Unit 15

Processor Organization and Common Processor Components

Overview

- We will explore the common hardware components that are used to build larger devices such as a processor

You Can Do That…

Cloud & Distributed Computing
(CyberPhysical, Databases, Data Mining,etc.)

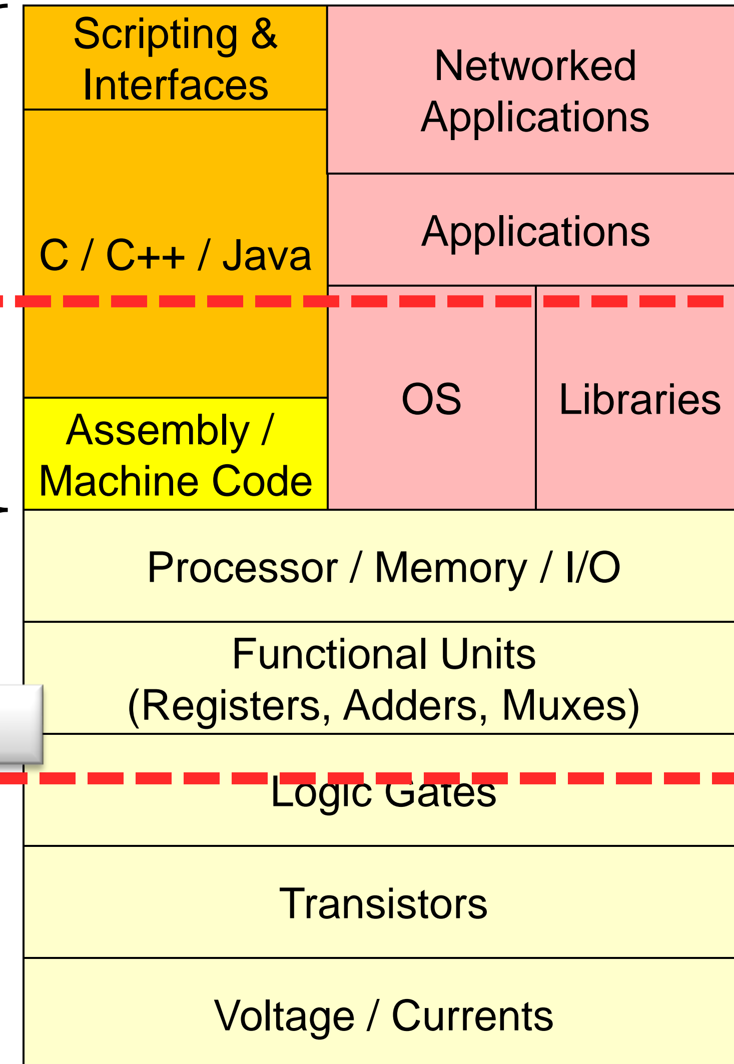
Applications
(AI, Robotics, Graphics, Mobile)

Systems & Networking
(Embedded Systems, Networks)

Architecture
(Processor & Embedded HW)

Where we will head now…

Devices & Integrated Circuits
(Semiconductors & Fabrication)



Motivation

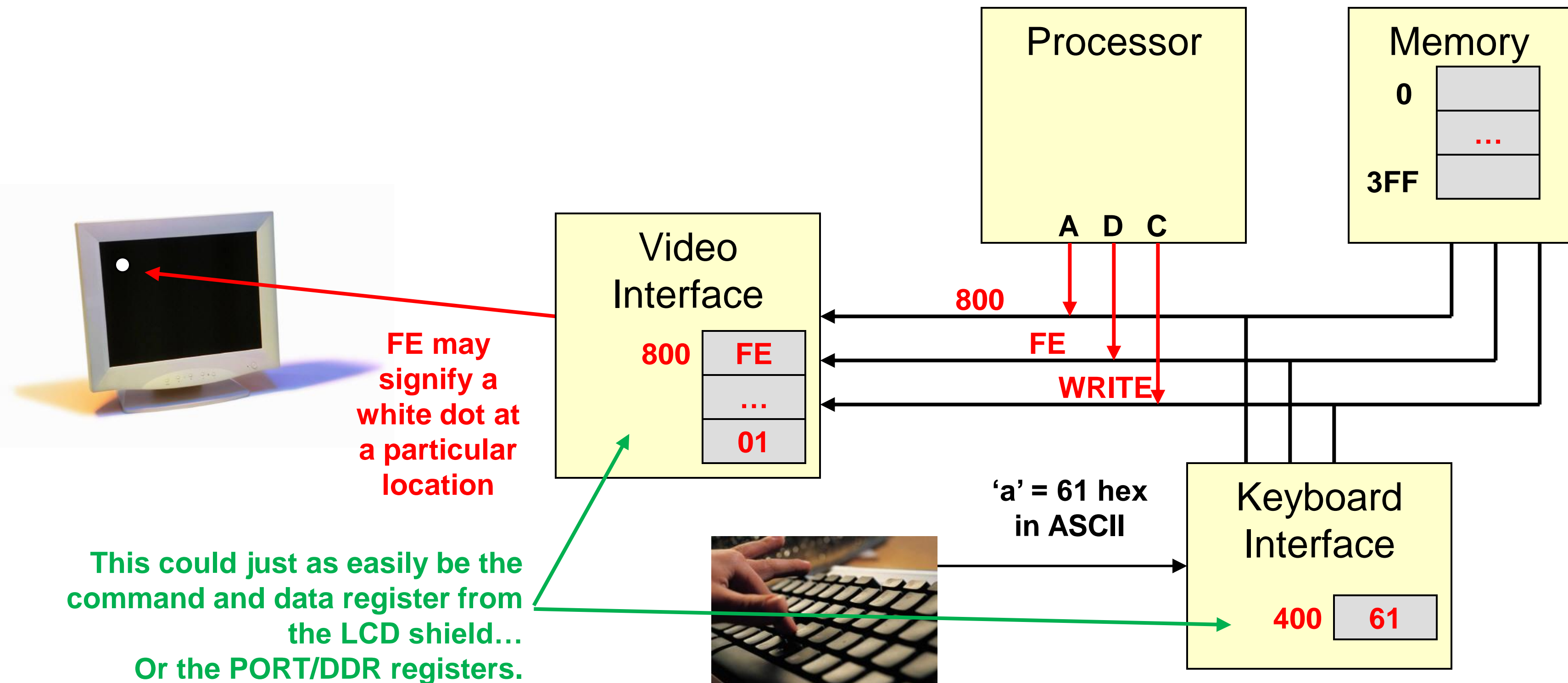
- Now that you have some understanding…
  - Of how hardware is designed and works
  - Of how software can be used to control hardware
- We will look at how to:
  - Build hardware that can execute software (i.e. design a processor)
  - Examine ideas to improve performance
  - Understand issues that drive the current industry and market

Computer Organization

- Three primary sets of components
  - Processor
  - Memory
  - I/O (everything else)
- Tell us where things live?
  - Running code
  - Compiled program (not running)
  - Circuitry to execute code
  - Source code file
  - Data variables
  - Data for the pixels being displayed on your screen

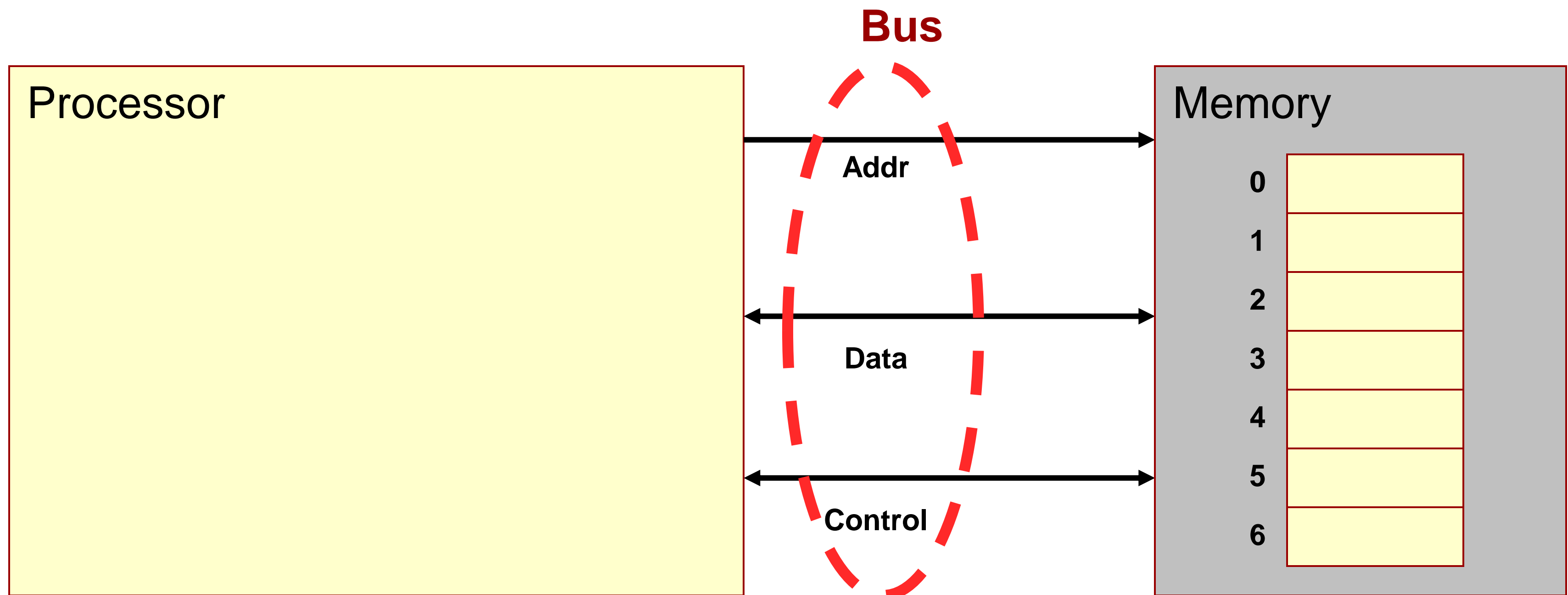
Input / Output

- Processor performs reads and writes to communicate with I/O devices just as it does with memory
  - I/O devices have locations (i.e. **registers**) that contain data that the processor can access
  - These registers are assigned unique addresses just like memory



