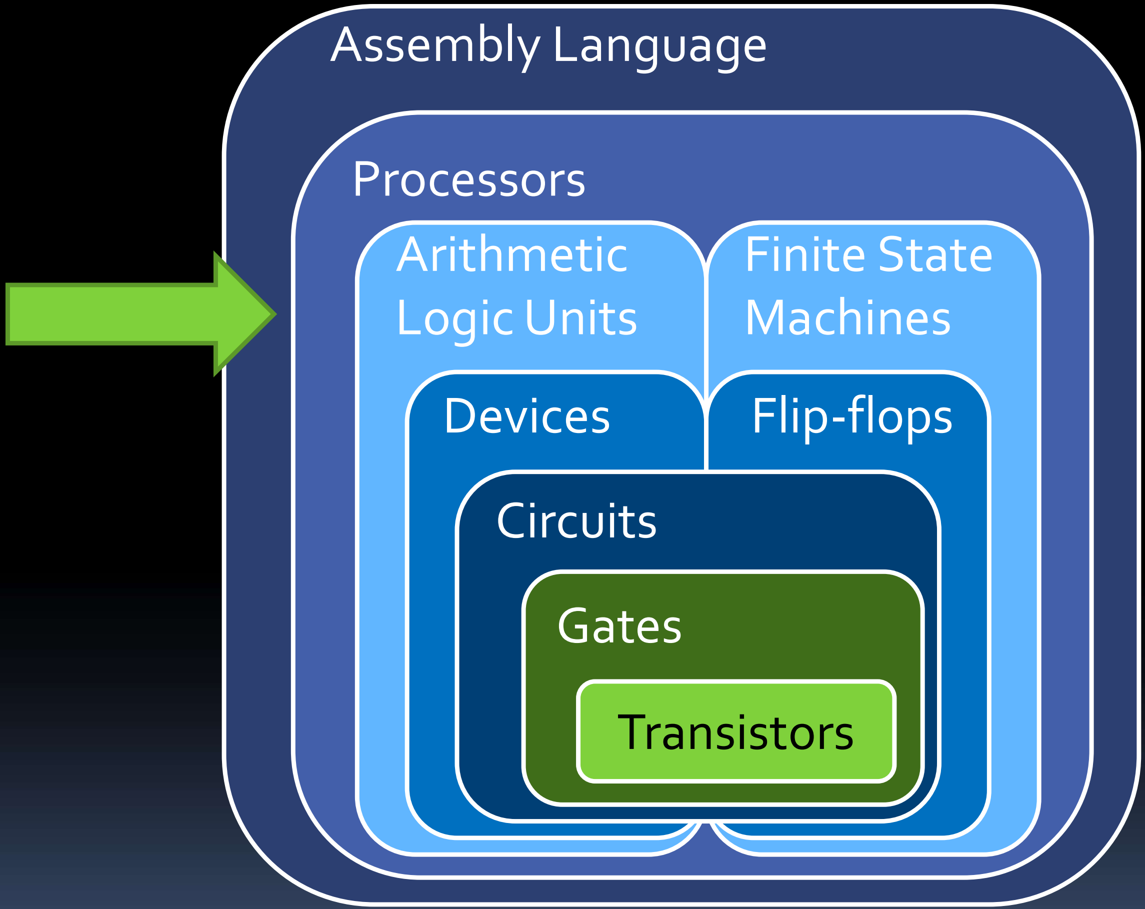
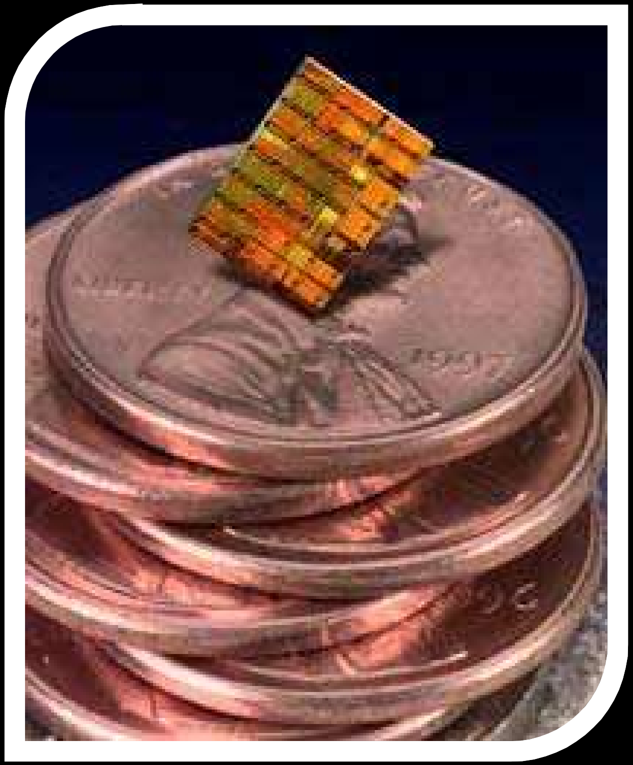
Processor Components

Where we are now

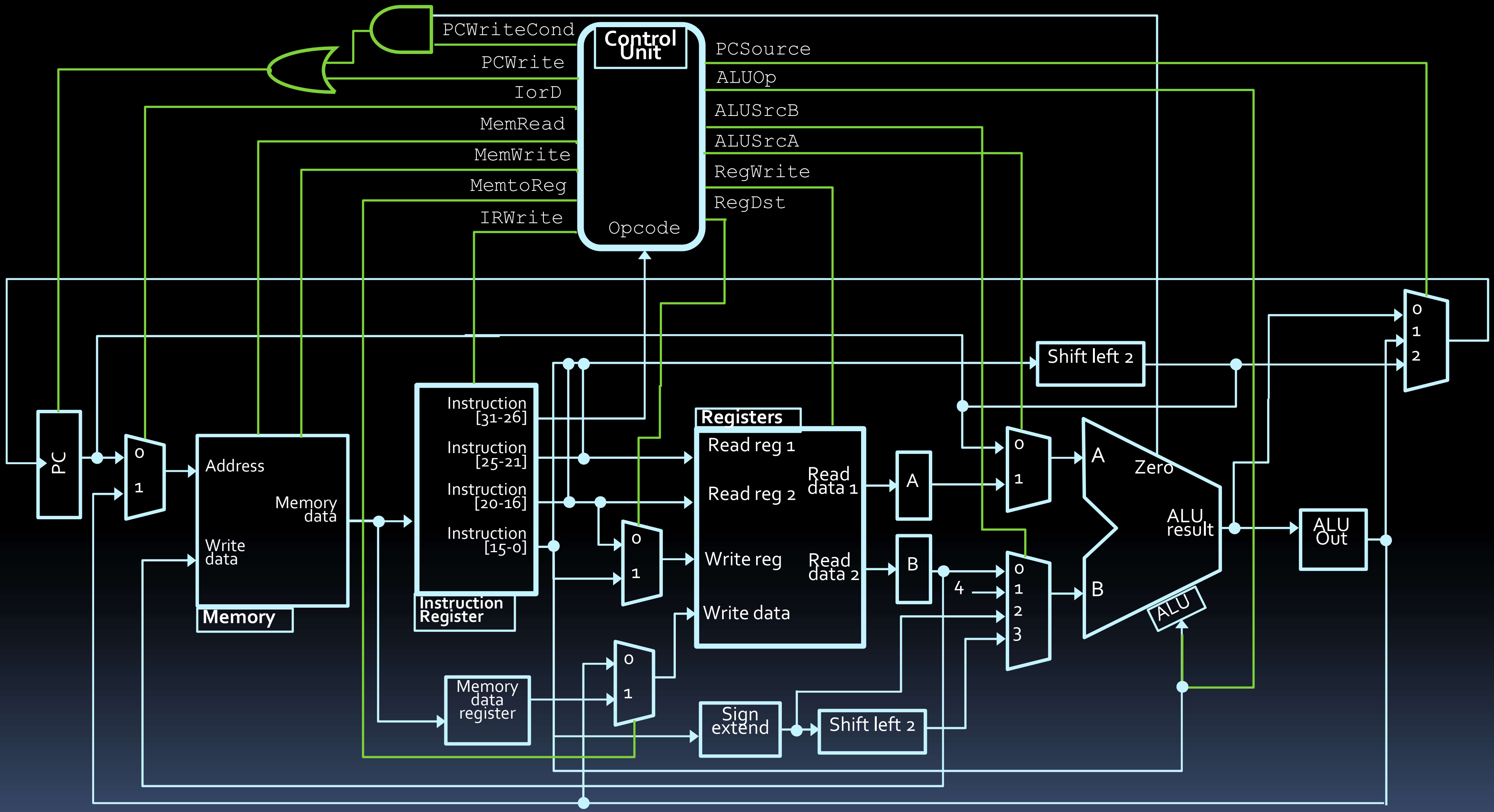


Microprocessors



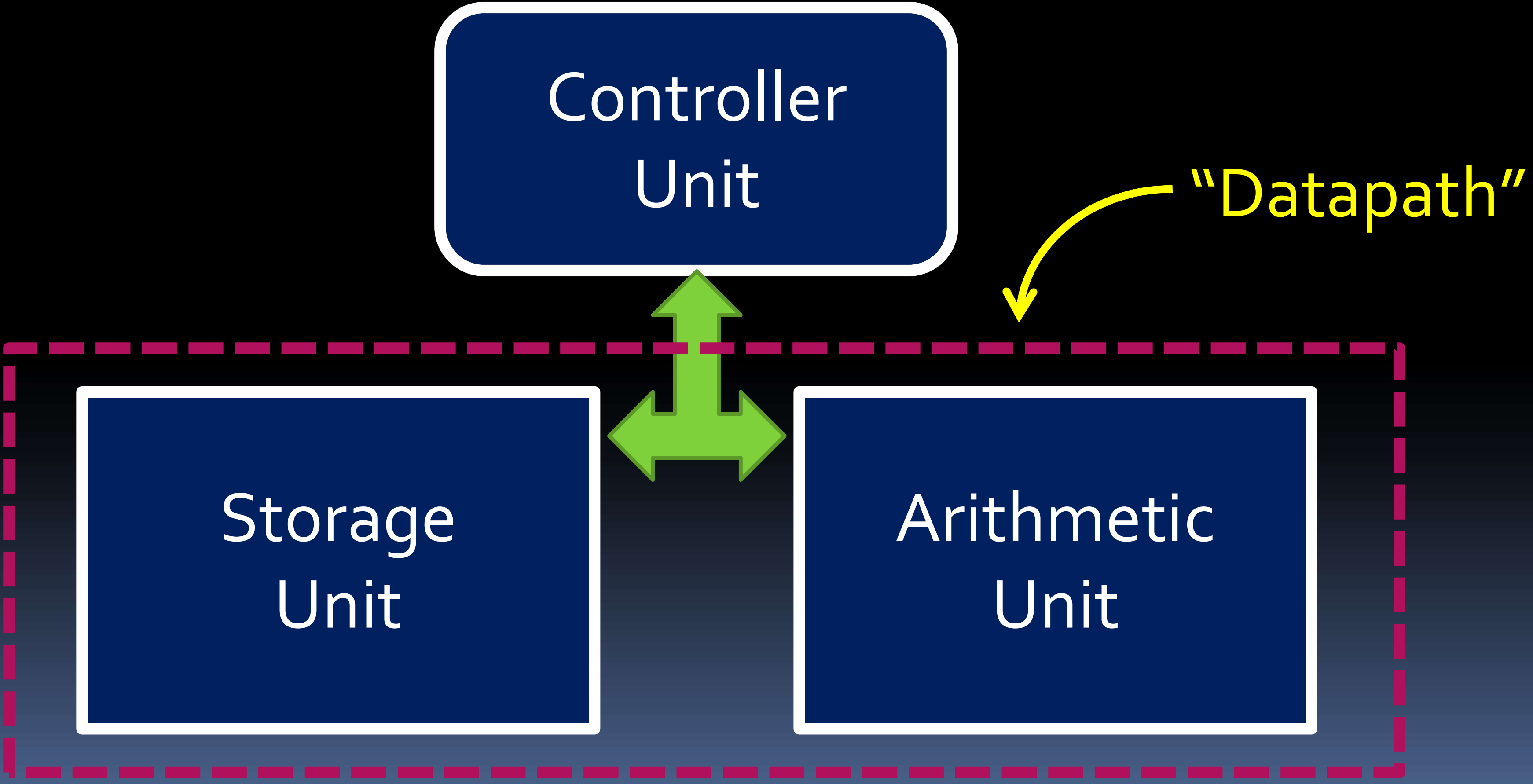
- So far, we've been talking about making devices, such as adders, counters and registers.
- The ultimate goal is to make a microprocessor, which is a digital device that processes input, can store values and produces output, according to a set of on-board instructions.

