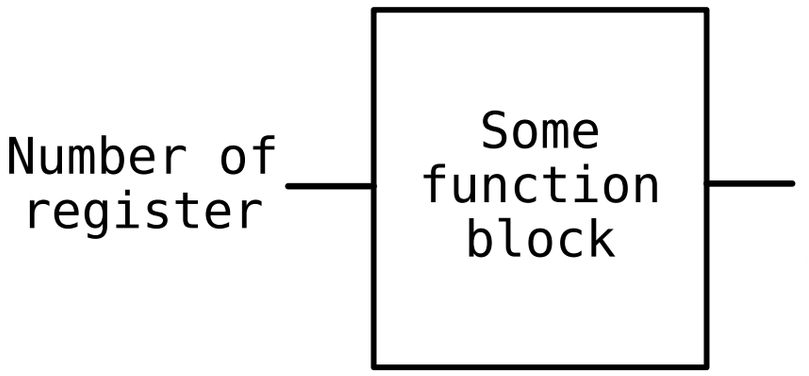


- How do we change values of a specific reg.?
- Recall that `clk` is used to control the change of the state



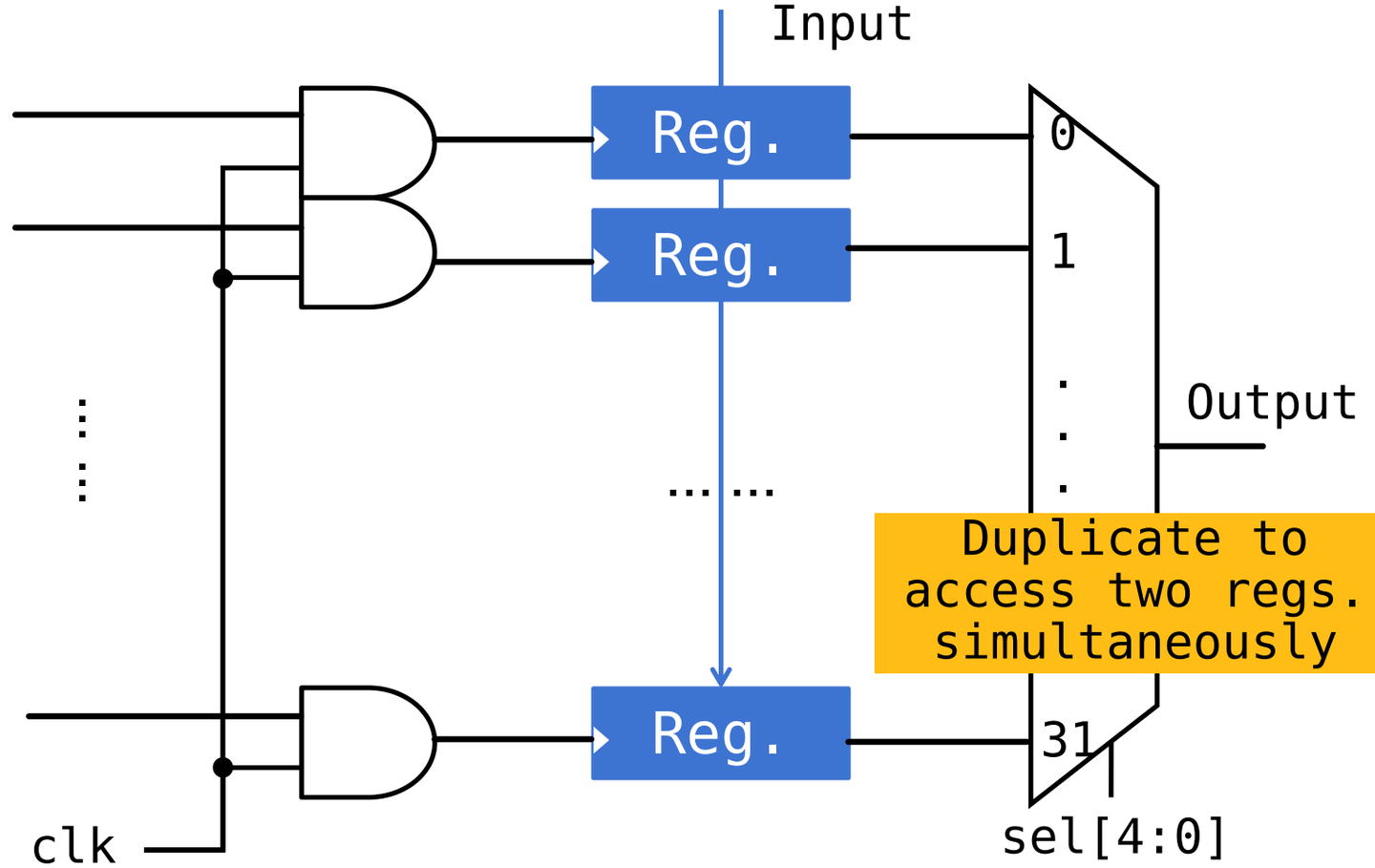
Useful building blocks-Register file

- The register file is the component that contains all the general purpose registers of the microprocessor