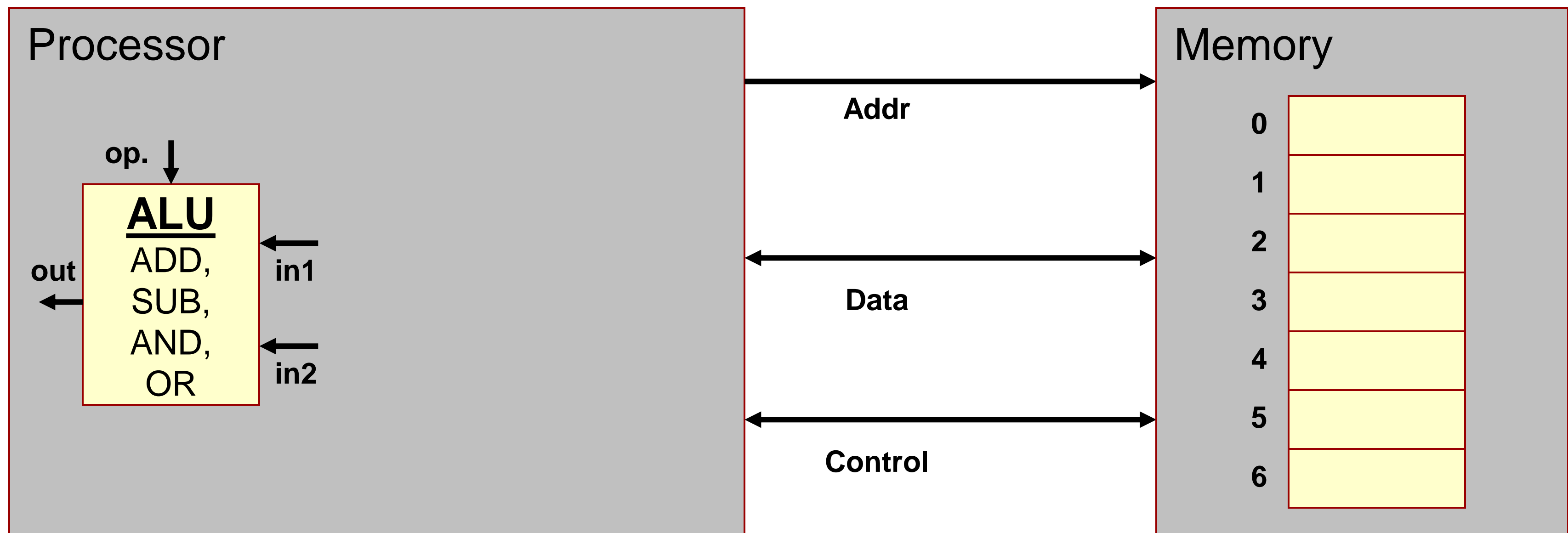
Processor

- 3 Primary Components inside a processor
  - ALU
  - Registers
  - Control Circuitry
- Connects to memory and I/O via **address**, **data**, and **control** buses (**bus** = group of wires)



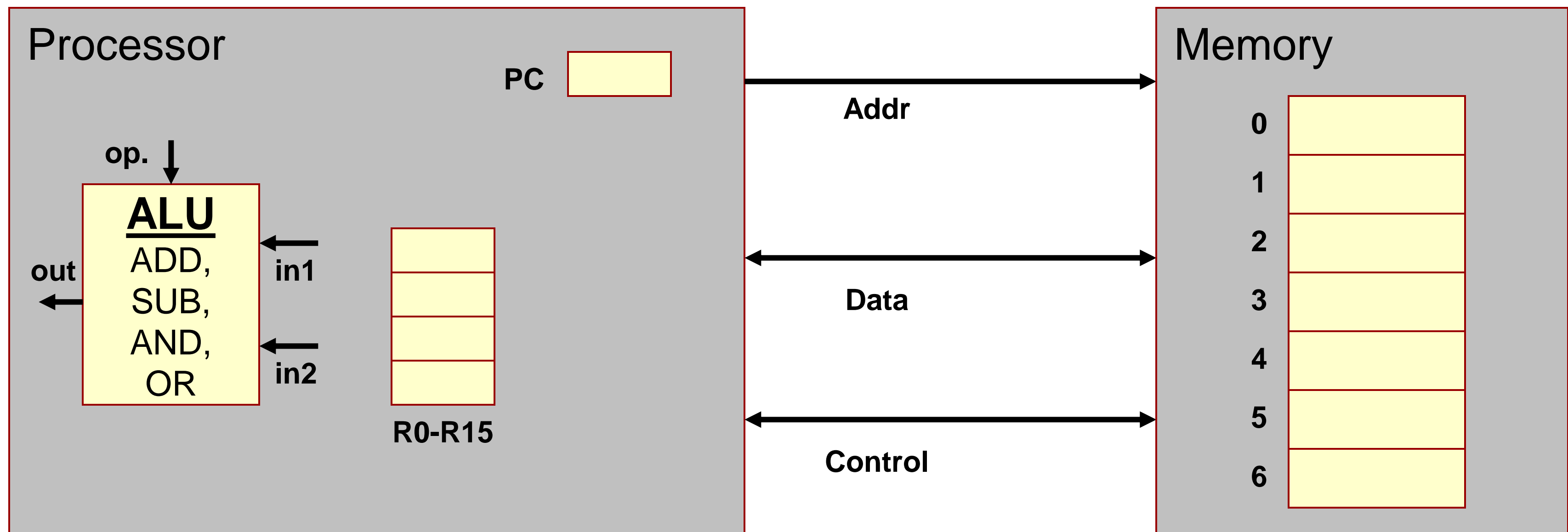
Arithmetic and Logic Unit (ALU)

- Executes arithmetic operations like addition and subtraction along with logical operations (AND, OR, etc.)



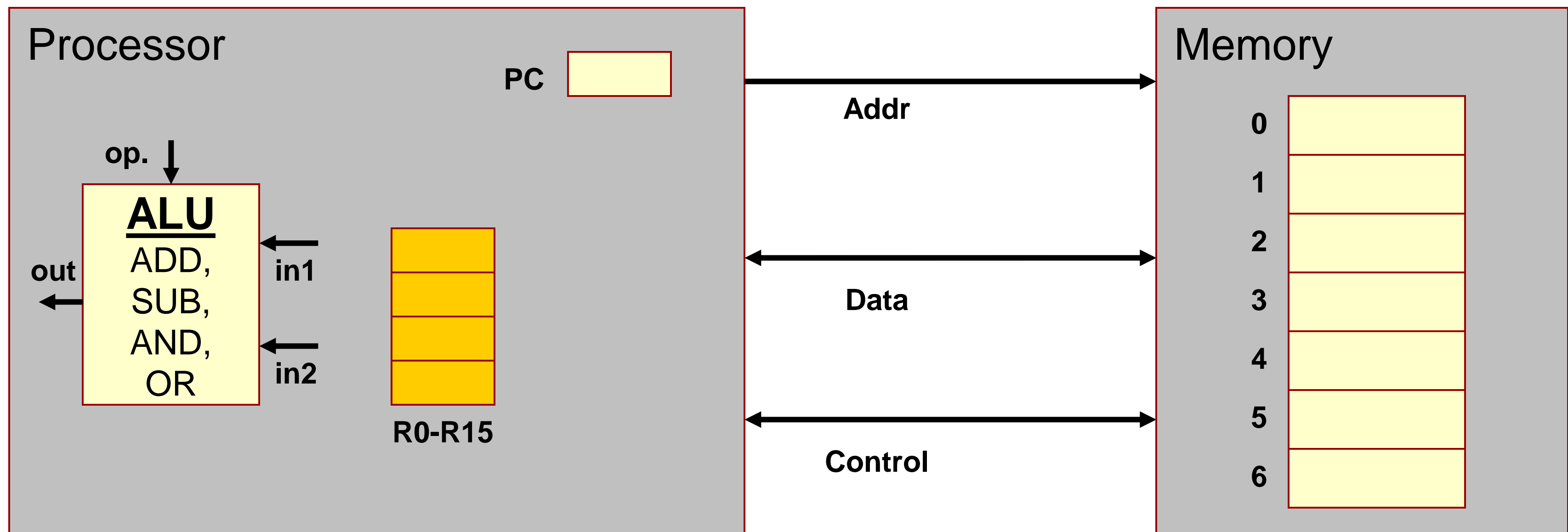
Registers

- Some are for general use by software
  - Registers provide fast, temporary storage locations within the processor (to avoid having to read/write slow memory)
- Others are required for specific purposes to ensure proper operation of the hardware



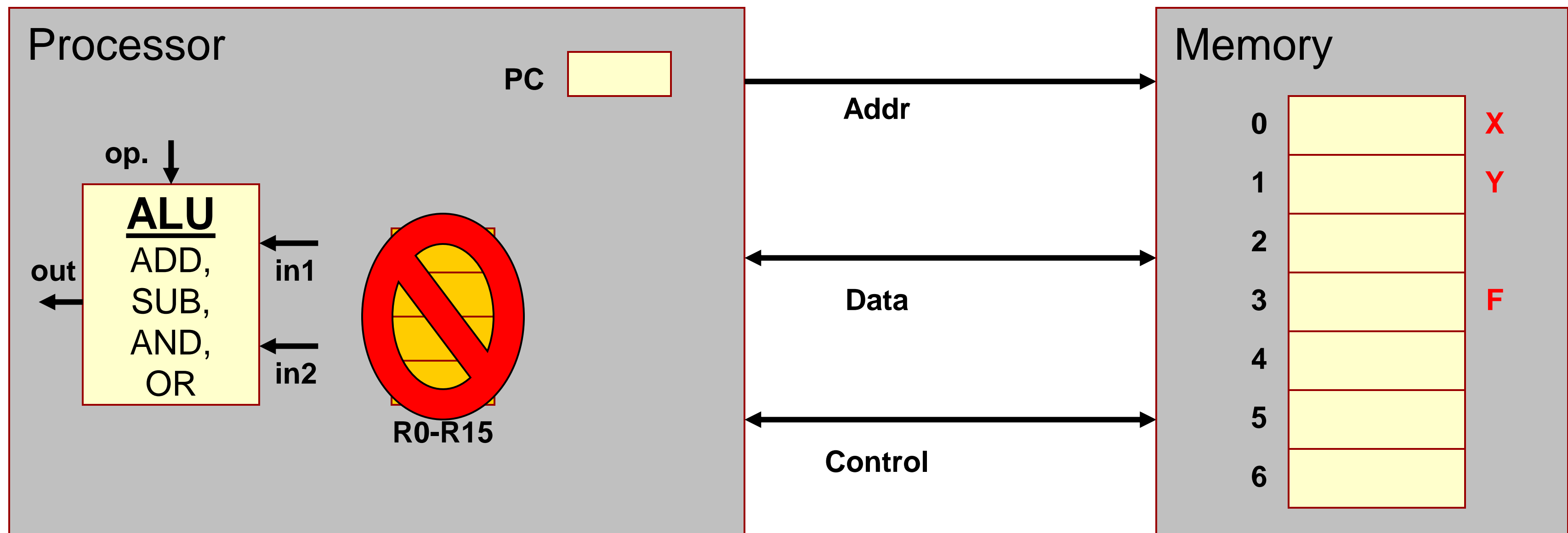
General Purpose Registers

- Registers available to software instructions for use by the programmer/compiler
- Instructions use these registers as inputs (source locations) and outputs (destination locations)