The Final Destination



Deconstructing processors

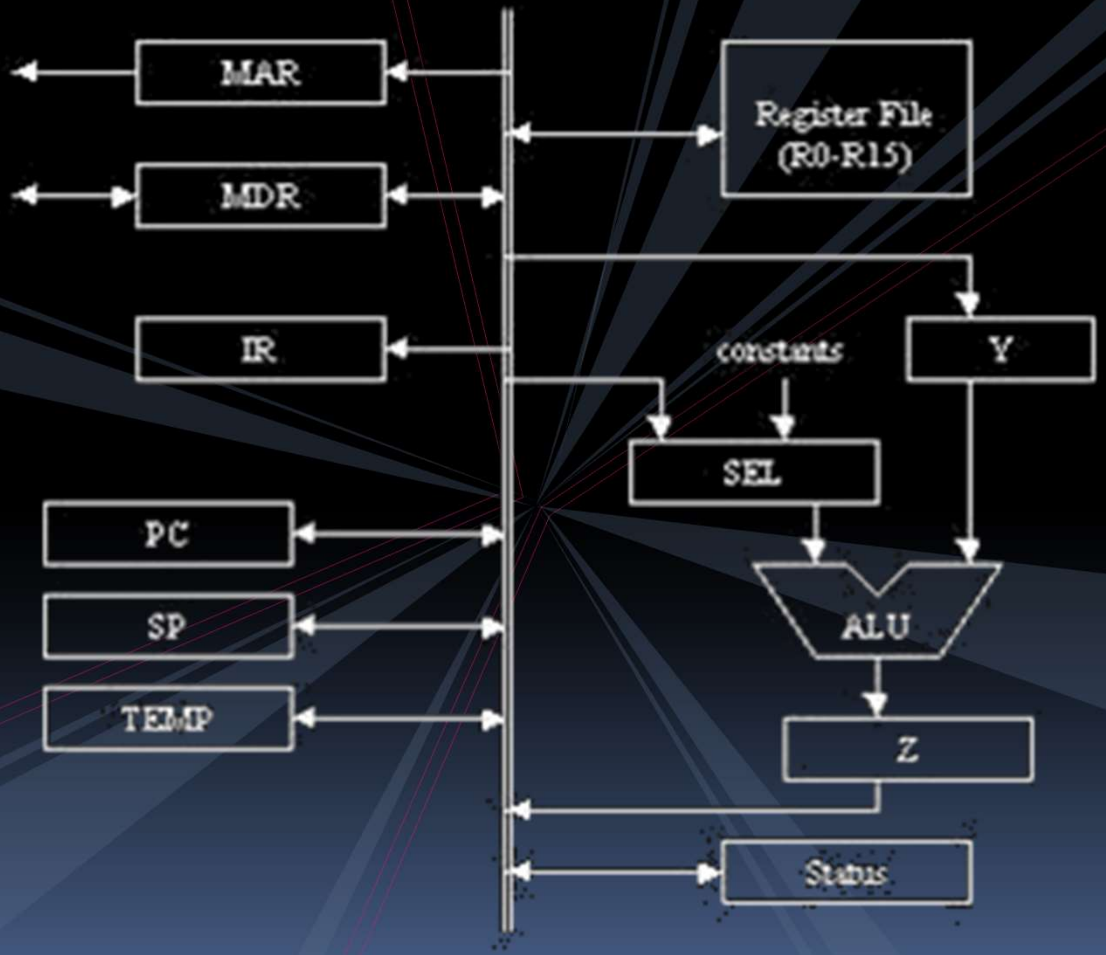
- Simpler at a high level:



Datapath vs. Control

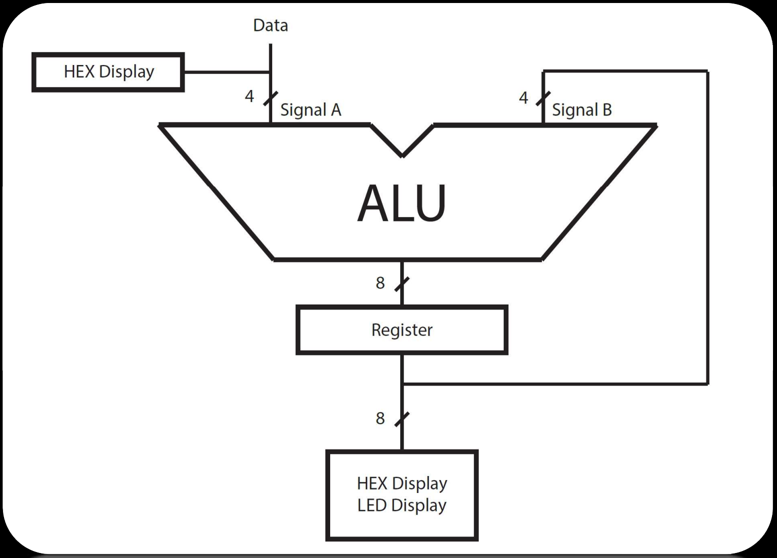
- Datapath: where all data computations take place.
  - Often a diagram version of real wired connections.
- Control unit: orchestrates the actions that take place in the datapath.
  - The control unit is a big finite-state machine that instructs the datapath to perform all appropriate actions.

Datapath example



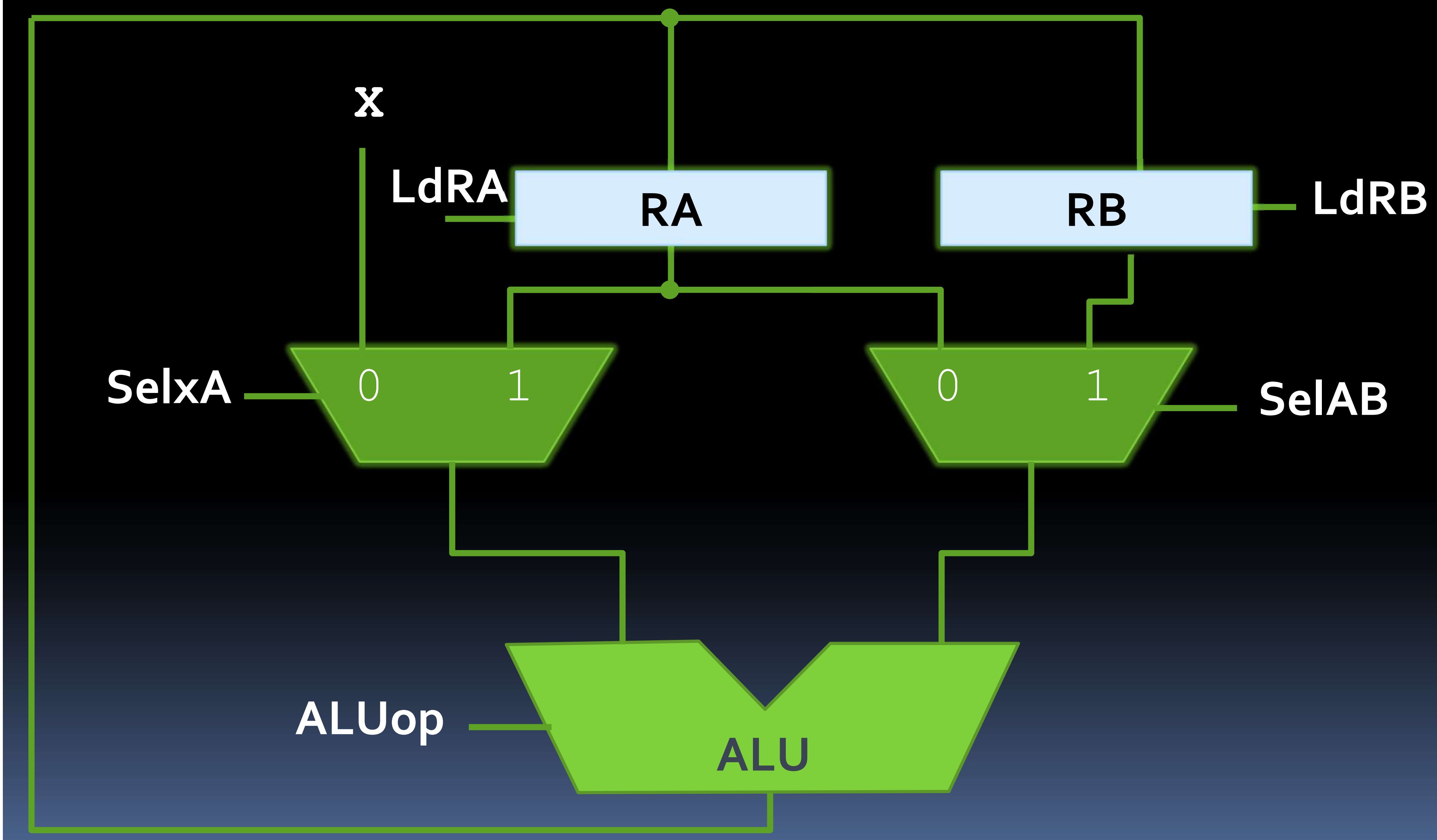
Thinking back to Lab 4

- Recall that you created an ALU an a register for Lab 4.
  - The ALU stores its output into the register,
  - The register provides an input value to the ALU.



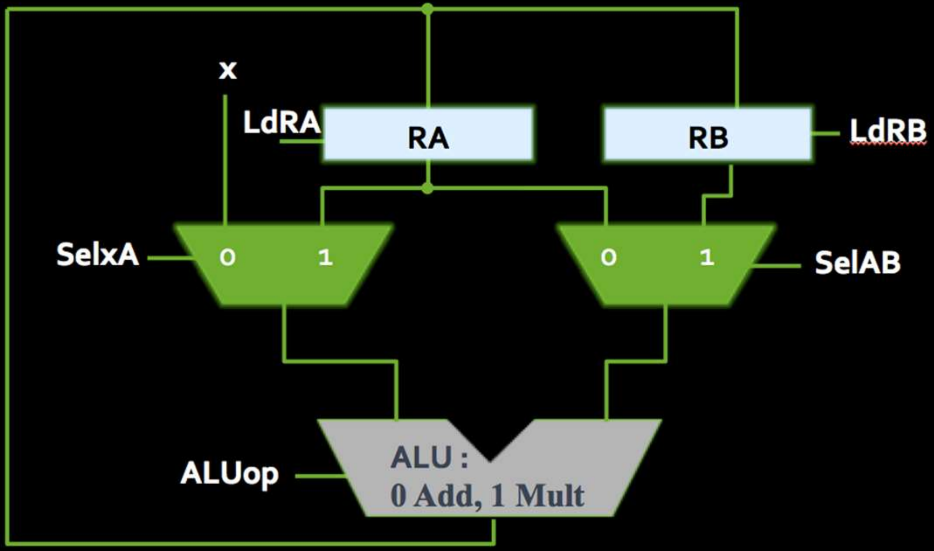
- What if there were was more than one register available to store ALU output values?
  - How do we indicate which register to write to?
  - How do we determine the ALU inputs?

Lab 4 ALU with 2 registers



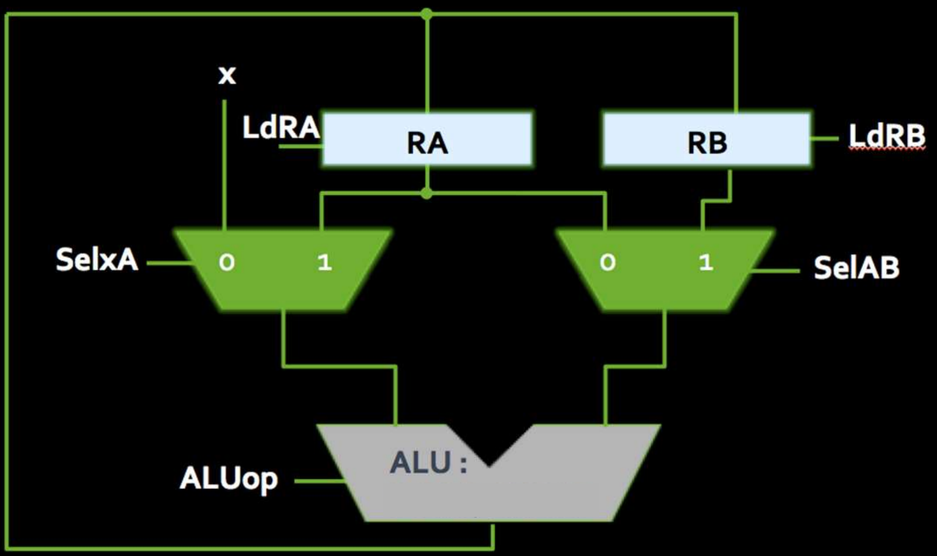
Lab 4 ALU with 2 registers

- Things to consider:
  - With 2 registers, we need to indicate which register (if any) will store the ALU output value.
    - The register write signals LdRA and LdRB handle this.
  - We also need to indicate whether X, register A, or register B will provide the input values to the ALU.
    - The multiplexers determine the inputs A and B of the ALU by setting the values for SelxA and SelAB



Datapath signals

- Use input signals to operate this datapath:
  - LdRA = Load Reg A
  - LdRB = Load Reg B
  - ALUop = Add vs Mult
  - SelxA = First ALU input
    - 0 → input comes from X, 1 → input comes from RA
  - SelAB = Select second ALU input
    - 0 → input comes from RA, 1 → input comes from RB



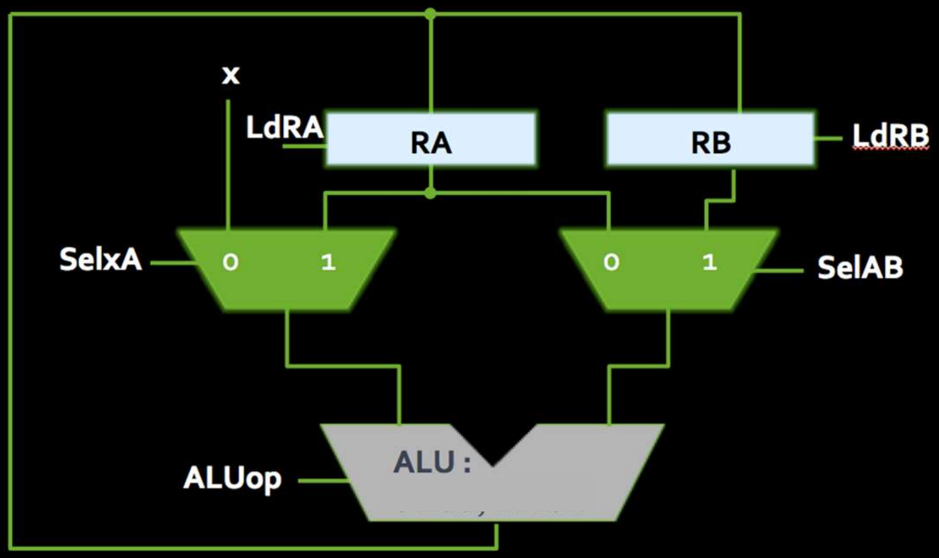
- How do we use this to perform a calculation?