- A register file should provide data given the register numbers
- A register file should be able to change the stored value



(Address) decoder
or
demultiplexer

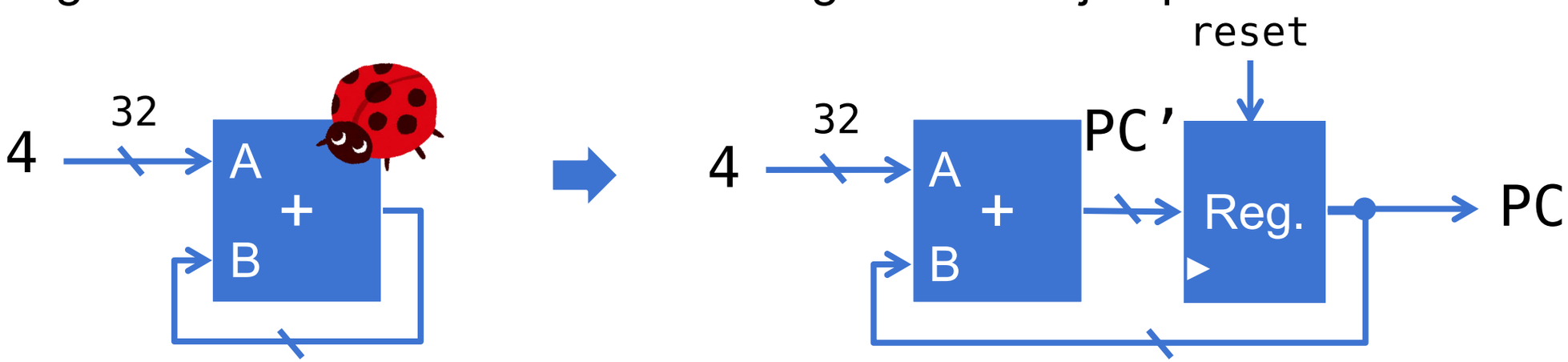
- How do we change values of a specific reg.?