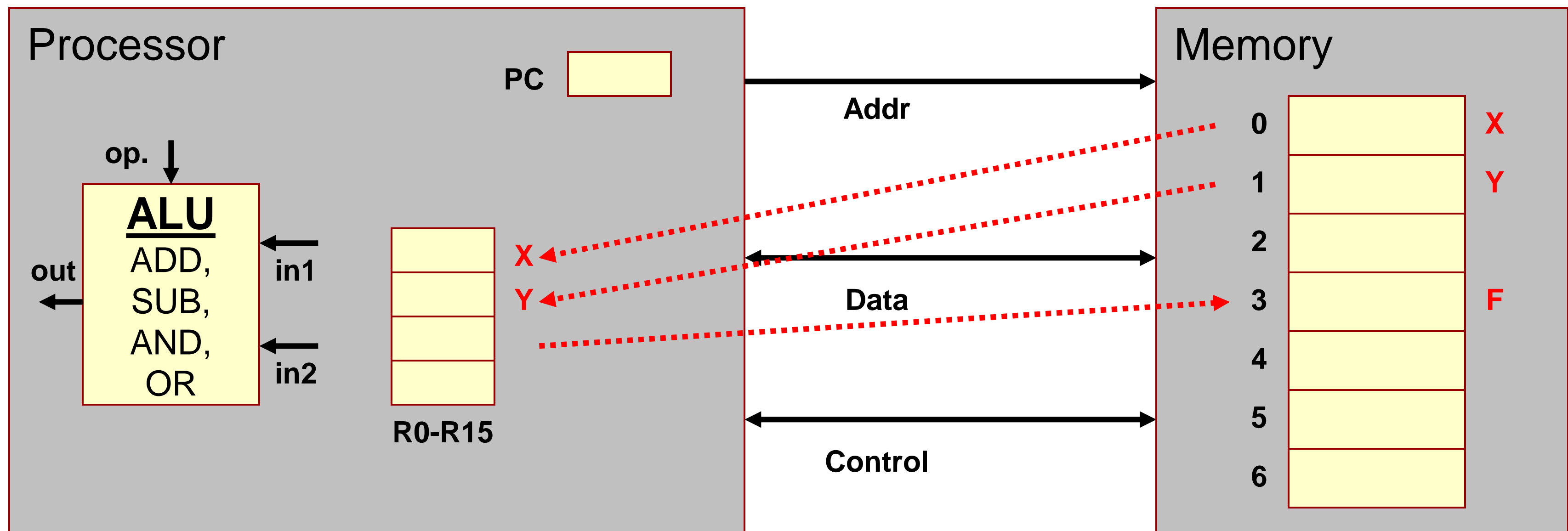
What if we didn't have registers?

- Example w/o registers: F = (X+Y) – (X*Y)
  - Requires an ADD instruction, MULtiply instruction, and SUBtract Instruction
  - w/o registers
    - ADD: Load X and Y from memory, store result to memory
    - MUL: Load X and Y again from mem., store result to memory
    - SUB: Load results from ADD and MUL and store result to memory
    - 9 memory accesses



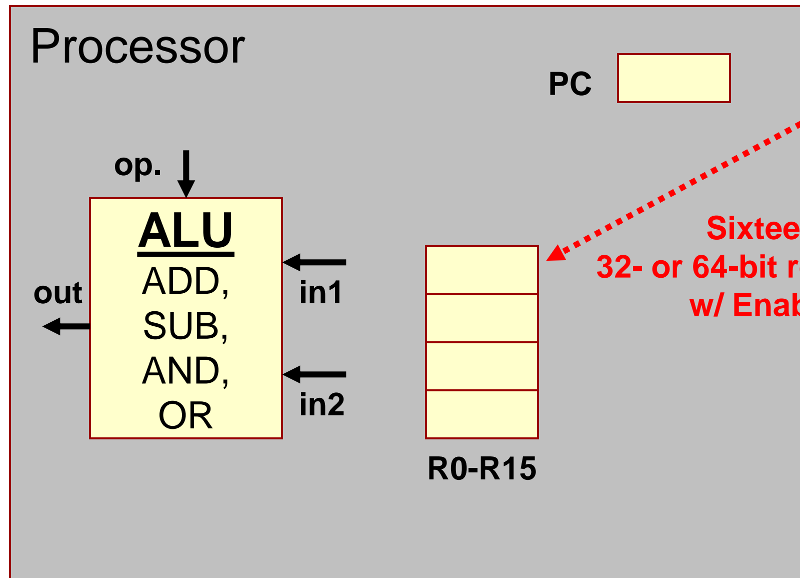
What if we have registers?

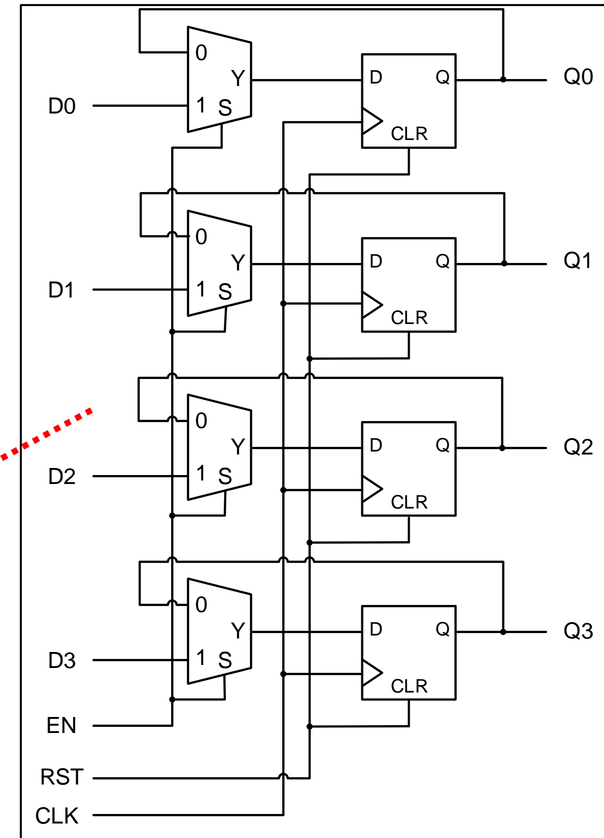
- Example w/ registers: F = (X+Y) – (X*Y)
  - Load X and Y into registers
  - ADD: R0 + R1 and store result in R2
  - MUL: R0 * R1 and store result in R3
  - SUB: R2 – R3 and store result in R4
  - Store R4 back to memory
  - 3 total memory access

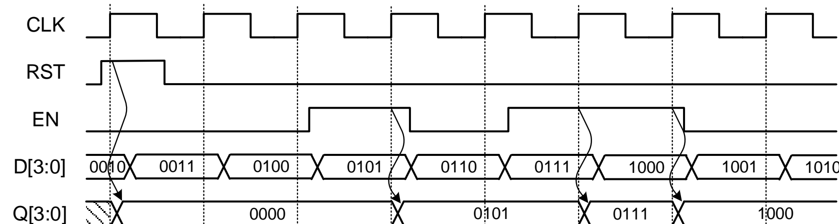


Connecting What We've Learned (1)

- What really are the registers shown below?
  - Registers w/ enables (flip-flops with muxes)

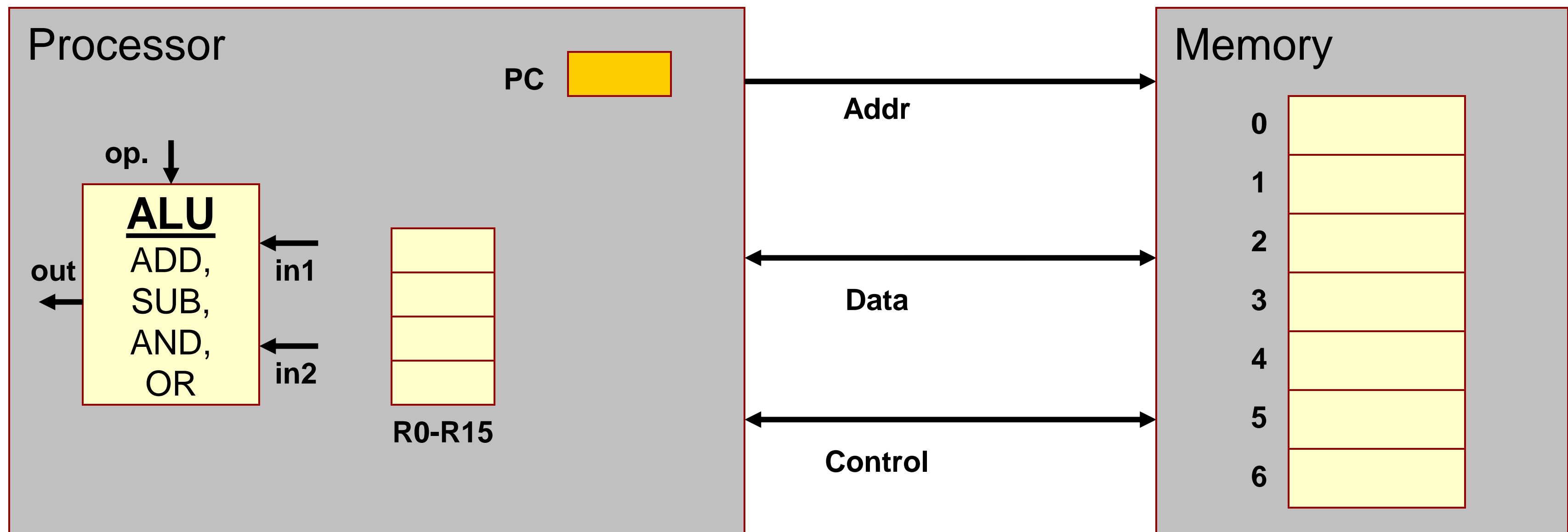






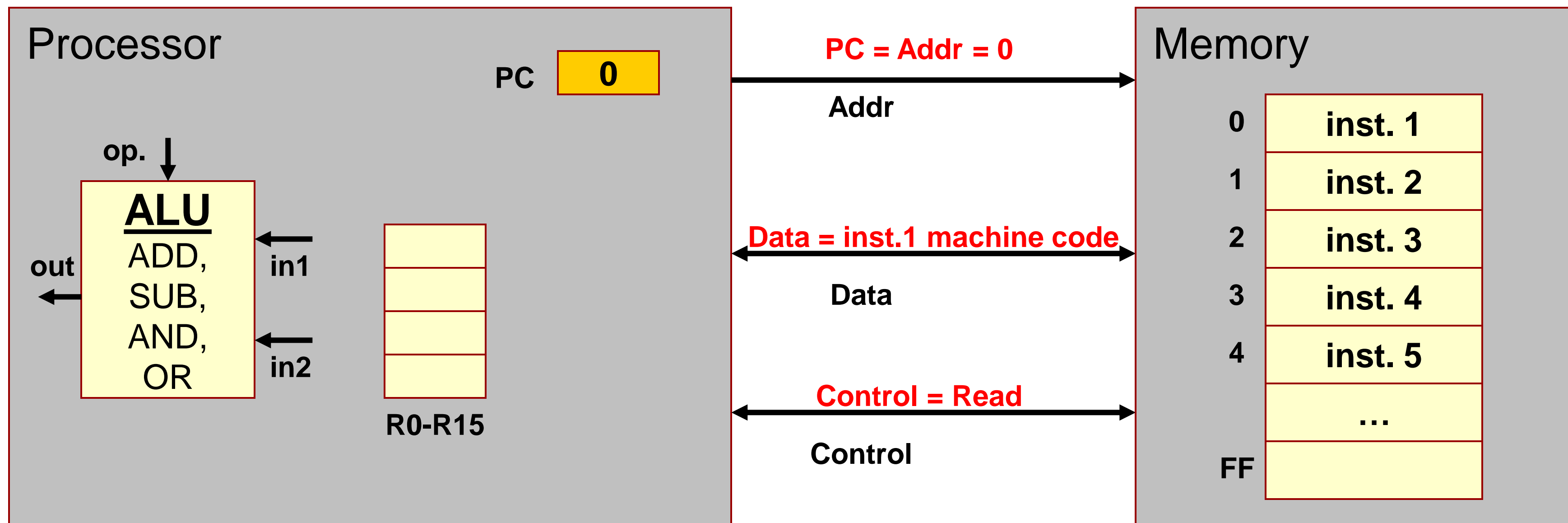
Other Registers

- Some bookkeeping information is needed to make the processor operate correctly
- Example: Program Counter (PC)
  - Recall that the processor must fetch instructions from memory before decoding and executing them
  - PC register holds the address of the next instruction to fetch