Example: Calculating $x^2 + 2x$

- Let's assume we have an external value X, supplied from outside the datapath.
  - How do we use the datapath to calculate $x^2 + 2x$?
- Need to coordinate the datapath components:
  - ALU (to add, subtract and multiply values)
  - Multiplexers (to determine what the inputs should be to the ALU)
  - Registers (to hold values used in the calculation)

Calculating $x^2 + 2x$

- To calculate $x^2 + 2x$:
  - Load X into RA & RB
  - Multiply RA by RB
    - Store result in RA
  - Add X to RA
    - Store result in RA
  - Add X to RA again
    - Result from ALU is $x^2 + 2x$.



- How do we make these operations happen?

Making the calculation

High-level Steps

- Load X into RA & RB
- Multiply RA & RB
  - Store result in RA
- Add X to RA
  - Store result in RA
- Add X to RA again
  - ALU output is $x^2 + 2x$.

Control Signals

Transfer operation; will discuss shortly

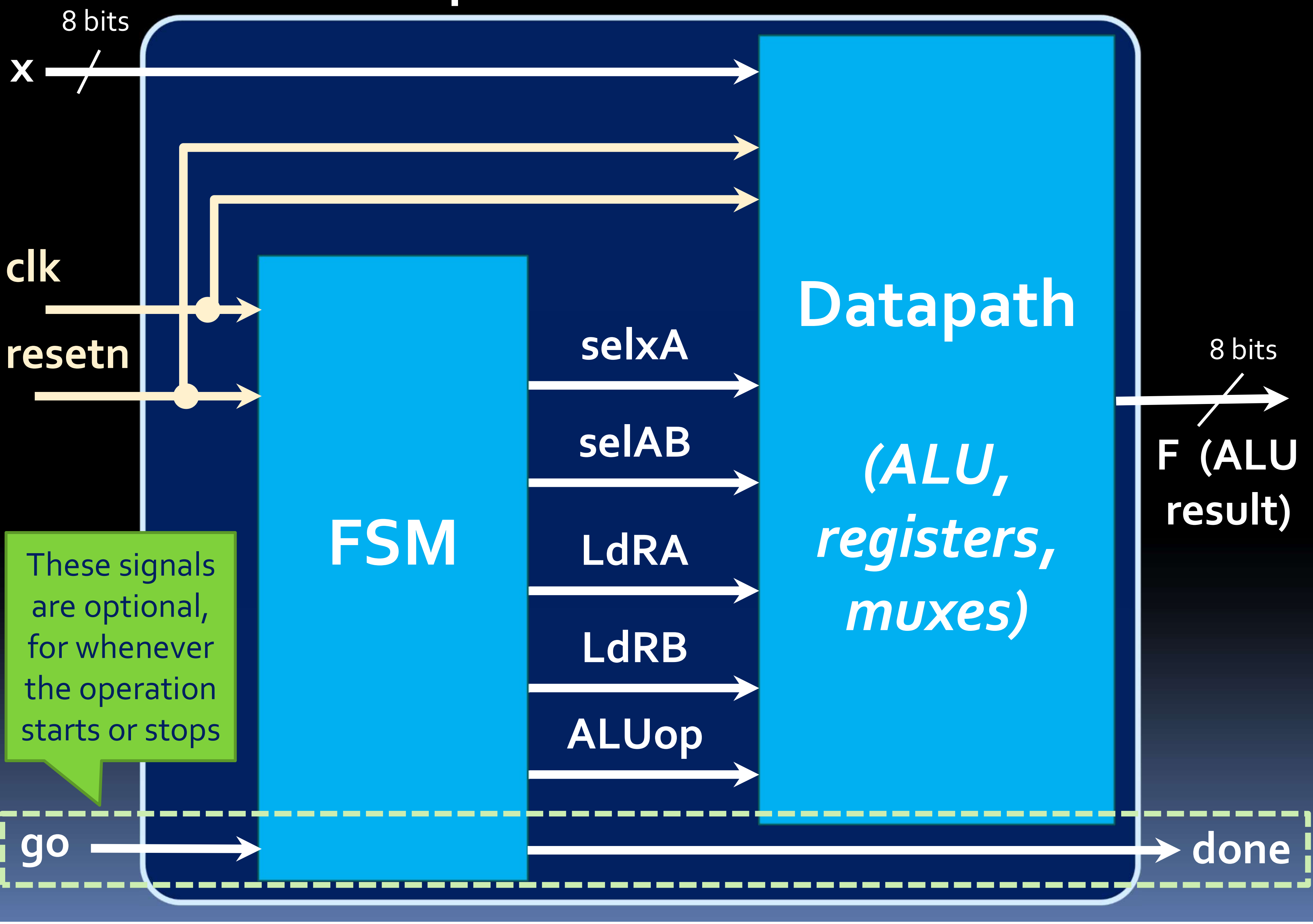
- SelxA = 0, ALUop = A, LdRA = 1, LdRB = 1
- SelxA = 1, SelAB = 1, ALUop = Mult, LdRA = 1
- SelxA = 1, SelAB = 1, ALUop = Add, LdRA = 1
- SelxA = 1, SelAB = 1, ALUop = Add

- **Who sends these signals?**

The Control Unit

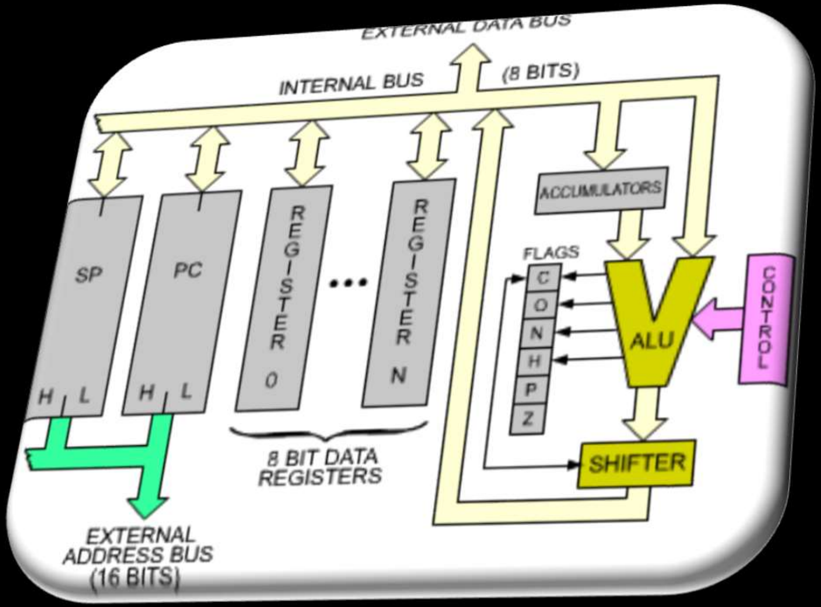
- Essentially, a giant Finite State Machine
  - Synchronized to system-wide signals (clock, resetn)
- Outputs the datapath control signals
  - SelxA, SelAB => control mux outputs (ALU inputs)
  - ALUop => controls ALU operation
  - LdRA, LdRB => controls loading for registers RA, RB
- Some architectures also output a done signal, when the computation is complete
  - Yet another output; not shown in our datapaths

Datapath + Control



These signals are optional, for whenever the operation starts or stops

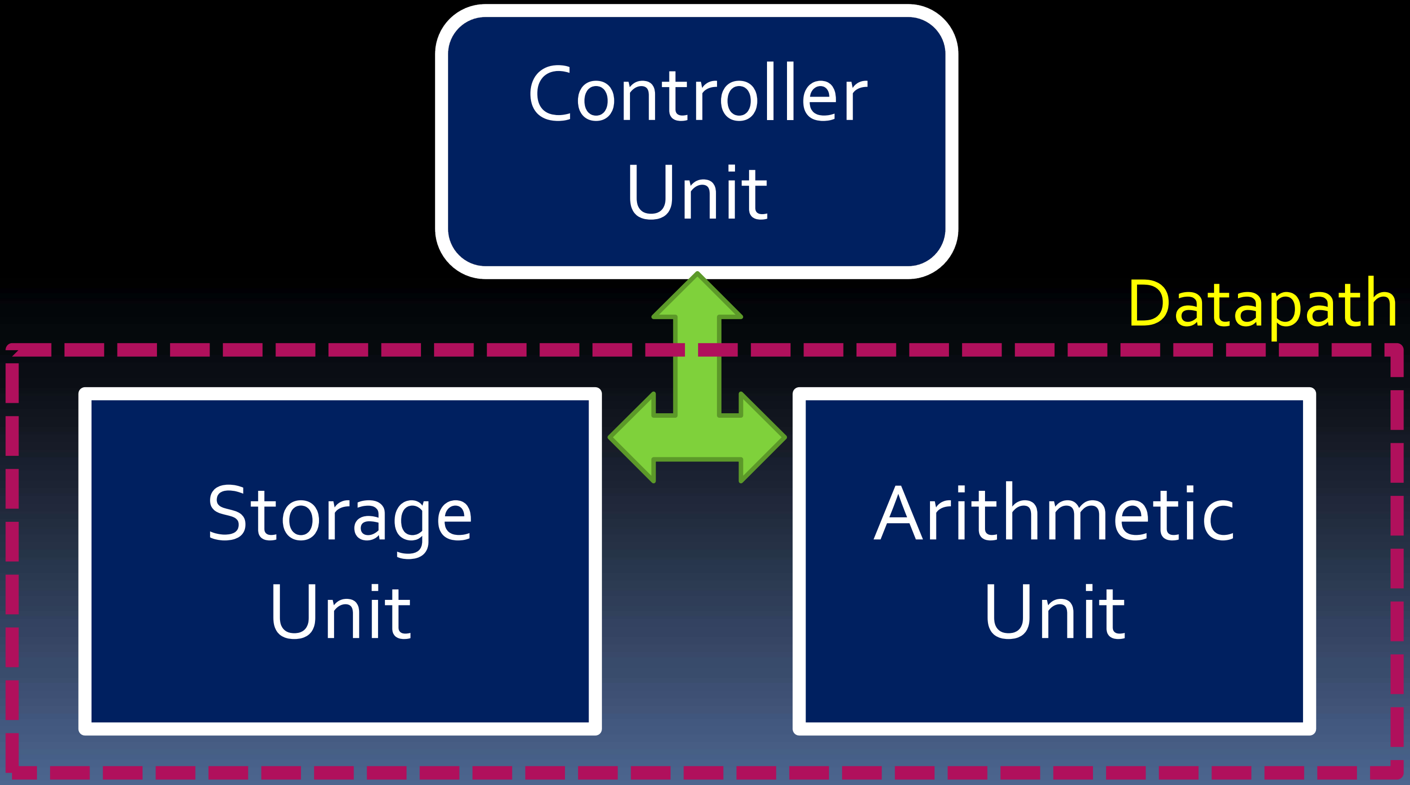
Microprocessors



- This datapath example is a combination of the major devices we’ve discussed so far:
  - Registers to store values.
  - An ALU (adders, shifters, etc) to process data.
  - Finite state machines to control the process.
- Microprocessors are the basis of all computing since the 1970’s, and can be found in nearly every sort of electronics.

Microprocessor components

- To understand the full microprocessor, we'll visit each major component in depth.
  - Starting with the arithmetic unit.