- Reg. file design completed

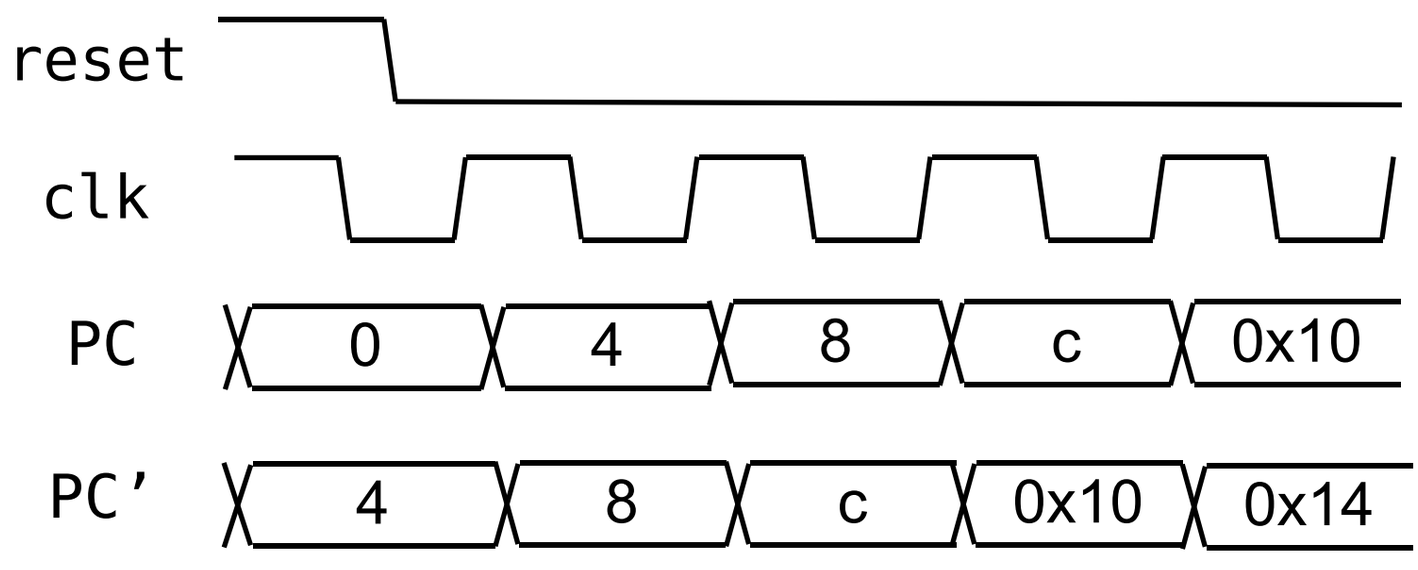
We have covered PC register previously

- Synchronous digital circuit can have feedback, e.g., iterative accumulator
  - e.g. PC = PC + 4 without considering branch or jump