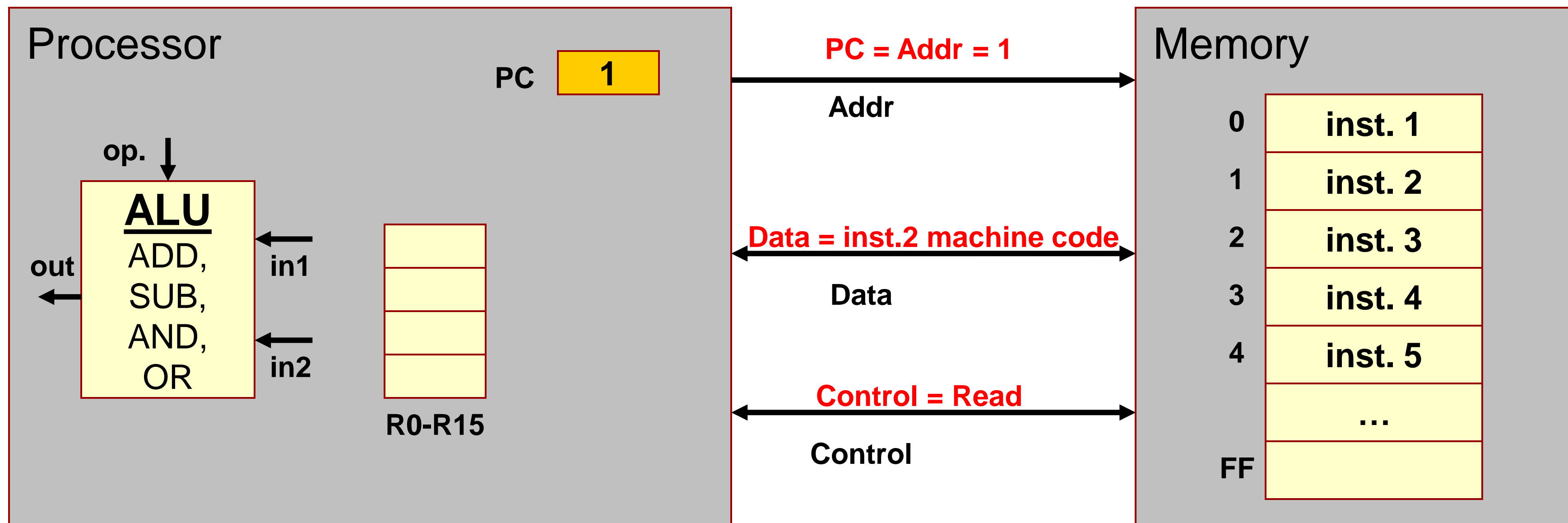
Fetching an Instruction

- To fetch an instruction
  - PC contains the address of the instruction
  - The value in the PC is placed on the address bus and the memory is told to read
  - The PC is incremented, and the process is repeated for the next instruction



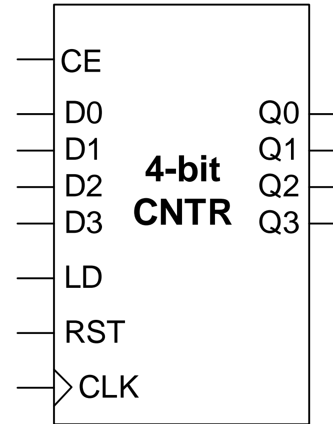
Fetching an Instruction

- To fetch an instruction
  - PC contains the address of the instruction
  - The value in the PC is placed on the address bus and the memory is told to read
  - The PC is incremented, and the process is repeated for the next instruction



Connecting What We've Learned (2)

- What is the PC really?
  - A counter



CLK	RST	LD	CE	Q*
0,1	X	X	X	Q
⇈	1	X	X	0
⇈	0	1	X	D[3:0]
⇈	0	0	1	Q+1
⇈	0	0	0	Q

32- or 64-bit Counter w/ Enable

Processor

PC **1**

op.

ALU
ADD, SUB, AND, OR

out **in1** **in2**

R0-R15

PC = Addr = 1

Addr

Data = inst.2 machine code

Data

Control = Read

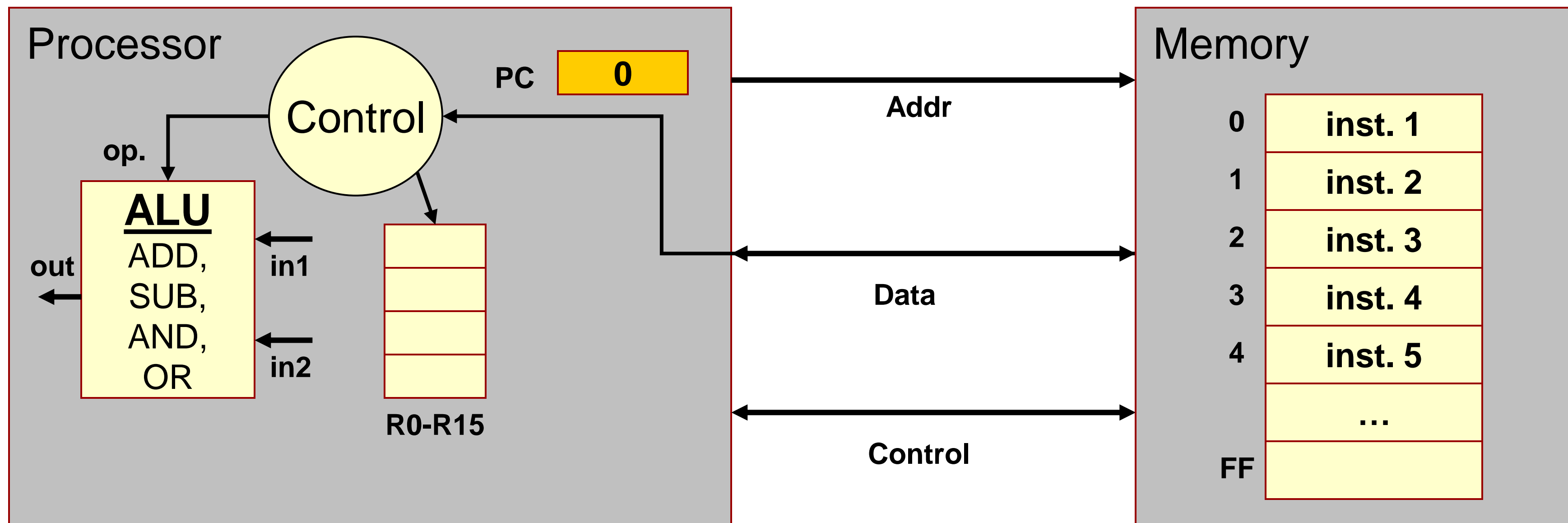
Control

Memory

0	**inst. 1**
1	**inst. 2**
2	**inst. 3**
3	**inst. 4**
4	**inst. 5**
	…
FF	

Control Circuitry

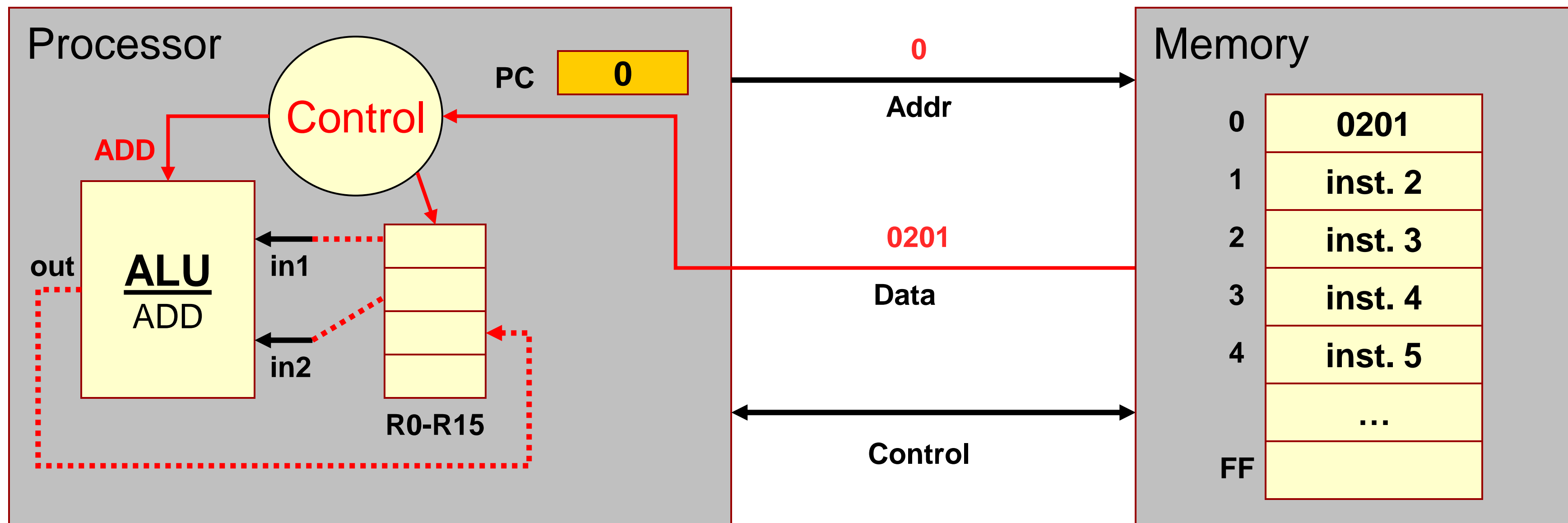
- Control circuitry is used to decode the instruction and then generate the necessary signals to complete its execution
- Controls the ALU
- Selects registers to be used as source and destination locations (using muxes)