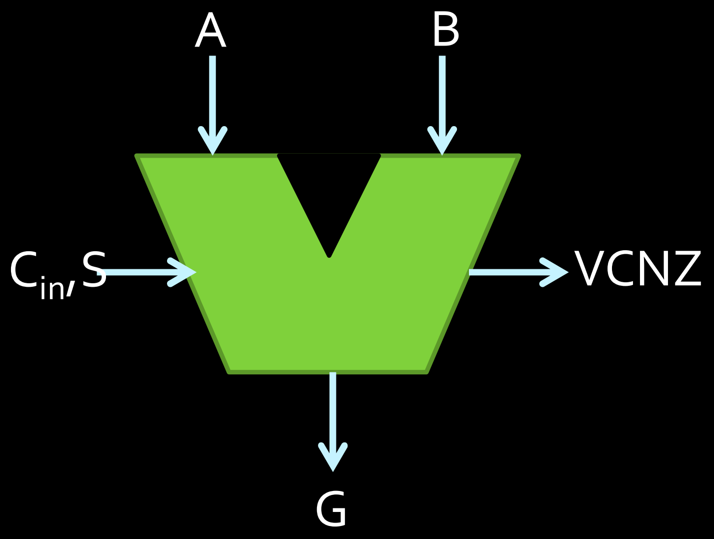
The Arithmetic Unit

aka: the Arithmetic Logic Unit (ALU)

Arithmetic Logic Unit

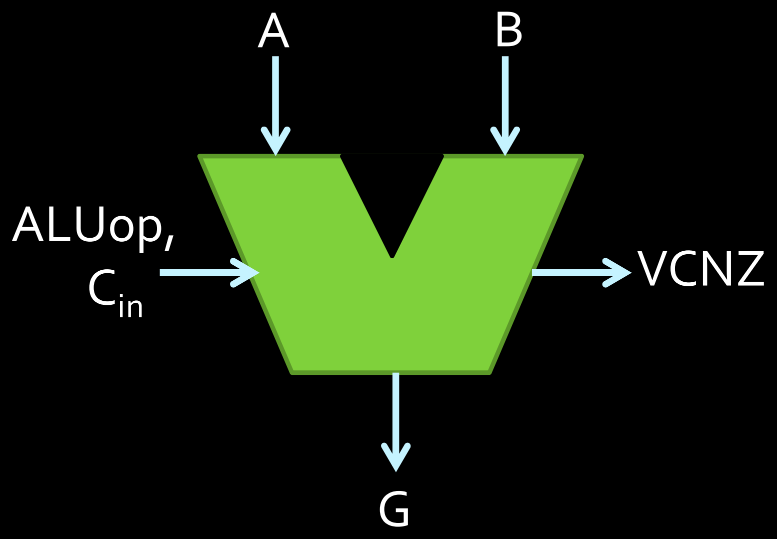
- The first microprocessor applications were calculators.
  - Recall the unit on adders and subtractors.
  - These are part of a larger structure called the Arithmetic Logic Unit (ALU).
    - Like the ones you made for the labs.
- This larger structure is responsible for the processing of all data values in a basic CPU.

ALU inputs



- The ALU performs all of the arithmetic operations covered in this course so far, and logical operations as well (AND, OR, NOT, etc.)
  - Input S represents select bits (in this case, S_2 S_1 & S_o) that specify the operation to perform.
  - The carry bit C_{in} is used in operations such as incrementing an input value or the overall result.

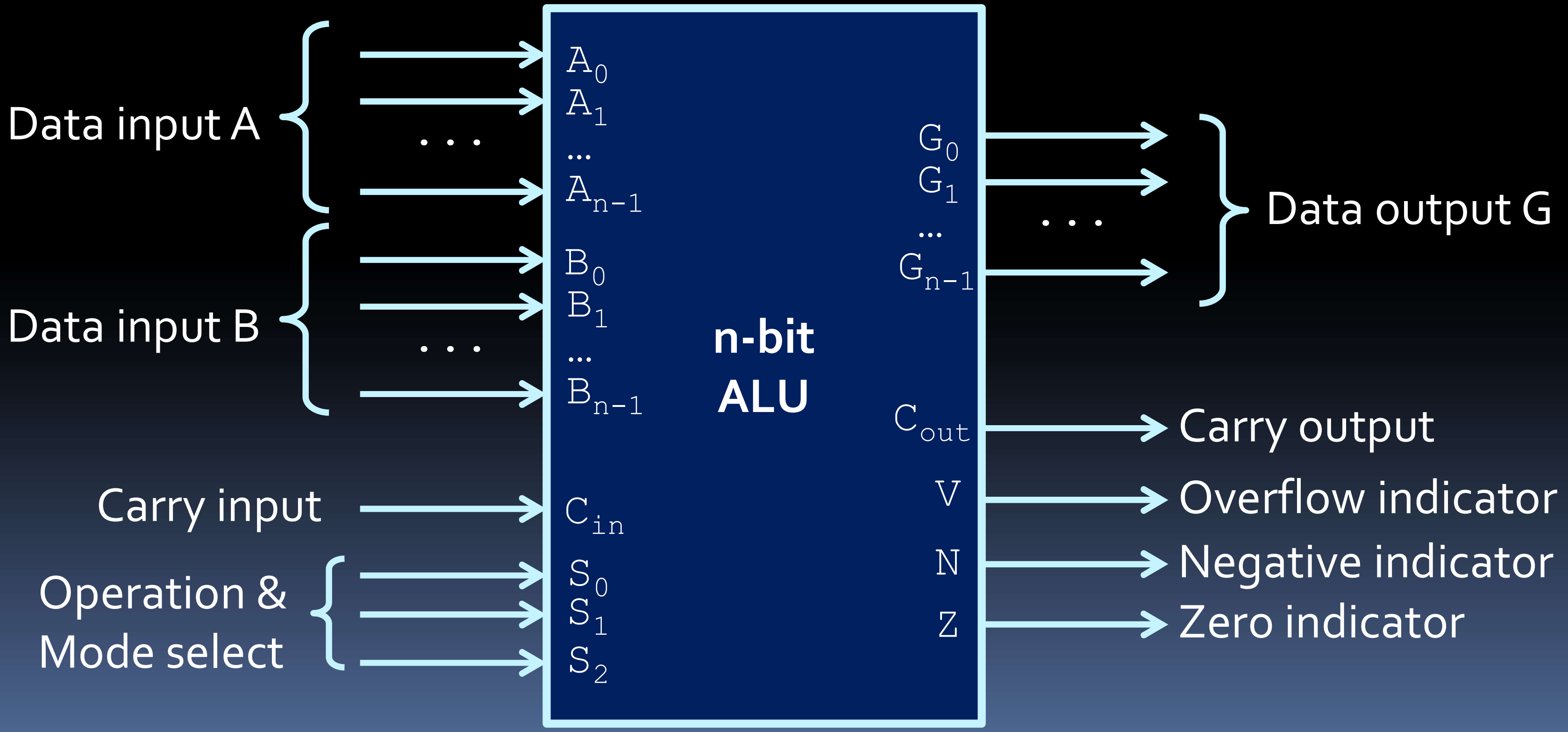
ALU outputs



- In addition to the input signals, there are output signals V, C, N & Z which indicate special conditions in the arithmetic result:
  - **V**: overflow condition
    - The result of the operation could not be stored in the n bits of G, meaning that the result is incorrect.
  - **C**: carry-out bit
    - Used to detect errors in unsigned arithmetic.
  - **N**: Negative indicator
  - **Z**: Zero-condition indicator

ALU block diagram

- In addition to data inputs and outputs, this circuit also has:
  - outputs indicating the different conditions,
  - inputs specifying the operation to perform (similar to Sub).

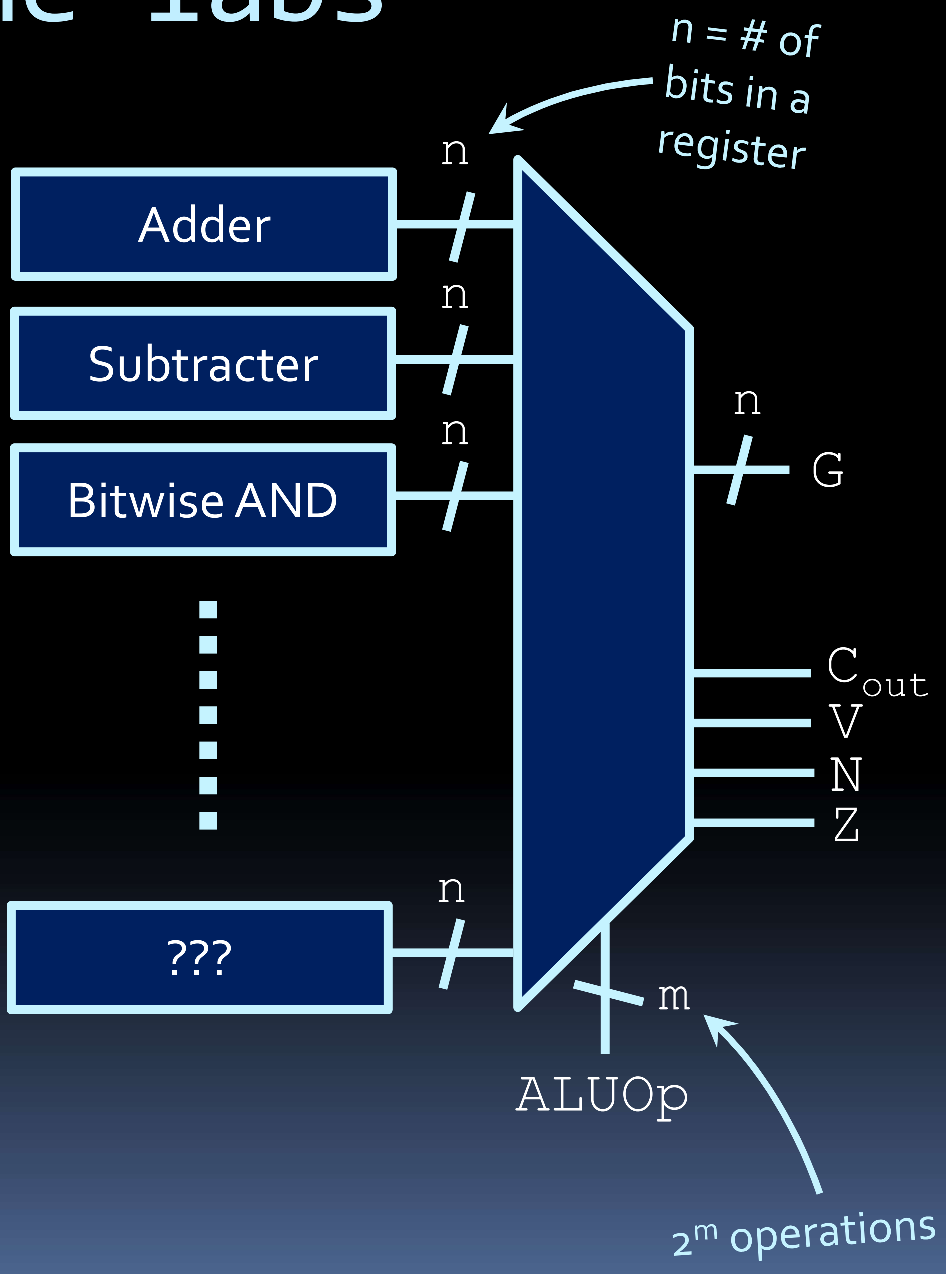


ALU Disclaimer

- There are multiple ways that the ALU can be implemented.
  - All implementations do the same general function (arithmetic and logical operations).
  - The operations that the ALU can perform, how it performs them, and specific input and output signals can vary.
- We will give you a couple of implementations (that you should learn), but just keep in mind that others are possible as well.

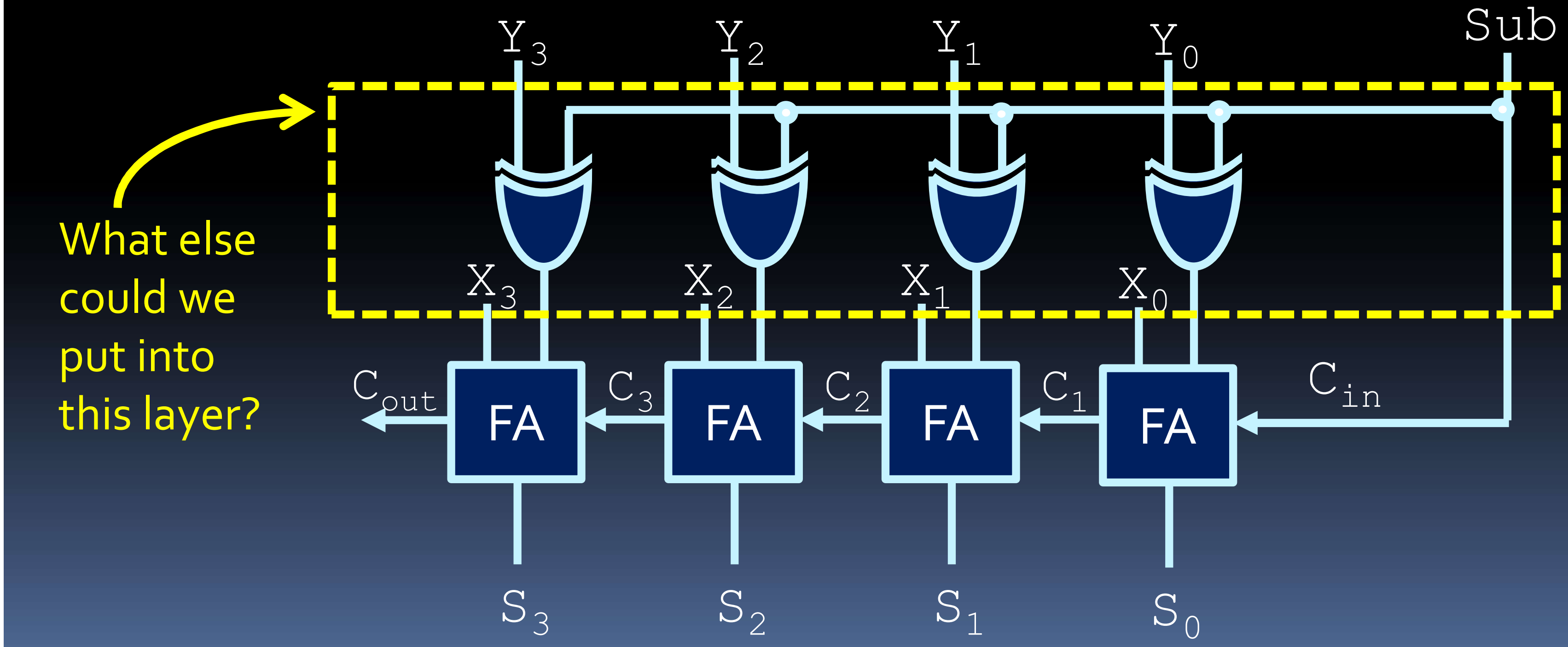
The ALU from the labs

- This design is easy to see and build, but not optimized for size.
- ALU designs need to find ways to avoid duplicating logic and reducing the footprint.
  - Tools like Quartus reduce these designs automatically when mapping to gates.