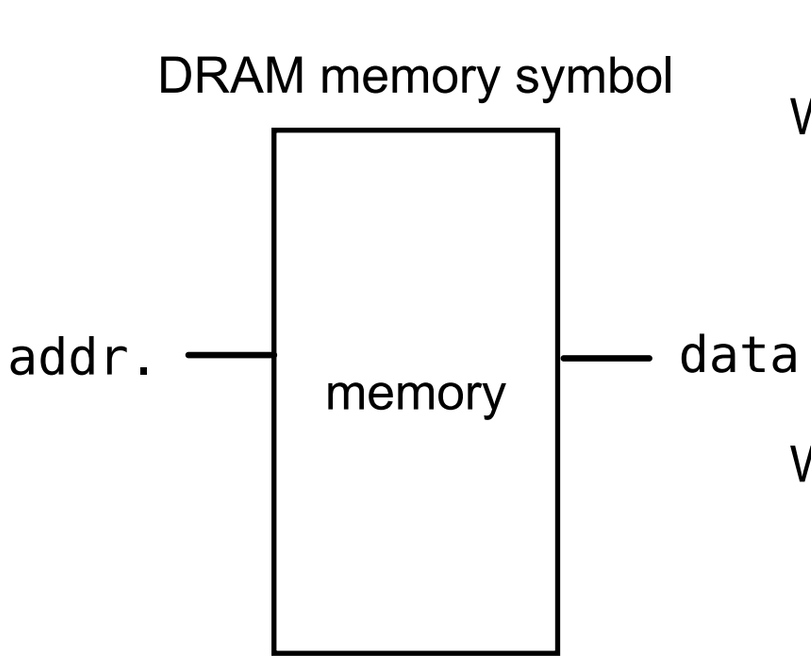
32

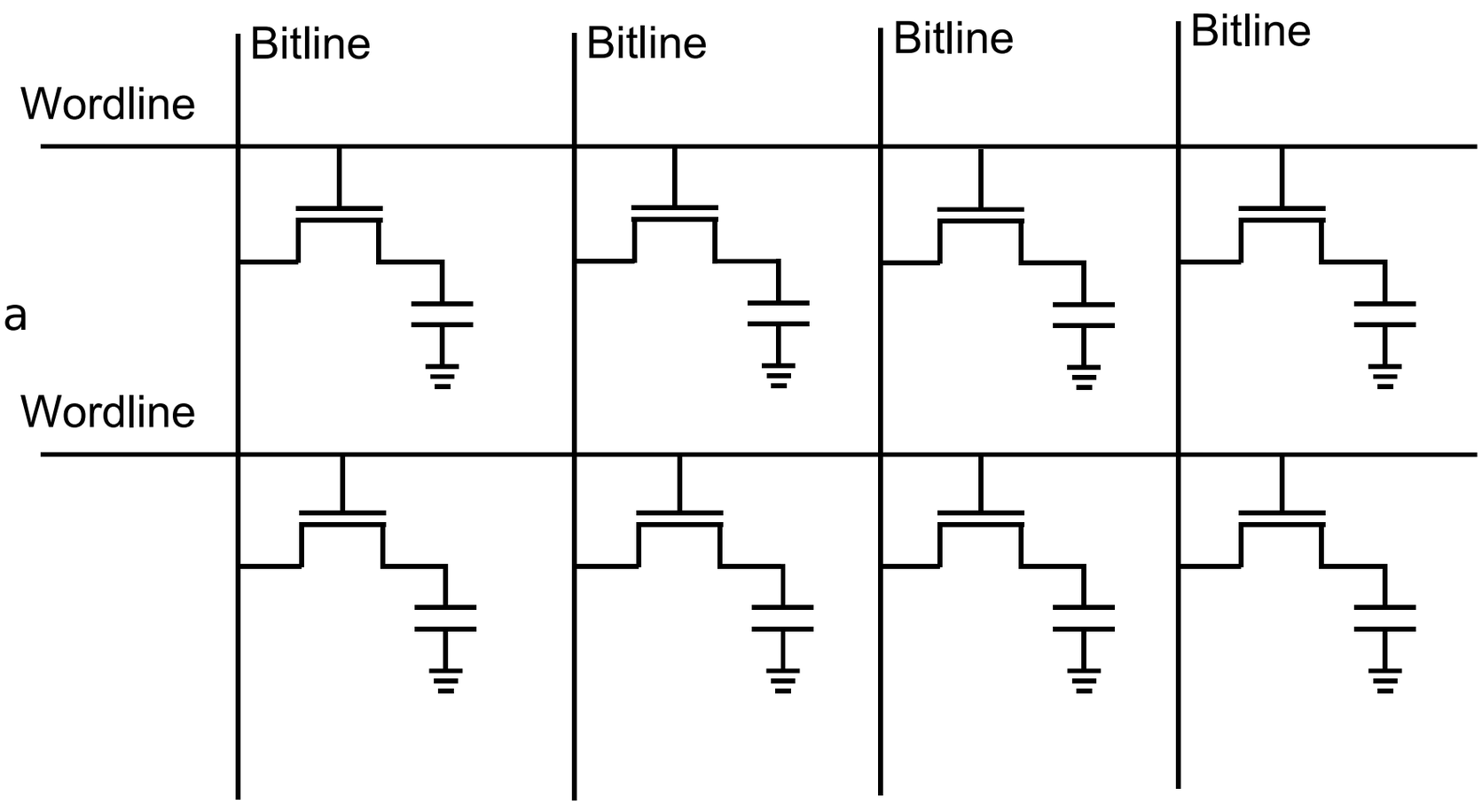
- Timing diagram



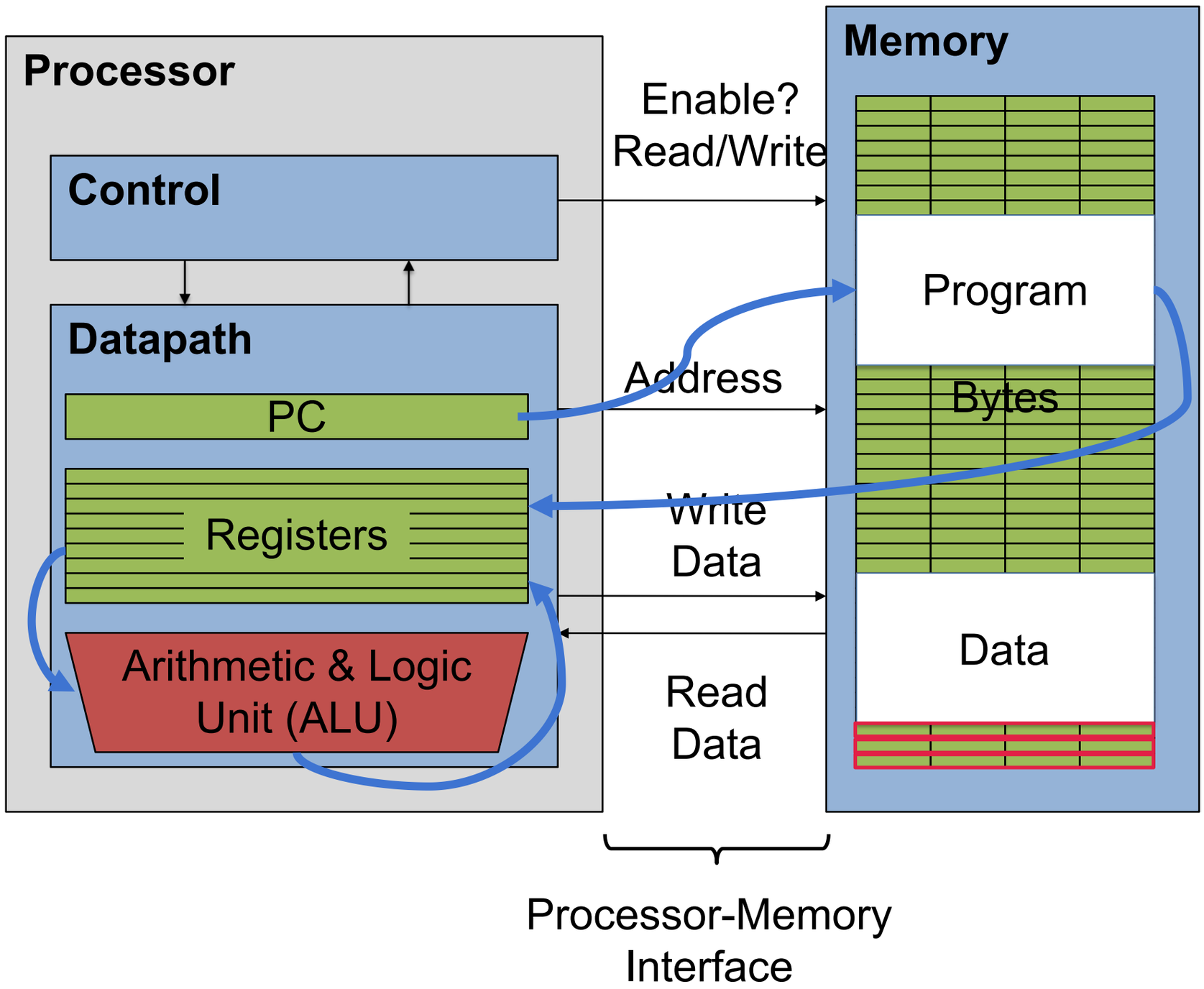
Useful building blocks-Memory

- Memory similar to register file except that the basic cell design is different
- Requires refresh for DRAM