Control Circuitry

- Assume 0x0201 is machine code for an ADD instruction of R2 = R0 + R1
- Control Logic will…
  - select the registers (R0 and R1)
  - tell the ALU to add
  - select the destination register (R2)

Opcode (4-bits)	Dst. Reg. (4-bits)	Src1 Reg. (4-bits)	Src2 Reg. (4-bits)



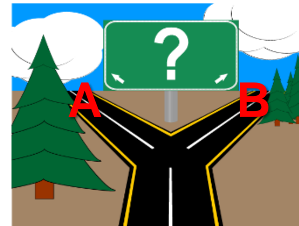
Starting our Design

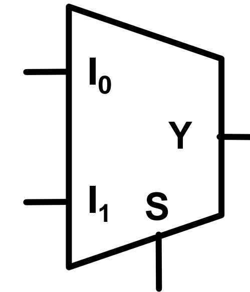
- Now let's start to design our ALU and see some other details of the CPU block diagram just presented

WIDE MUXES

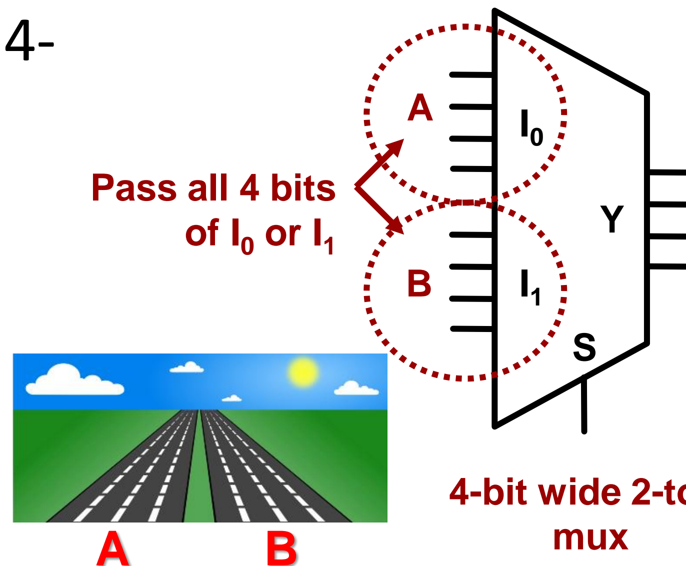
Building Wide Muxes

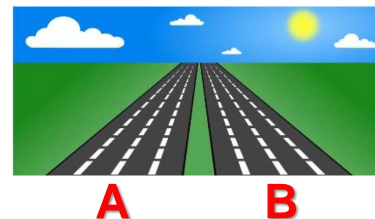
- So far muxes only have single bit inputs…
  - I_0 is only 1-bit
  - I_1 is only 1-bit
- What if we still want to select between 2 inputs but now each input is a 4-bit number
- Use a 4-bit wide 2-to-1 mux





1-bit wide 2-to-1 mux



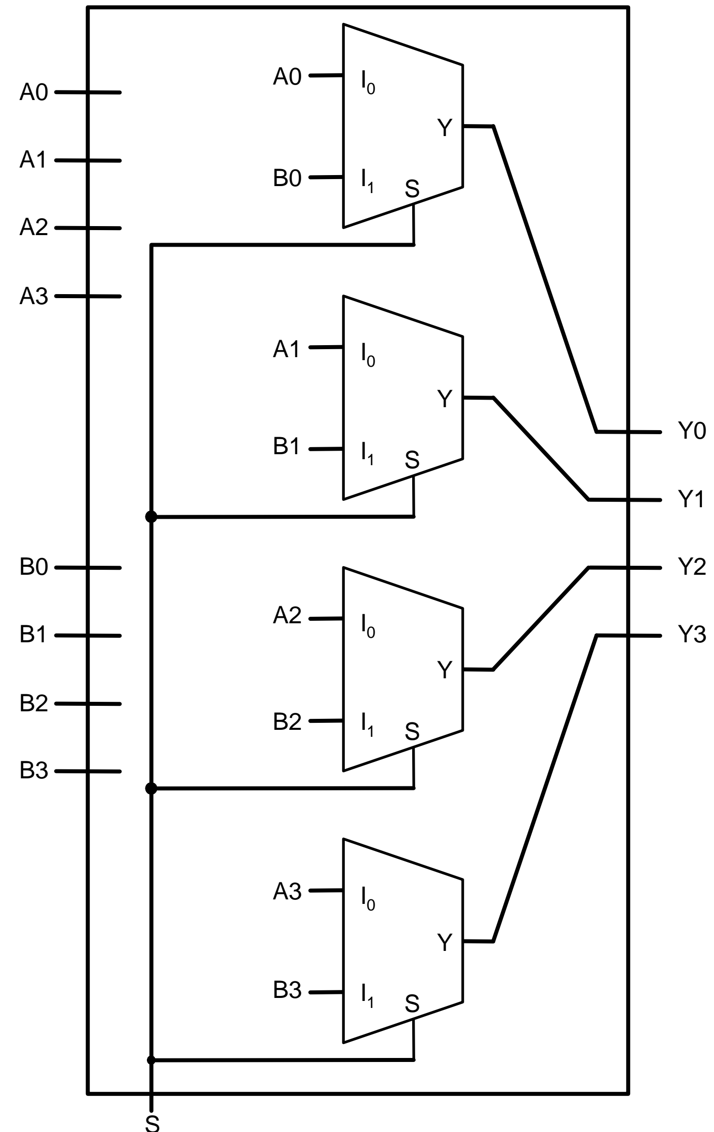


4-bit wide 2-to-1 mux

When we select I_0 or I_1 we want all 4-bits of that input to be passed

Building Wide Muxes

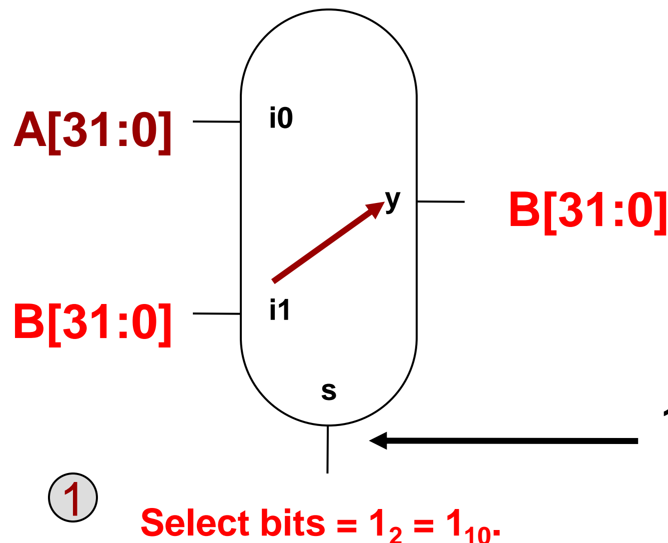
- Use one mux per "lane" (bit)
  - To build a 4-bit wide 2-to-1 mux, use 4 separate 2-to-1 muxes
- Operation:
  - When S=0, all muxes pass their I_0 inputs which means all the A bits get through
  - When S=1, all muxes pass their I_1 inputs which means all the B bits get through
- In general, to build an **m-bit wide (i.e. m-lane) n-to-1 mux**, use **m individual n-to-1 muxes**



Wide Multiplexer Example 1

- This 2-to-1, 32-bit wide mux is really:
  - 32 individual 2-to-1 muxes, each handling 1 "lane" of the 32-bit highway merger

2-to-1 Mux,
32-bit wide mux

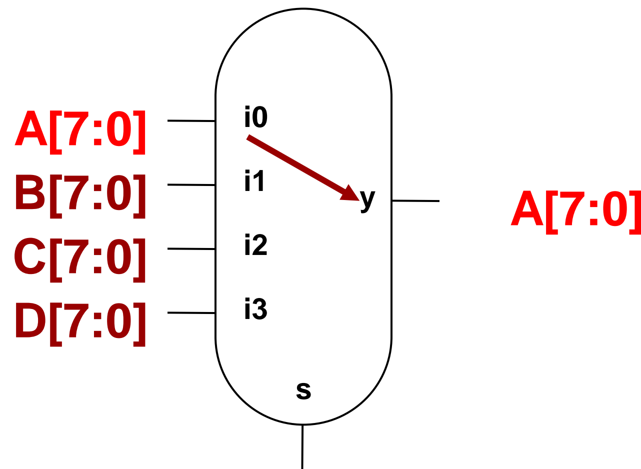


② Thus, input 1 = B[31:0] is selected and passed to the output

1 select line is needed since we are still selecting only one of the 2 inputs

Wide Multiplexer Example 2

- This 4-to-1, 8-bit wide mux is really:
  - 8 individual 4-to-1 muxes, each handling 1 "lane" of the 8-bit highway merger



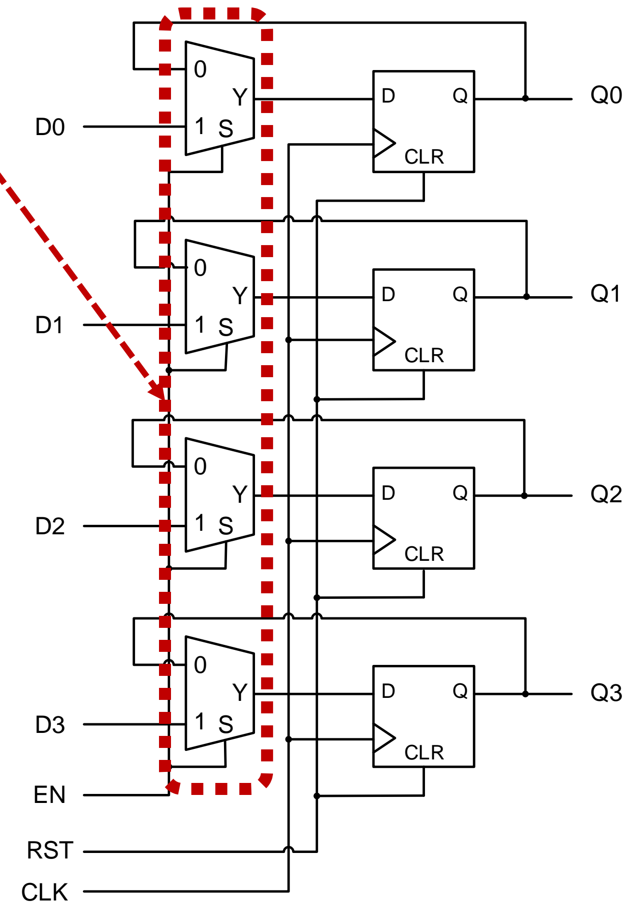
Exercise

- How many 1-bit wide muxes and of what size would you need to build a **4-to-1, 16-bit** wide mux (i.e. there are 4 numbers: W[15:0], X[15:0], Y[15:0] and Z[15:0] and you must select one)
- How many 1-bit wide muxes and of what size would you need to build a **8-to-1, 2-bit** wide mux?