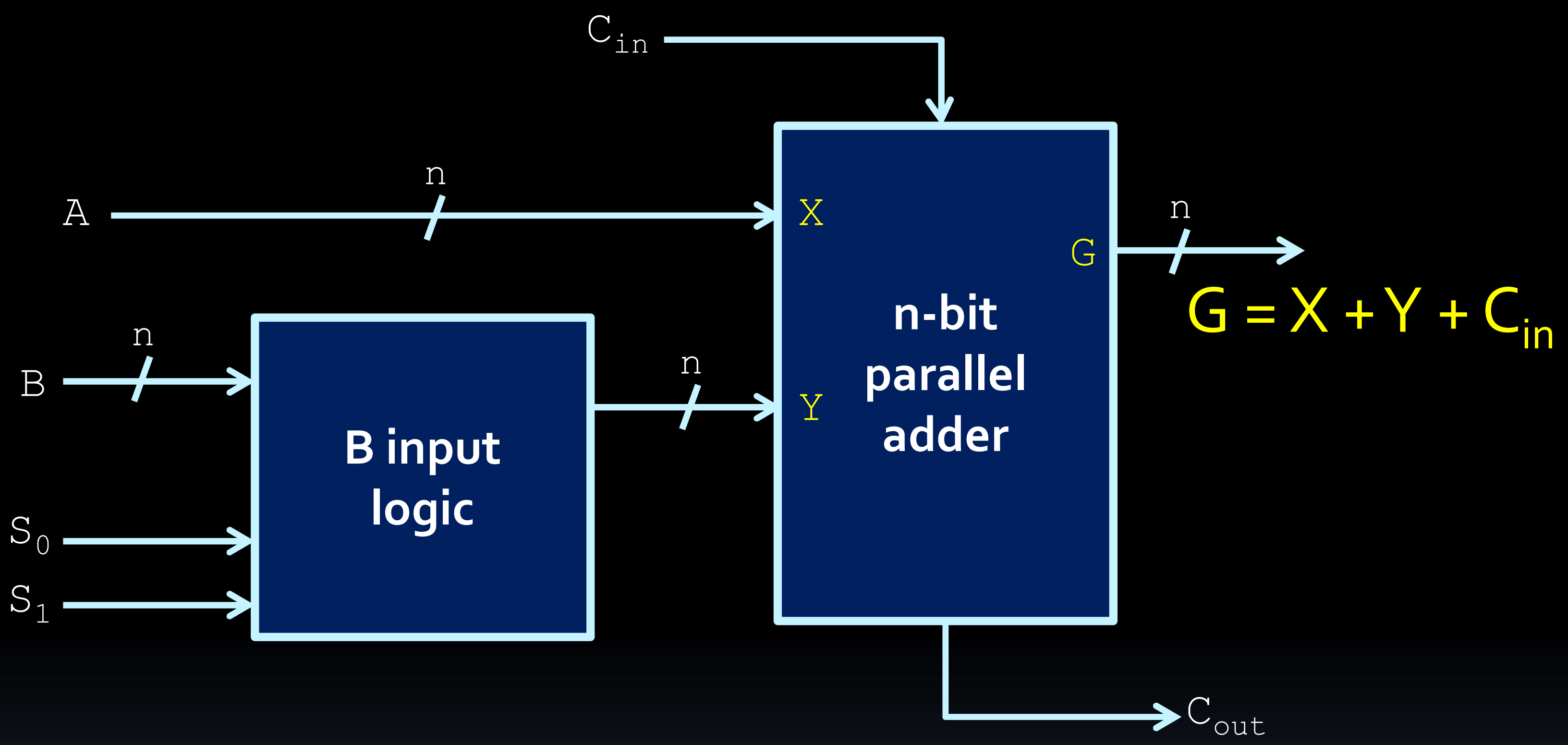


Revisiting the “A” of ALU

- When building the ALU operations from scratch, we start with the arithmetic side.
- Fundamentally, this side is made of an adder / subtractor unit, which we’ve seen already:

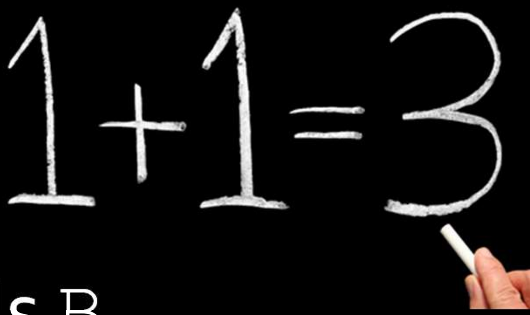


Arithmetic components



- In addition to addition and subtraction, many more operations can be performed by manipulating what is added to input A, as shown in the diagram above.

Arithmetic operations



- If the input logic circuit on the left sends B straight through to the adder, result is $G = A+B$
- What if B was replaced by all ones instead?
  - Result of addition operation: $G = A-1$
- What if B was replaced by $\overline{B}$ and C_{in} was high?
  - Result of addition operation: $G = A-B$
- And what if B was replaced by all zeroes?
  - Result is: $G = A$. (Not interesting, but useful!)

→ Instead of a Sub signal, the operation you want is signaled using the select bits S_0 & S_1.

Operation selection

Select bits		Y input	Result	Operation
S_1	S_0			
0	0	All 0s	G = A	Transfer
0	1	B	G = A+B	Addition
1	0	$\overline{B}$	$G = A+\overline{B}$	Subtraction - 1
1	1	All 1s	G = A-1	Decrement

- This is a good start! But something is missing…
- Wait, what about the carry bit?

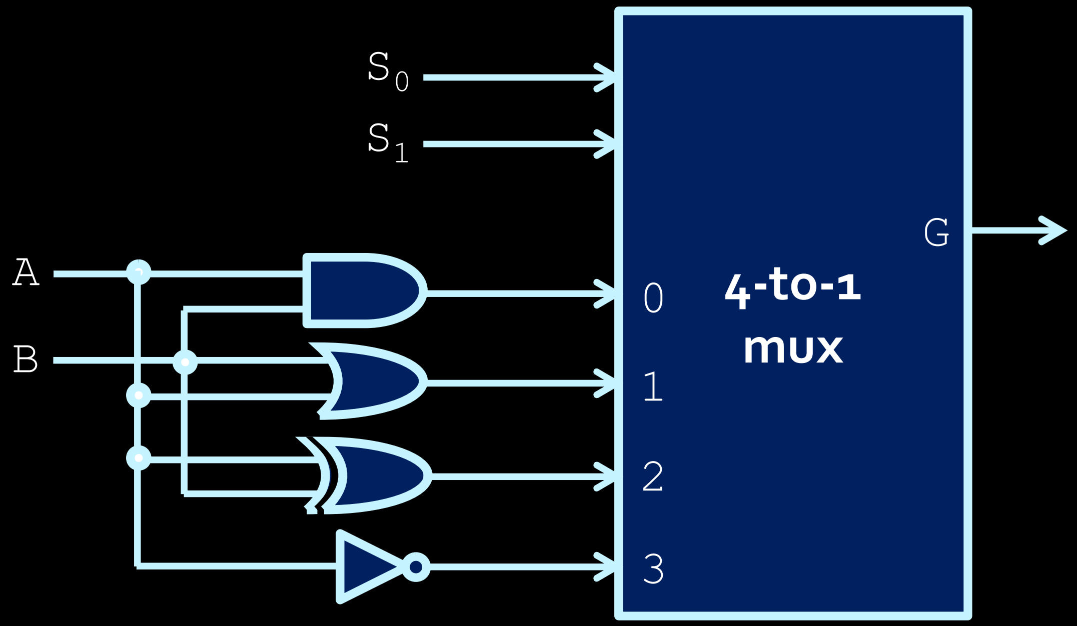
Full operation selection

Select		Input	Operation	
S_1	S_0	Y	$C_{in}=0$	$C_{in}=1$
0	0	All 0s	$G = A$ (transfer)	$G = A+1$ (increment)
0	1	B	$G = A+B$ (add)	$G = A+B+1$
1	0	$\overline{B}$	$G = A+\overline{B}$	$G = A+\overline{B}+1$ (subtract)
1	1	All 1s	$G = A-1$ (decrement)	$G = A$ (transfer)

- Based on the values on the select bits and the carry bit, we can perform any number of basic arithmetic operations by manipulating what value is added to A.

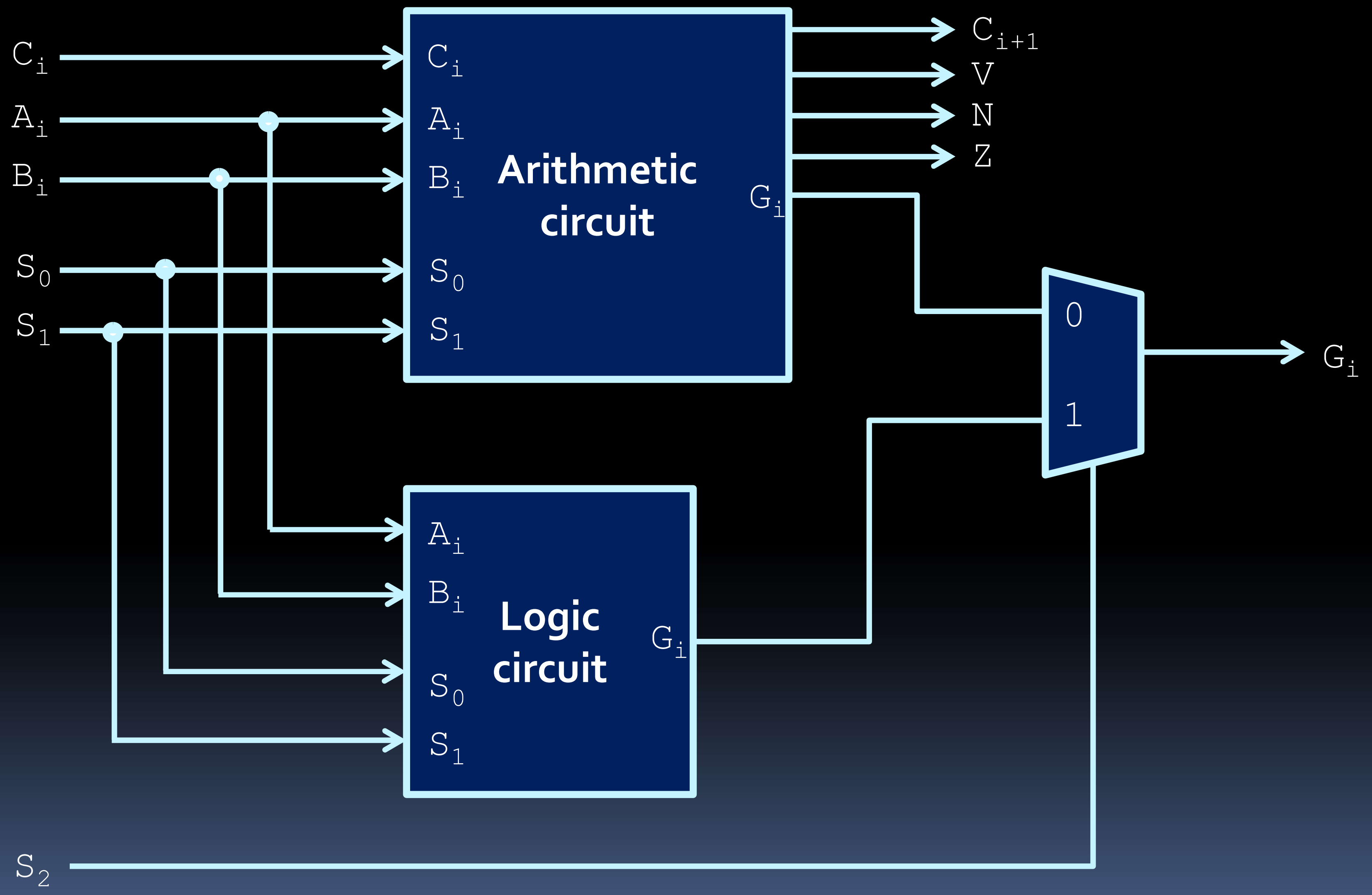
The “L” of ALU

- We also want a circuit that can perform logical operations, in addition to arithmetic ones.
- How do we tell which operation perform?
  - Another select bit!



- If $S_2 = 1$, then the output of the logic circuit block appears at the ALU output.
- Multiplexer is used to determine which block (logical or arithmetic) goes to the output.