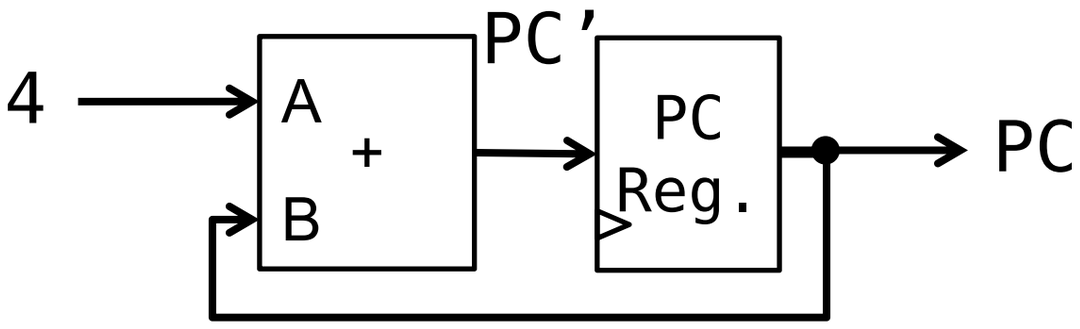




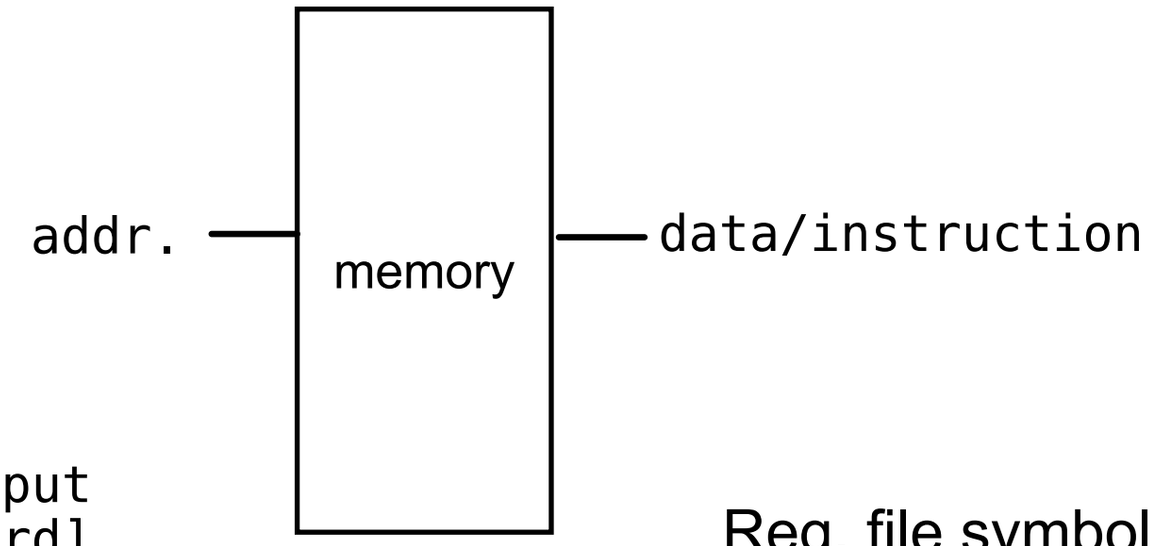
Datapath

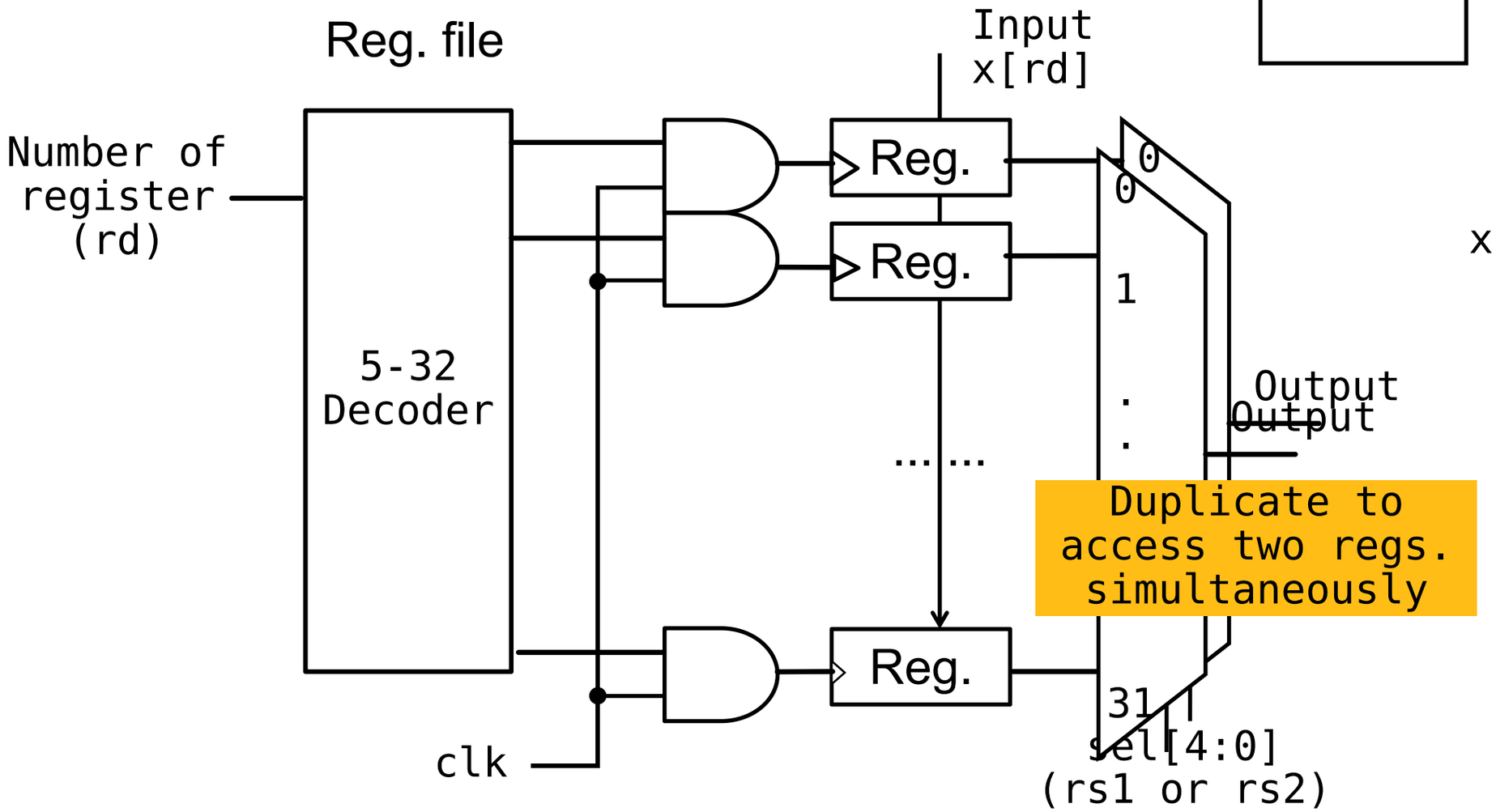

Processor
Control
Datapath
PC
Registers
Arithmetic & Logic Unit (ALU)
Enable?
Read/Write
Address
Write Data
Read Data
Memory
Program
Bytes
Data
Processor-Memory Interface

Datapath



DRAM memory symbol





Reg. file symbol