Wide Muxes in Registers w/ Enables

- Registers (D-FF’s) will sam[...] bit every clock edge and pass it to Q
- Sometimes we may want to hold the value of Q and ignore D even at a clock edge
- We can add an enable input and some logic in front of the D-FF to accomplish this

2-to-1, 4-bit wide mux

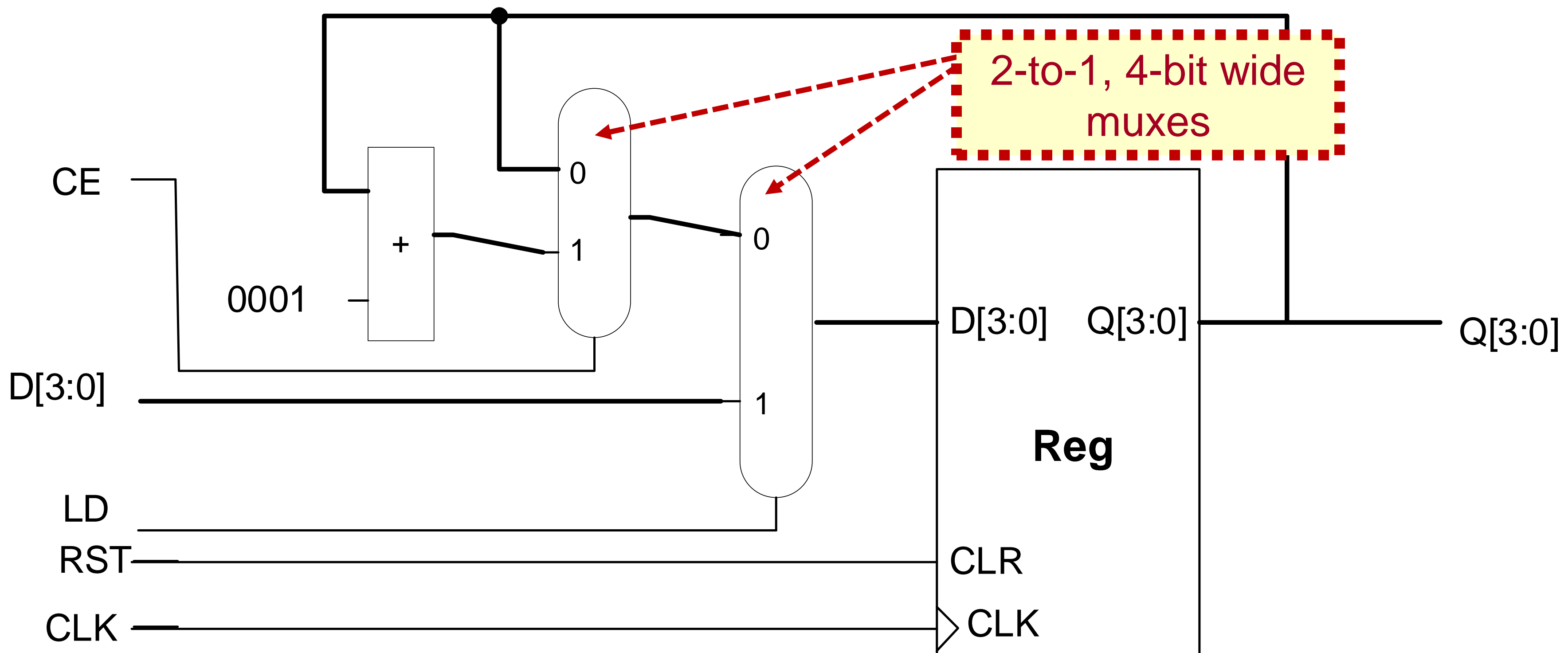


4-bit register with 4-bit wide 2-to-1 mux in front of the D inputs

CLK	RST	EN	D_i	Q_i^*
0,1	X	X	X	Q_i
↑	1	X	X	0
↑	0	0	X	Q_i
↑	0	1	0	0
↑	0	1	1	1

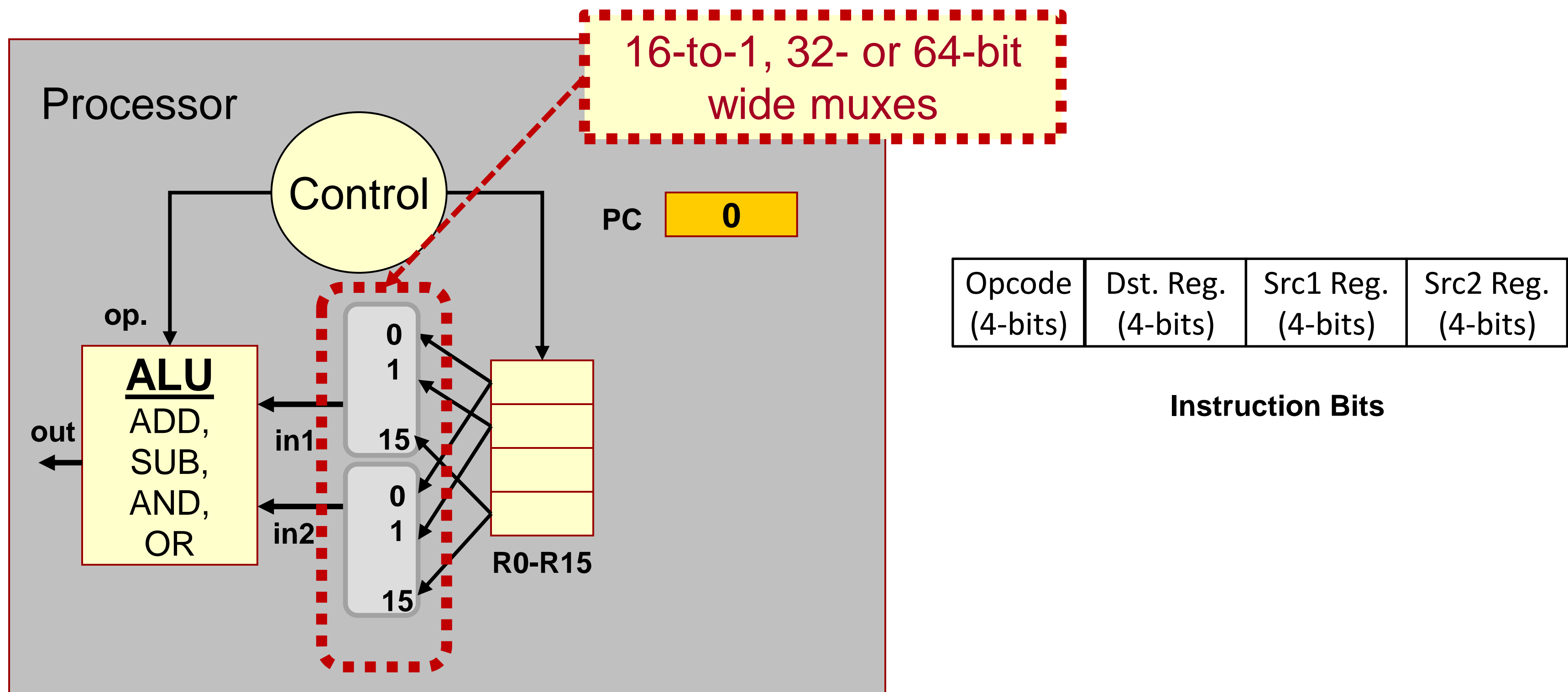
Wide Muxes in Counters

- Sketch the design of the 4-bit counter presented on the previous slides



Wide Muxes for Processor Register Selection

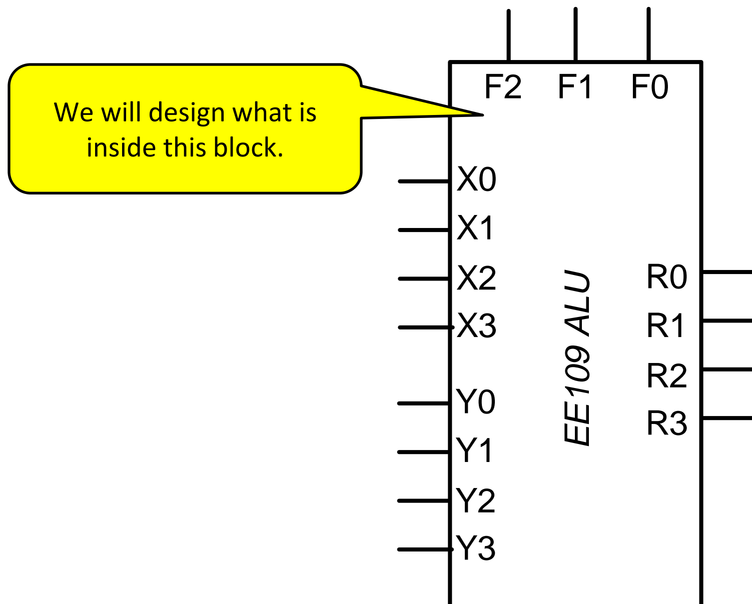
- An instruction specifies which registers to perform an operation
  - We the use a mux to select those register values to connect to the ALU



ALU DESIGN

Arithmetic and Logic Units

- Arithmetic and Logic Units (ALUs) can perform 1 of many potential arithmetic or logic operations
- Let's define and design an ALU that will perform various operations…