ALU: Arithmetic + Logic

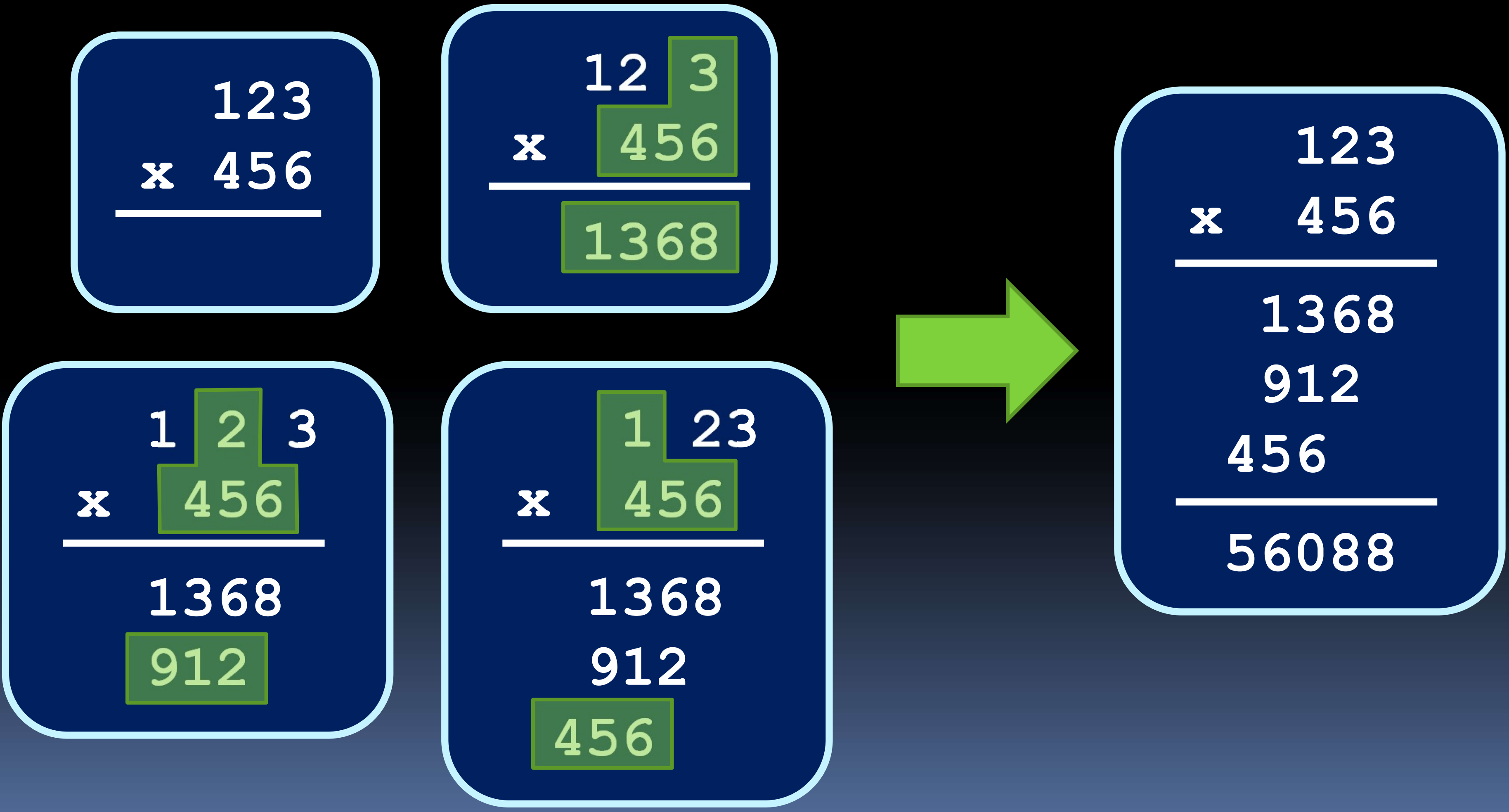


What about multiplication?

- Multiplication (and division) operations are always more complicated than other arithmetic (plus, minus) or logical (AND, OR) operations.
- Three major ways that multiplication can be implemented in circuitry:
  - Layered rows of adder units.
  - An adder/shifter circuit
  - Booth’s Algorithm

Binary Multiplication

- Revisiting grade 3 math…



```
   123
x  456
------
```

```
   12 3
x   456
-------
   1368
```

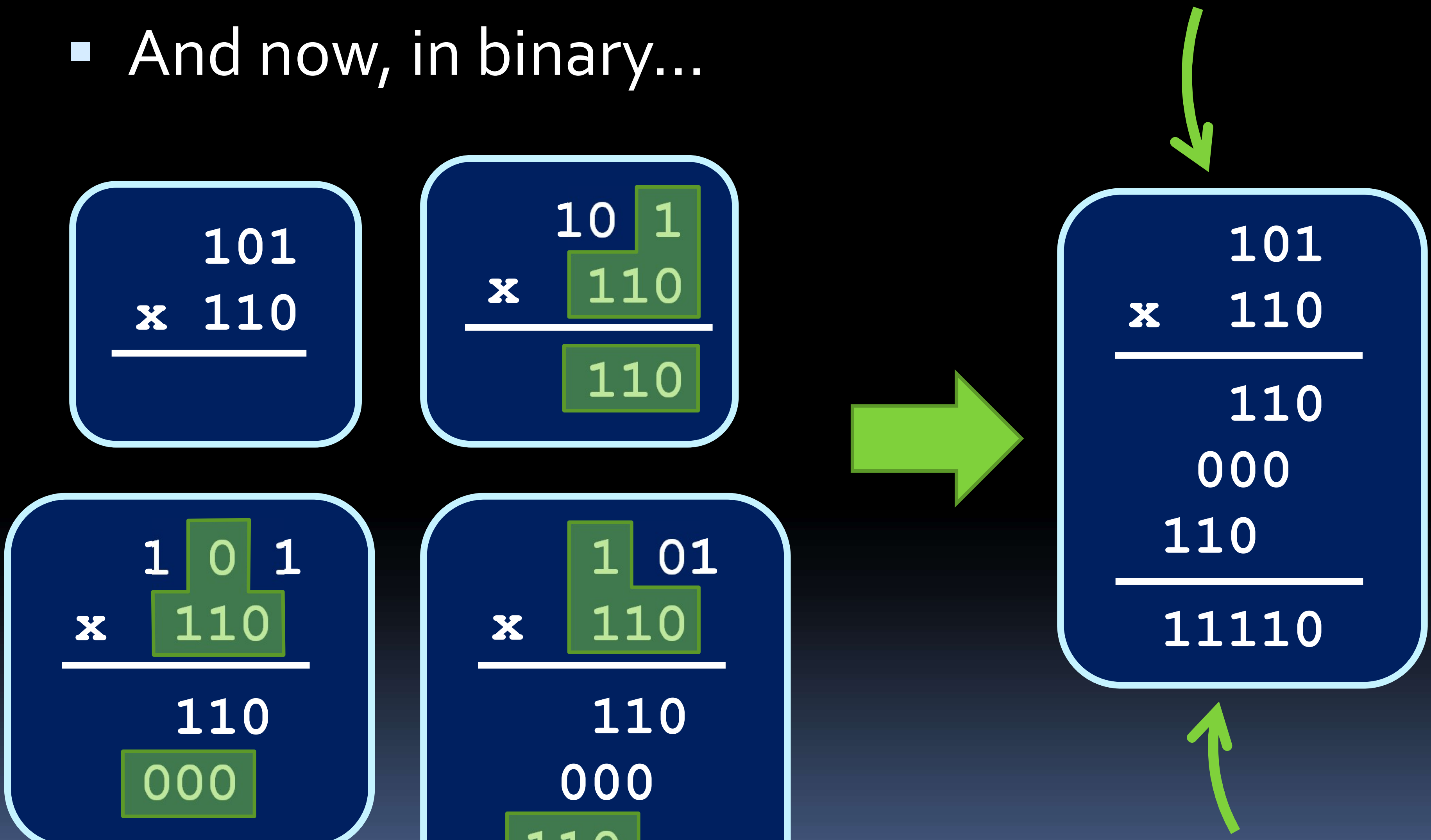
```
   1 2 3
x   456
--------
   1368
   912
```

```
   1 23
x   456
-------
   1368
   912
  456
```

```
     123
x    456
--------
    1368
    912
   456
--------
   56088
```

Binary Multiplication

- And now, in binary…



```
   101
x  110
```

```
   101
x  110
------
   110
```

```
   101
x  110
------
   110
  000
```

```
   101
x  110
------
   110
  000
 110
```

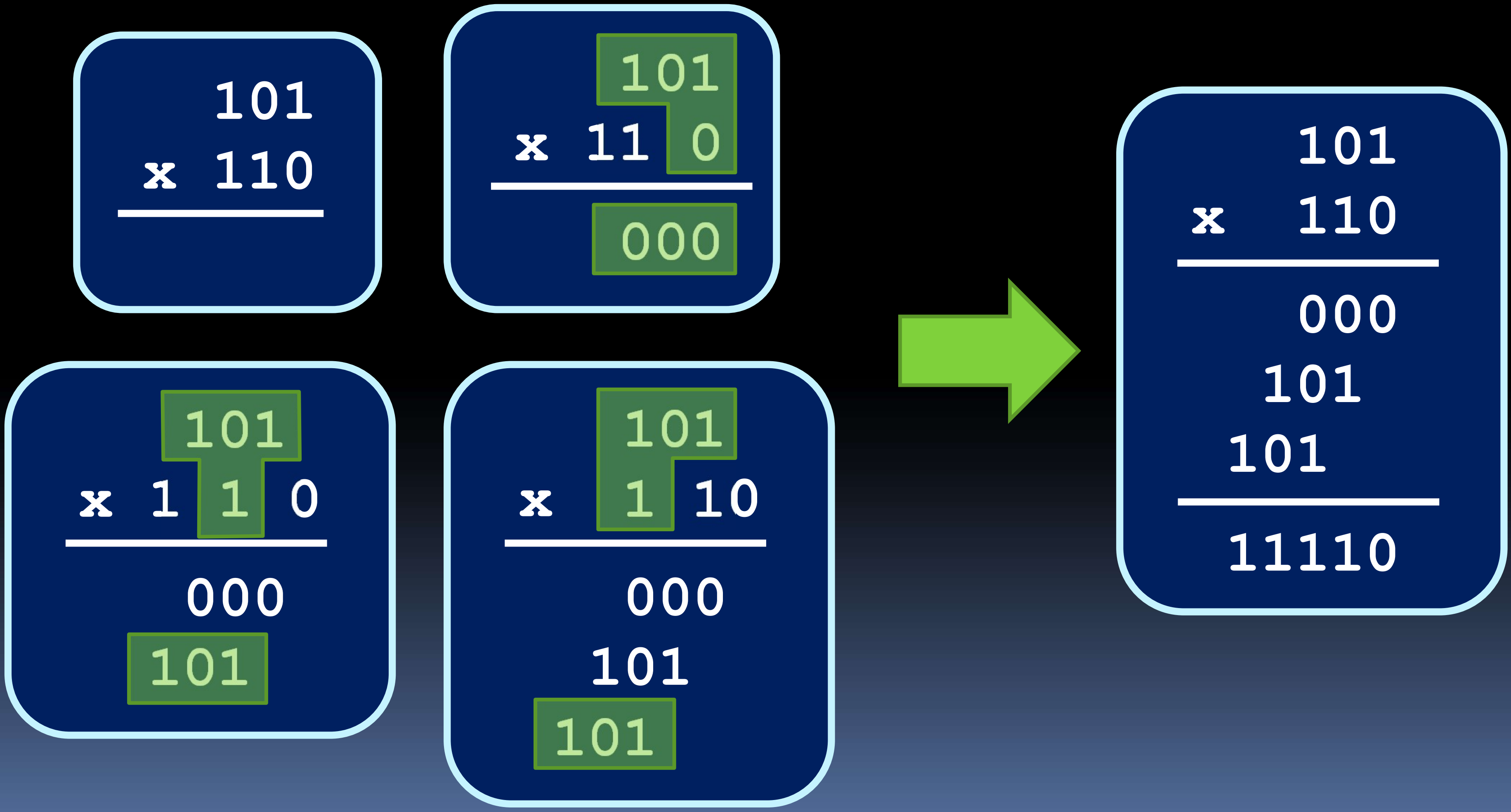
5*6 (unsigned)

```
    101
x   110
-------
    110
   000
  110
-------
  11110
```