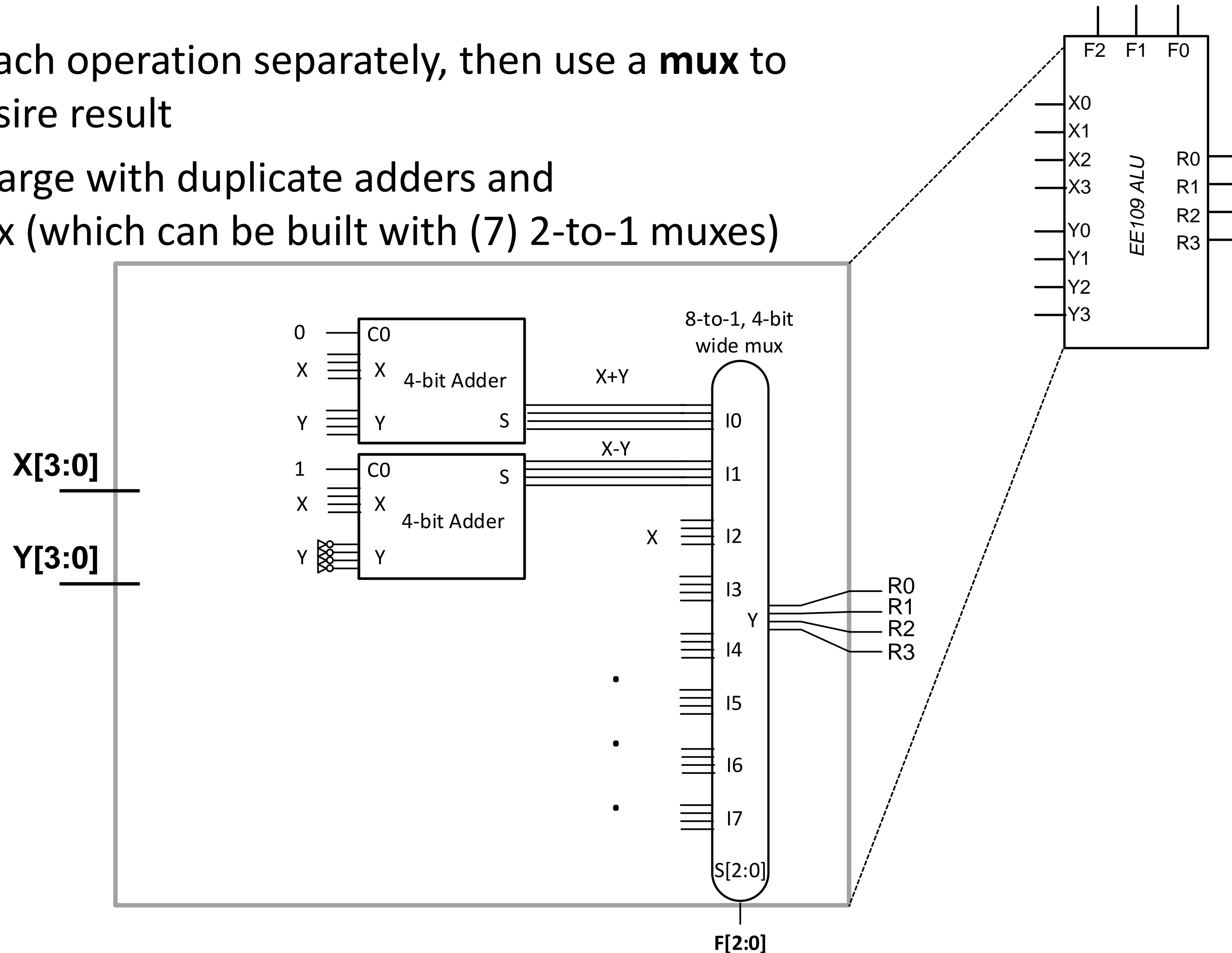
We just made up these code assignments and the various operations. Remember, we definitely need to support ADD, SUB, AND, and CLR (R=0).

F[2:0]	Op./Result
000	R = X + Y
001	R = X - Y
010	R = X
011	R = Y - X
100	R = X & Y
101	Unused
110	R = 0
111	Unused

Brute Force

- Implement each operation separately, then use a **mux** to select the desire result
- This is quite large with duplicate adders and an 8-to-1 mux (which can be built with (7) 2-to-1 muxes)

F[2:0]	Op./Result
000	R = X + Y
001	R = X - Y
010	R = X
011	R = Y - X
100	R = X & Y
101	Unused
110	R = 0
111	Unused

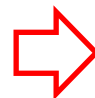


Optimizing Our Design

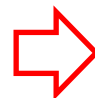
- Identify where logic can be shared
  - First 4 operations can be implemented with an adder if we select the inputs carefully
  - Break the operations into 2 groups with a mux choosing the final output form the appropriate group

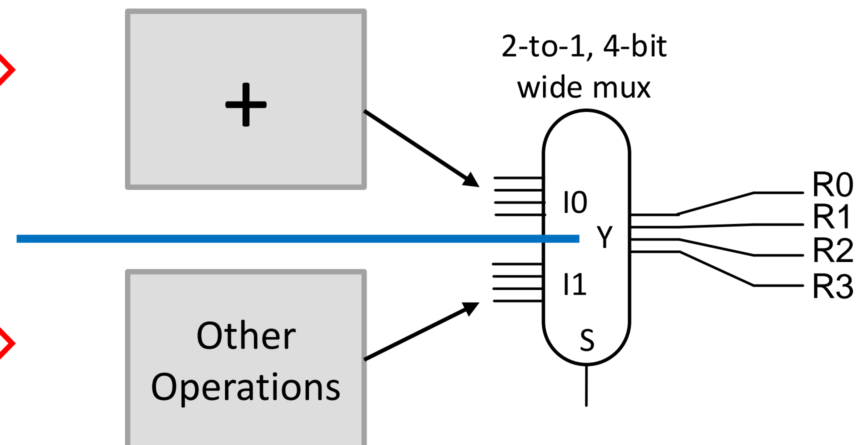
F[2:0]	Op./Result
000	R = X + Y
001	R = X - Y
010	R = X (X + 0)
011	R = Y - X
100	R = X & Y
101	Unused
110	R = 0
111	Unused

Can all be implemented with one adder.



Difficult or impossible to be implemented with just an adder

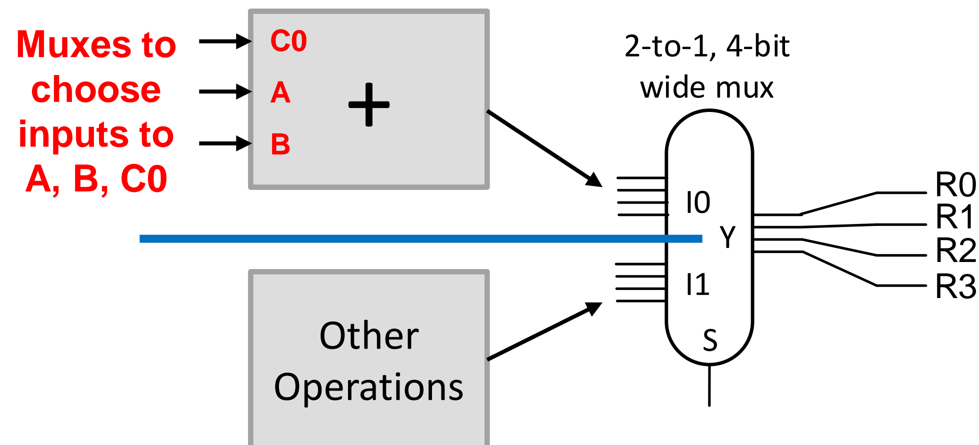




Optimizing Our Design

- With the design partitioned into 2 subgroups, we can work on each subgroup separately (divide and conquer!)
- For each of the arithmetic operations, determine what inputs we would want to connect to the adder and then we'll use muxes to achieve those options

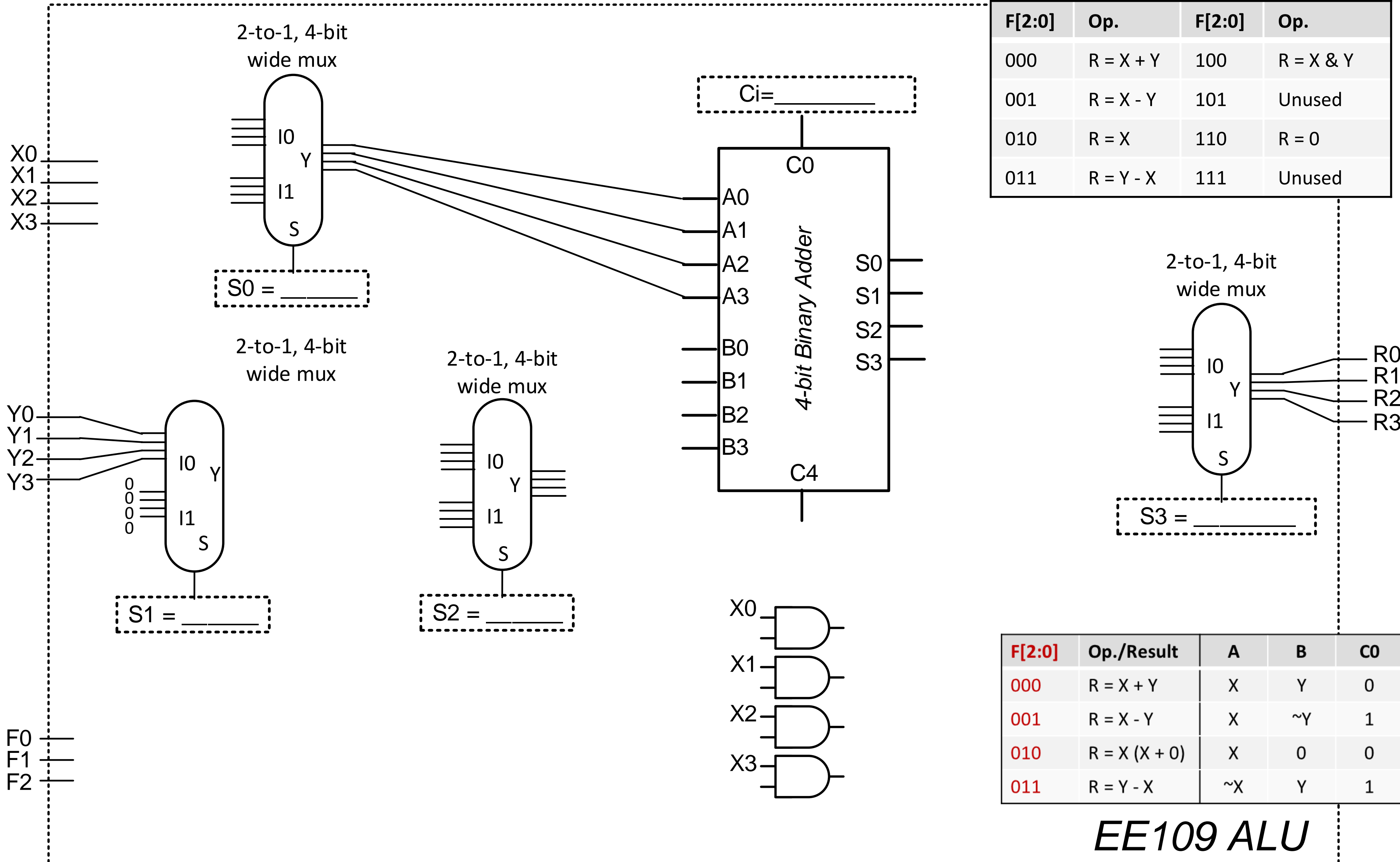
F[2:0]	Op./Result	A	B	C0
000	R = X + Y	X	Y	0
001	R = X - Y	X	~Y	1
010	R = X (X + 0)	X	0	0
011	R = Y - X	~X	Y	1



Blank ALU To Complete

- We will now complete the entire ALU but do so in **two** steps
- **Step 1: Datapath**
  - Determine muxes, adders, and other components necessary as well as their DATA (as opposed to control) connections
  - Do NOT worry how they will be controlled…imagine they can be "magically" controlled to do what we desire
- **Step 2: Control**
  - Replace the "magic" by writing a truth table for each control signal and then implementing that logic
  - Complete the truth table by taking a single input case at a time (e.g. R=X-Y) and determine the needed control signal value