Result: 30

Binary Multiplication

- Or seen another way….



```
   101
x  110
------
```

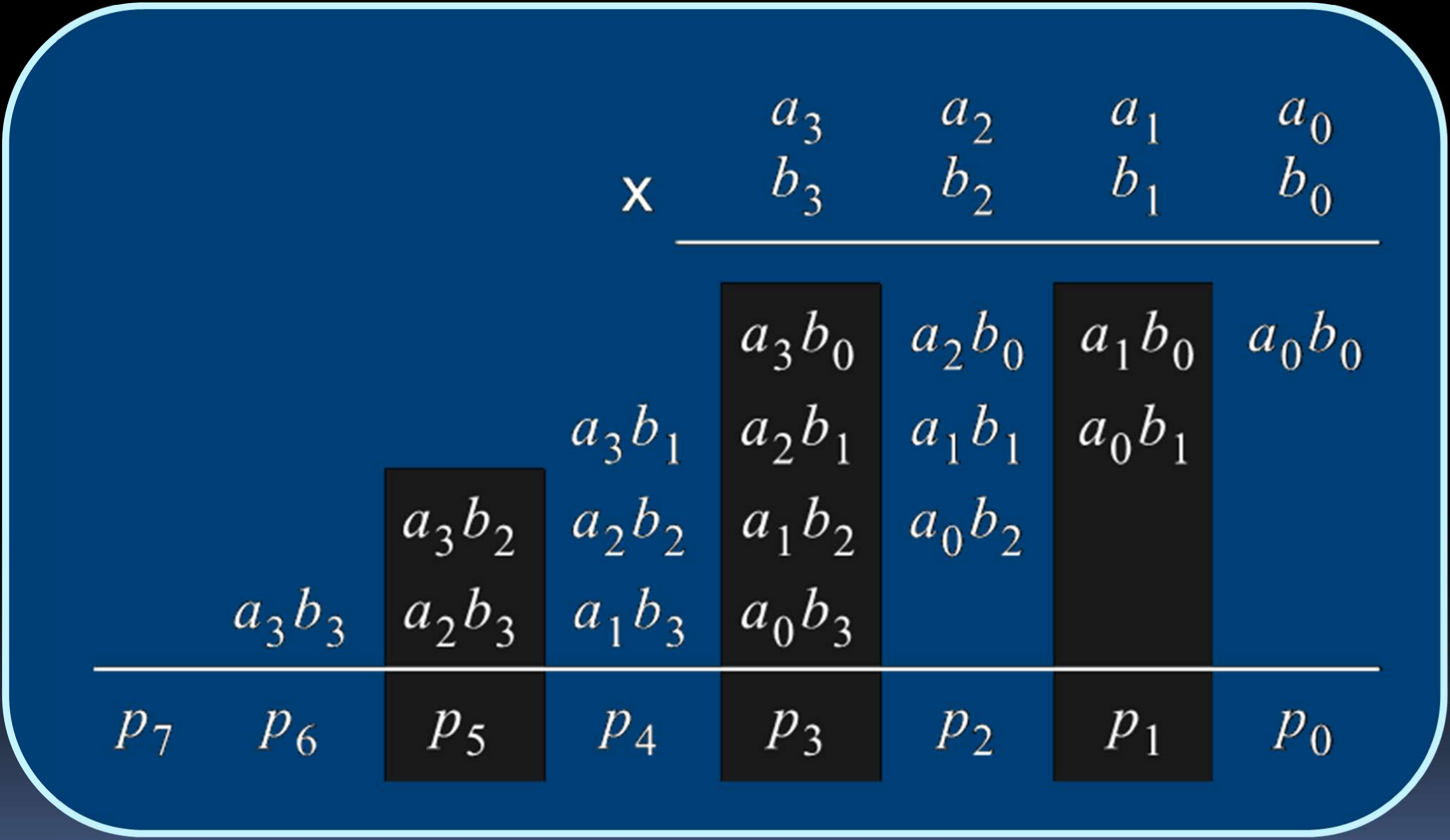
```
   101
x  11 0
-------
   000
```

```
    101
x  1 1 0
--------
    000
   101
```

```
    101
x   1 10
--------
    000
   101
  101
```

```
    101
x   110
-------
    000
   101
  101
-------
  11110
```

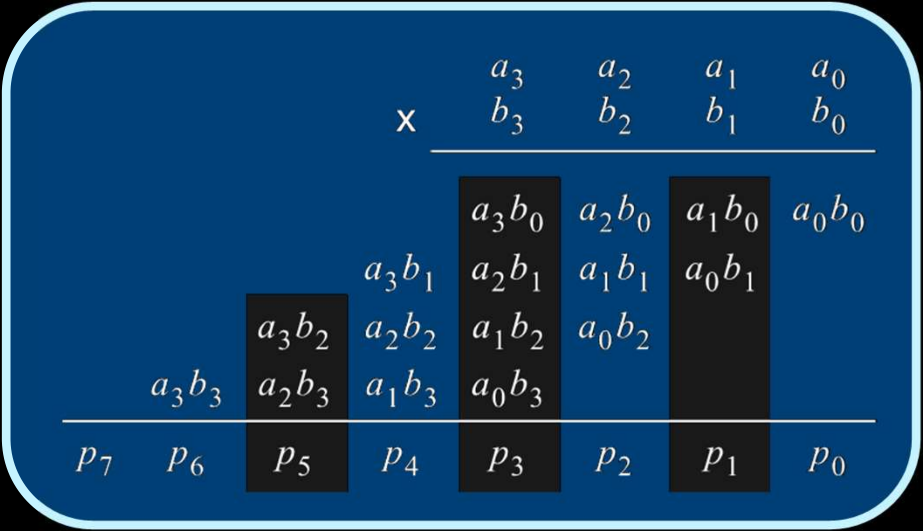
Binary Multiplication



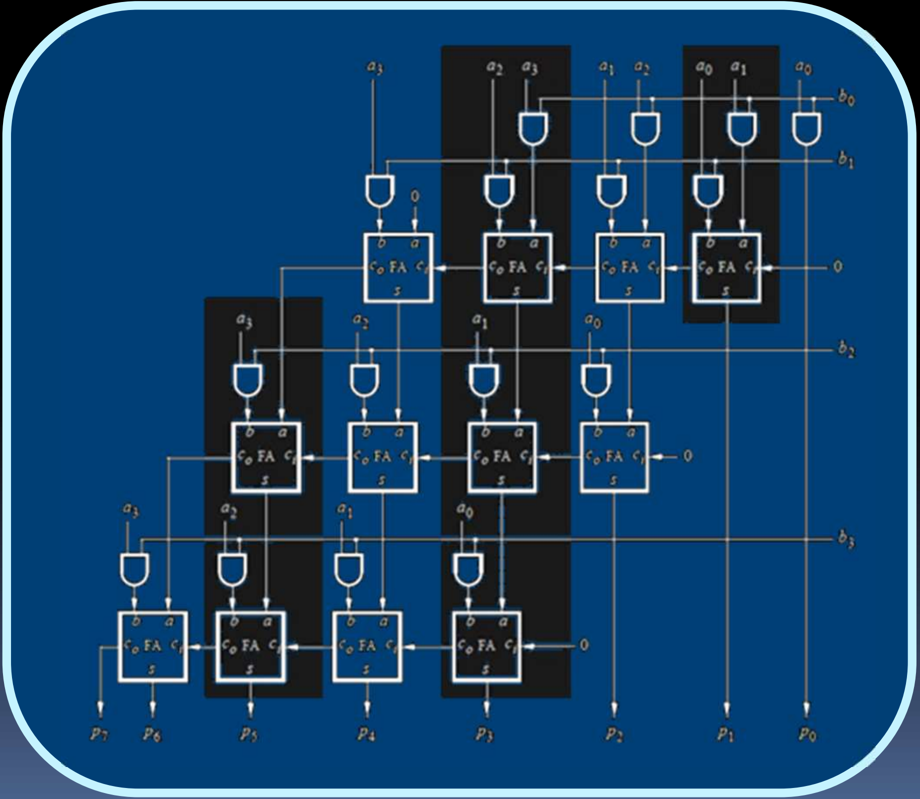
$$
\begin{array}{cccccccc}
 & & & & a_3 & a_2 & a_1 & a_0 \\
 & & & \times & b_3 & b_2 & b_1 & b_0 \\
\cline{5-8}
 & & & & a_3b_0 & a_2b_0 & a_1b_0 & a_0b_0 \\
 & & & a_3b_1 & a_2b_1 & a_1b_1 & a_0b_1 & \\
 & & a_3b_2 & a_2b_2 & a_1b_2 & a_0b_2 & & \\
 & a_3b_3 & a_2b_3 & a_1b_3 & a_0b_3 & & & \\
\hline
p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0
\end{array}
$$

Implementation

- Implementing this in circuitry involves the summation of several AND terms.
  - AND gates combine input signals.
  - Adders combine the outputs of the AND gates.

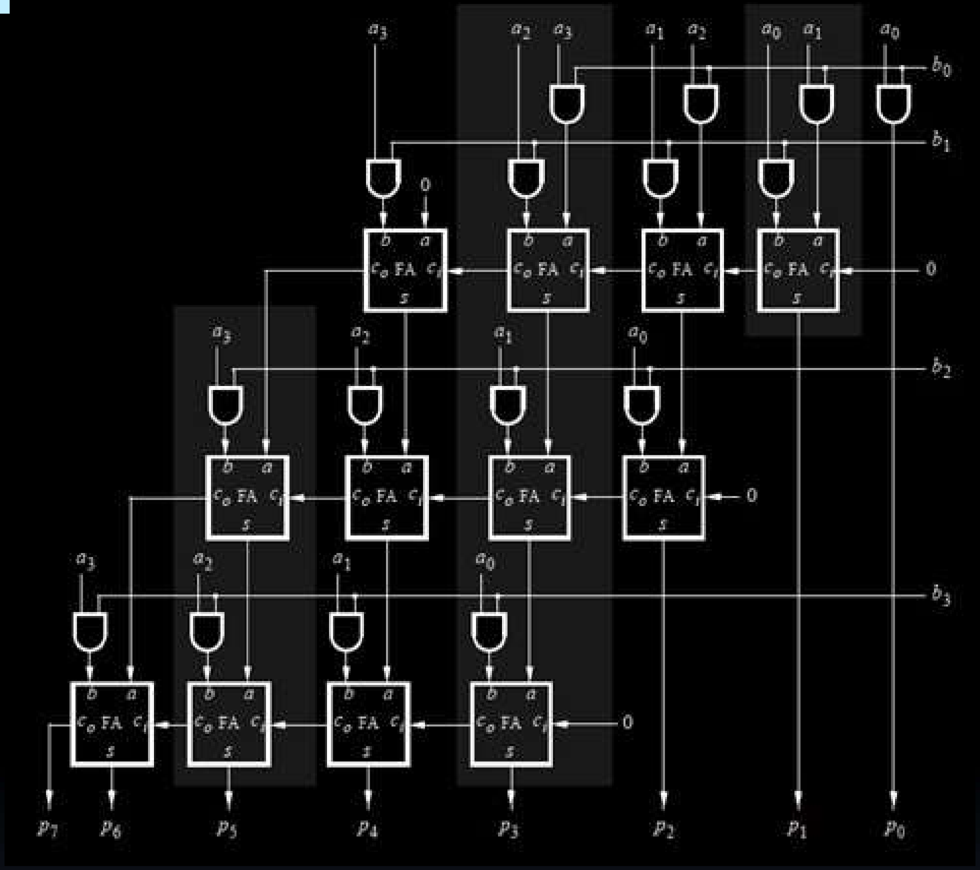


				a_3	a_2	a_1	a_0
			x	b_3	b_2	b_1	b_0
				a_3b_0	a_2b_0	a_1b_0	a_0b_0
			a_3b_1	a_2b_1	a_1b_1	a_0b_1	
		a_3b_2	a_2b_2	a_1b_2	a_0b_2		
	a_3b_3	a_2b_3	a_1b_3	a_0b_3			
p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0



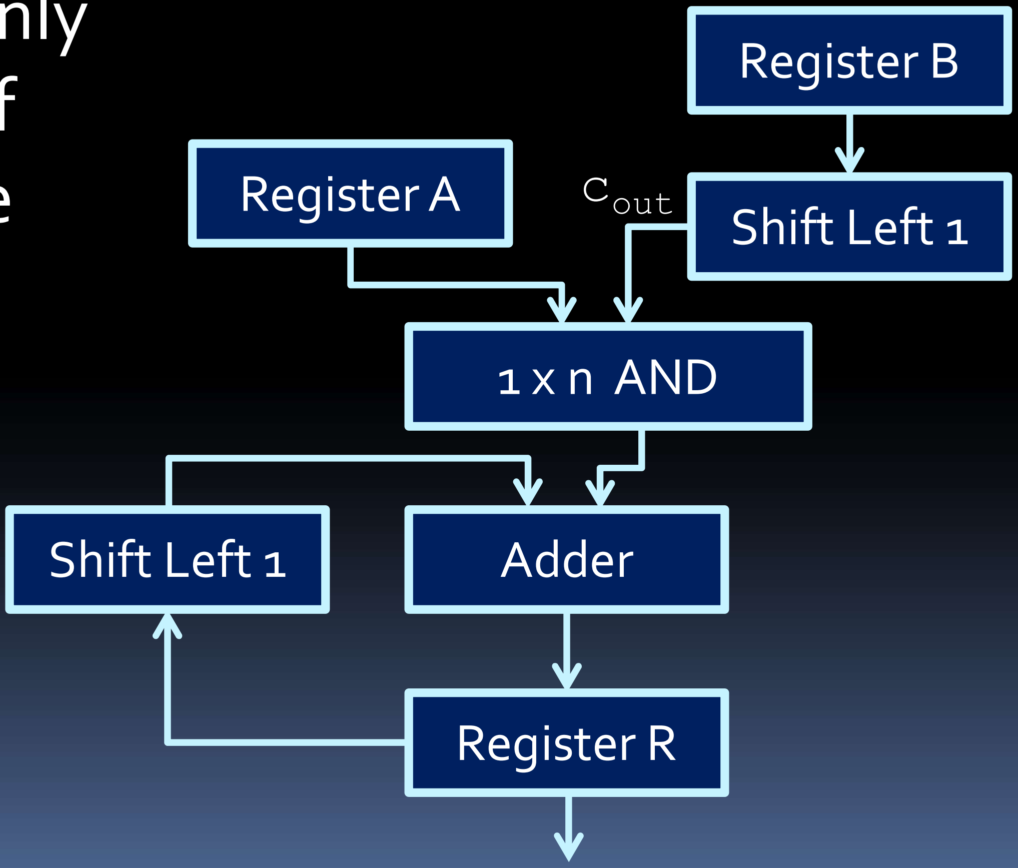
Multiplication

- This implementation results in an array of adder circuits to make the multiplier circuit.
- This can get a little expensive as the size of the operands grows.
    - N-bit numbers → O(1) time, but O(N^2) size.
- Is there an alternative to this circuit?