Blank ALU Datapath



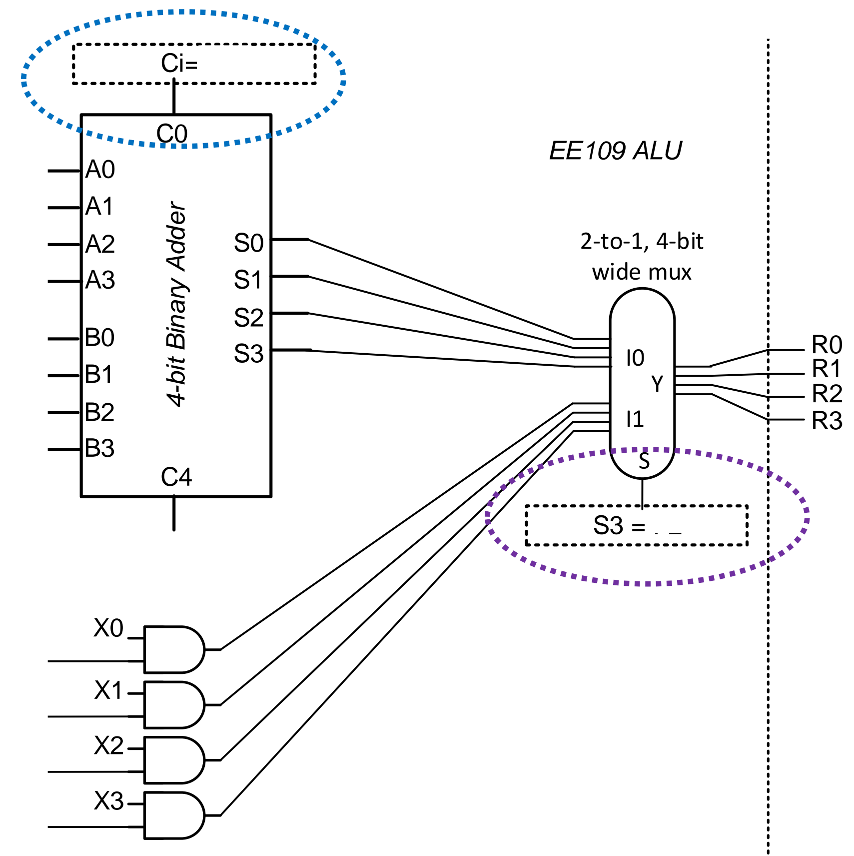
F[2:0]	Op.	F[2:0]	Op.
000	R = X + Y	100	R = X & Y
001	R = X - Y	101	Unused
010	R = X	110	R = 0
011	R = Y - X	111	Unused

F[2:0]	Op./Result	A	B	C0
000	R = X + Y	X	Y	0
001	R = X - Y	X	~Y	1
010	R = X (X + 0)	X	0	0
011	R = Y - X	~X	Y	1

ALU Control

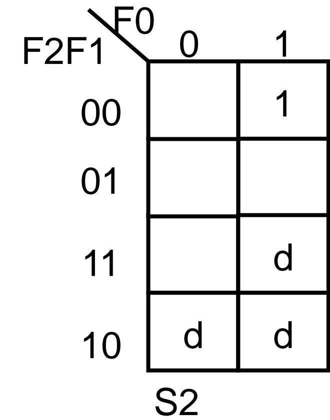
- Find the control logic by determining the desired control signal value for each operation

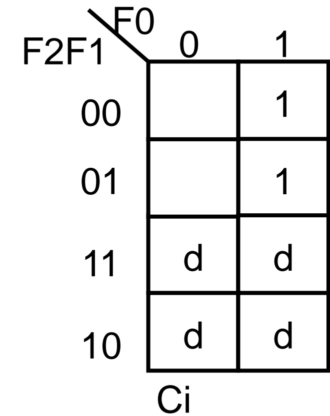
R	F[2:0]	Ci	S3
X+Y	000	0	0
X-Y	001	1	0
X	010	0	0
Y-X	011	1	0
X & Y	100	d	1
unused	101	d	d
0	110	d	1
unused	111	d	d



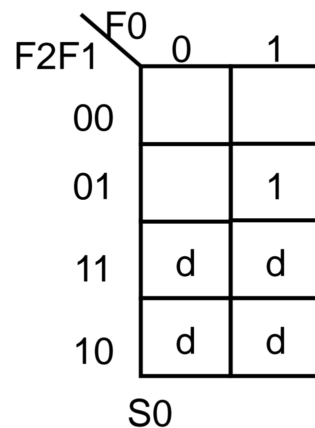
Final Control Logic

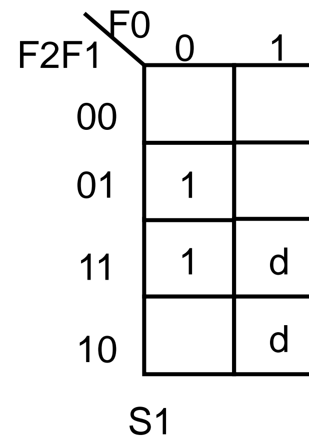
R	F[2:0]	S0	S1	S2	Ci	S3
X+Y	000	0	0	0	0	0
X-Y	001	0	0	1	1	0
X	010	0	1	0	0	0
Y-X	011	1	0	0	1	0
X & Y	100	d	0	d	d	1
unused	101	d	d	d	d	d
0	110	d	1	0	d	1
unused	111	d	d	d	d	d

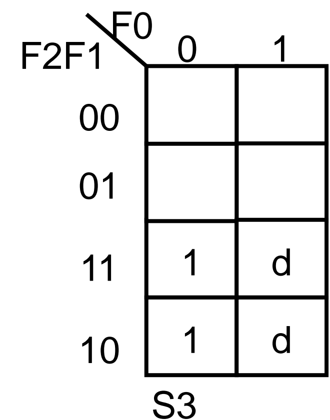




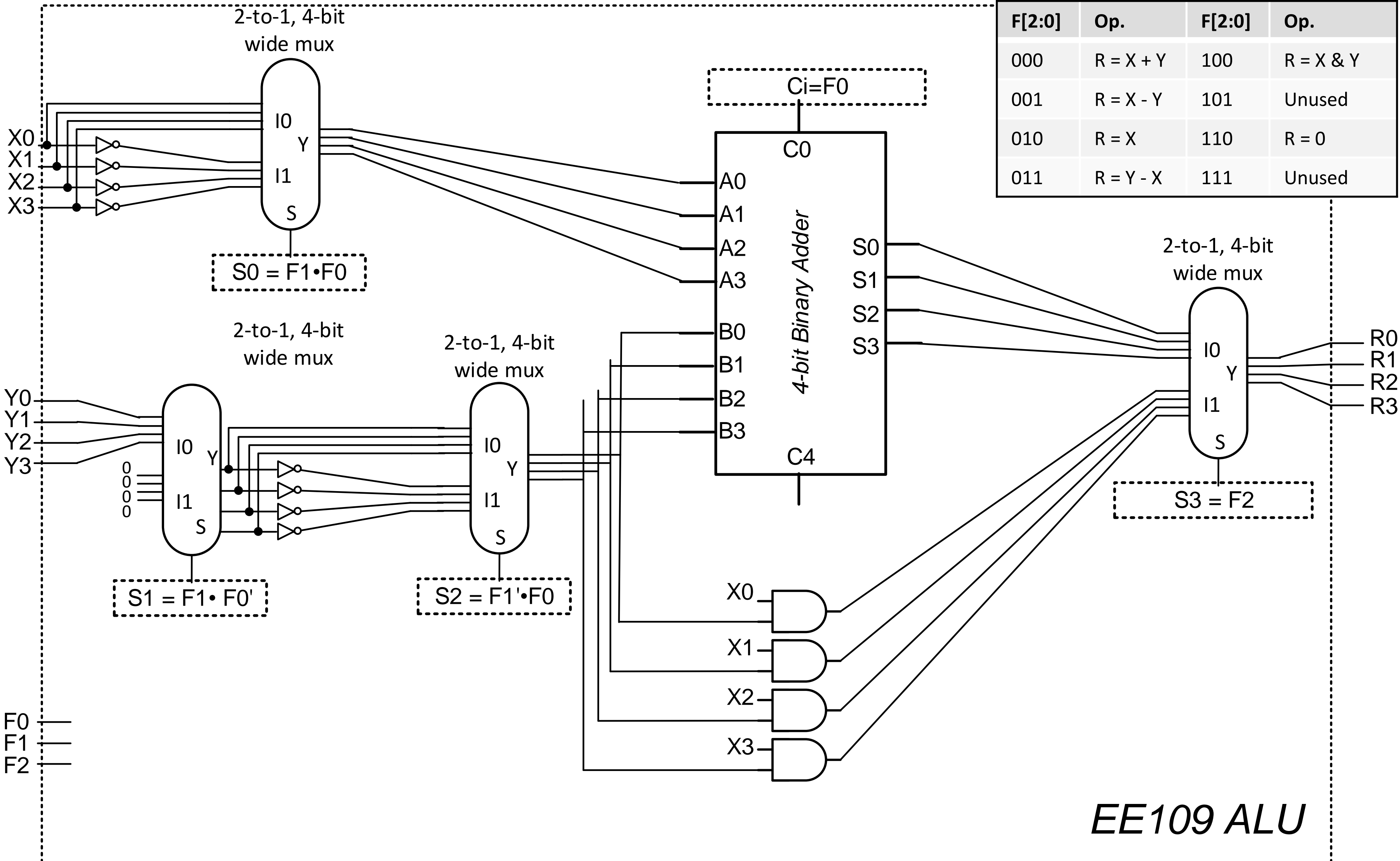
- S0 = F1•F0
- S1 = F1•F0'
- S2 = F1'•F0
- Ci = F0
- S3 = F2







Completed ALU



F[2:0]	Op.	F[2:0]	Op.
000	R = X + Y	100	R = X & Y
001	R = X - Y	101	Unused
010	R = X	110	R = 0
011	R = Y - X	111	Unused

Aside: Impacts of Coding (1)

- What if we changed the codes used for each operation?

We just made up these code assignments and the various operations. Remember, we definitely need to support ADD, SUB, AND, and CLR (R=0).

F[2:0]	Op./Result
000	R = X + Y
001	R = X - Y
010	R = X
011	R = Y - X
100	R = X & Y
101	Unused
110	R = 0
111	Unused

F[2:0]	Op./Result
000	R = X + Y
001	R = Y - X
010	R = X - Y
011	R = 0
100	R = X
101	R = X&Y
110	Unused
111	Unused

Aside: Impacts of Coding (2)

R	FS[2:0]	S0	S1	S2	Ci	S3
X + Y	000	0	0	0	0	0
Y - X	001	1	0	1	1	0
X - Y	010	0	0	1	1	0
0	011	d	1	0	d	1
X	100	0	1	0	0	0
X&Y	101	d	0	d	d	1
Unused	110	d	d	d	d	d
Unused	111	d	d	d	d	d

F2F1 \ F0	0	1
00		1
01	1	
11	d	d
10		d

F2F1 \ F0	0	1
00		1
01	1	d
11	d	d
10		d

- S0 = F0
- S1 = F2F0' + F1F0
- S2 = F1F0' + F1'F0
- Ci = F1+F0
- S3 = F1F0 + F2F0

F2F1 \ F0	0	1
00		1
01		d
11	d	d
10		d

F2F1 \ F0	0	1
00		
01		1
11	d	d
10	1	

F2F1 \ F0	0	1
00		
01		1
11	d	d
10		1

Notice how much more logic this coding yields.