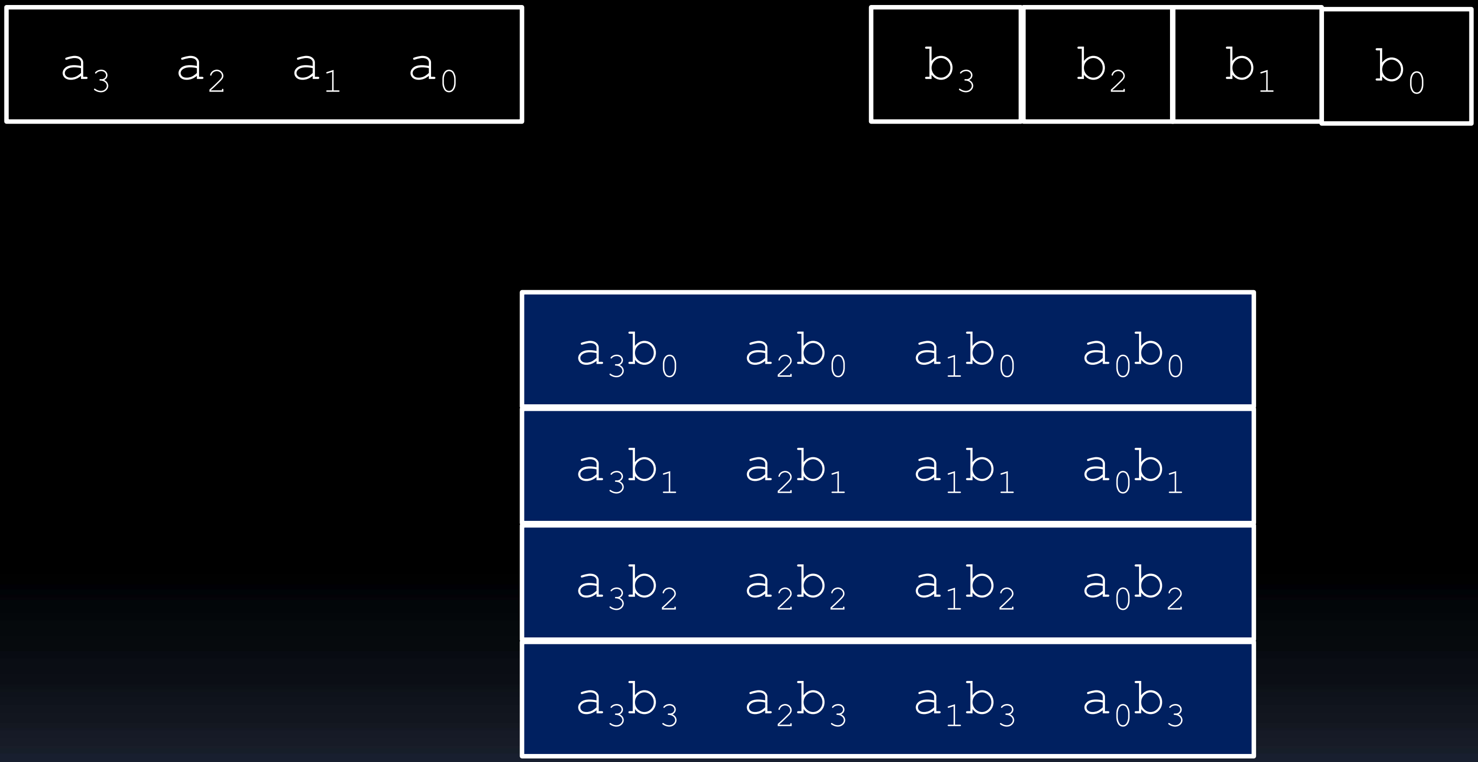


Accumulator circuits

- What if you could perform each stage of the multiplication operation, one after the other?
  - This circuit would only need a single row of adders and a couple of shift registers.
  - How wide does register R have to be?
  - Is there a simpler way to do this?



Accumulator, illustrated

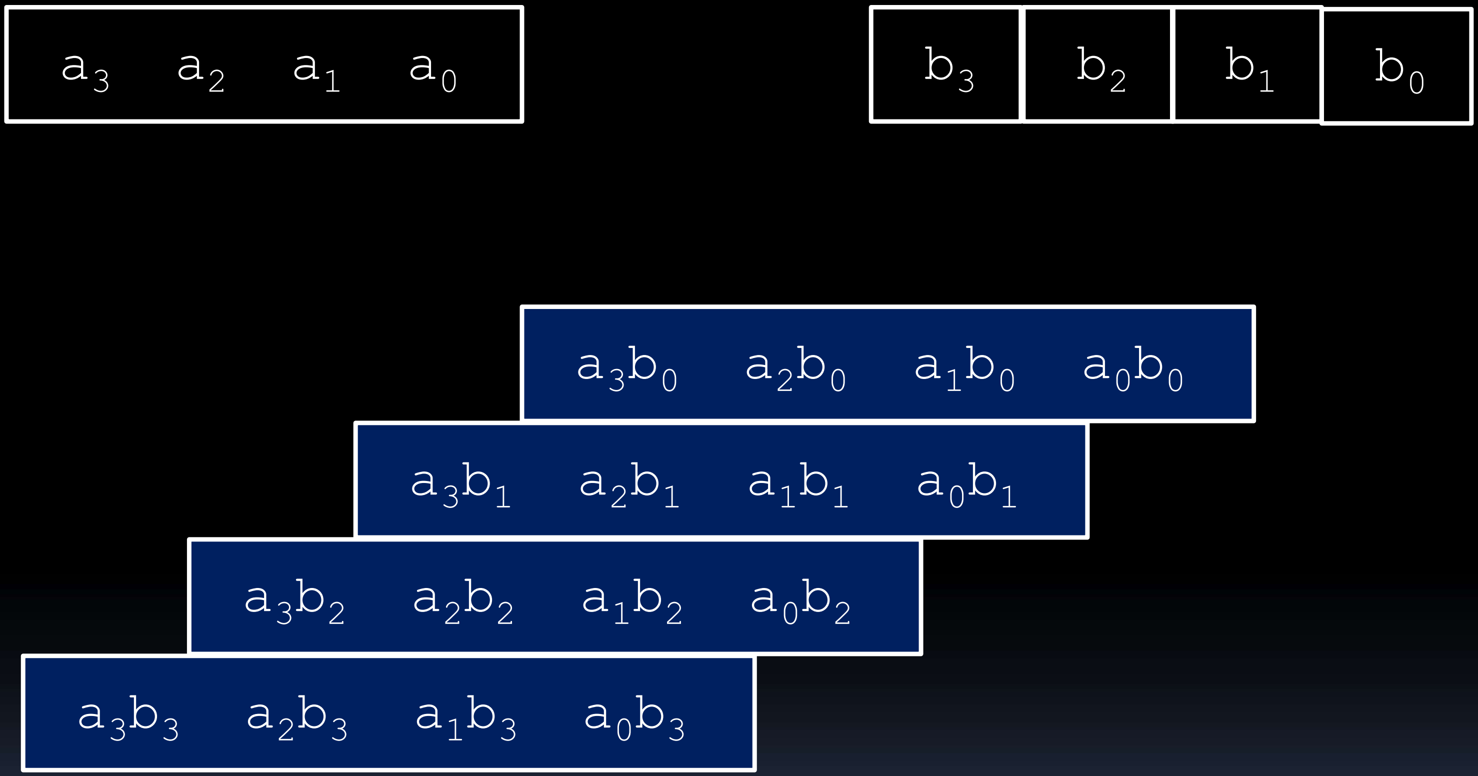


a_3 a_2 a_1 a_0

b_3 b_2 b_1 b_0

a_3b_0	a_2b_0	a_1b_0	a_0b_0
a_3b_1	a_2b_1	a_1b_1	a_0b_1
a_3b_2	a_2b_2	a_1b_2	a_0b_2
a_3b_3	a_2b_3	a_1b_3	a_0b_3

Accumulator, illustrated



$a_3 \quad a_2 \quad a_1 \quad a_0$

$b_3 \quad b_2 \quad b_1 \quad b_0$

$a_3b_0 \quad a_2b_0 \quad a_1b_0 \quad a_0b_0$

$a_3b_1 \quad a_2b_1 \quad a_1b_1 \quad a_0b_1$

$a_3b_2 \quad a_2b_2 \quad a_1b_2 \quad a_0b_2$

$a_3b_3 \quad a_2b_3 \quad a_1b_3 \quad a_0b_3$

- Is there a more efficient way to do this?

Booth's Algorithm

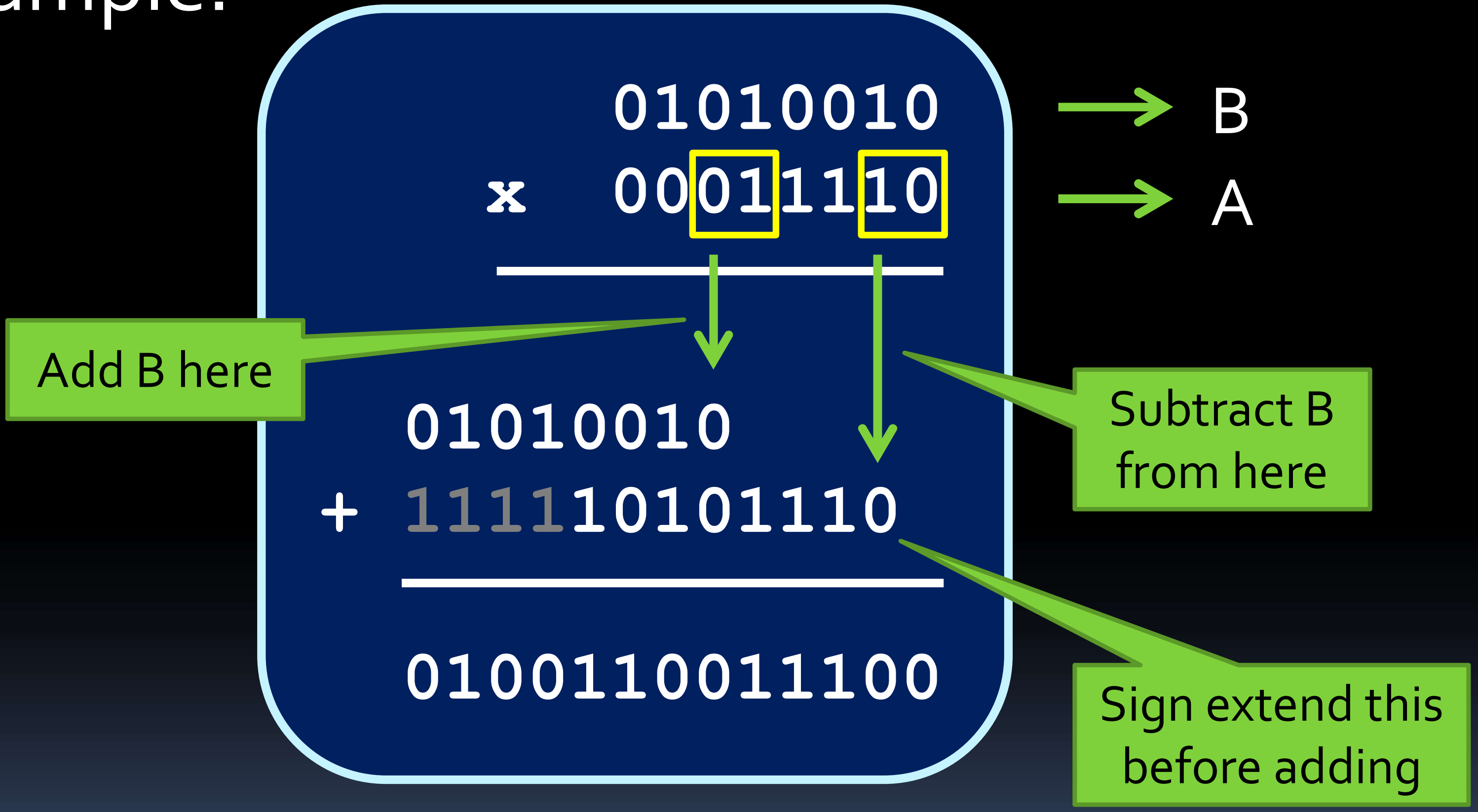
- Devised as a way to take advantage of circuits where shifting is cheaper than adding, or where space is at a premium.
  - Based on the premise that when multiplying by certain values (e.g. 99), it can be easier to think of this operation as a difference between two products.
- Consider the shortcut method when multiplying a given decimal value X by 9999:
  - `X*9999 = X*10000 - X*1`
- Now consider the equivalent problem in binary:
  - `X*001111 = X*010000 - X*1`

Booth's Algorithm

- This idea is triggered on cases where two neighboring digits in an operand are different.
  - If digits at `i` and `i-1` are `0` and `1`, the multiplicand is added to the result at position `i`.
  - If digits at `i` and `i-1` are `1` and `0`, the multiplicand is subtracted from the result at position `i`.
- The result is always a value whose size is the sum of the sizes of the two multiplicands.

Booth’s Algorithm

- Example:



```
        01010010
  x     00011110
  ----------------
    01010010
  + 111110101110
  ----------------
    0100110011100
```

→ B

→ A

Add B here

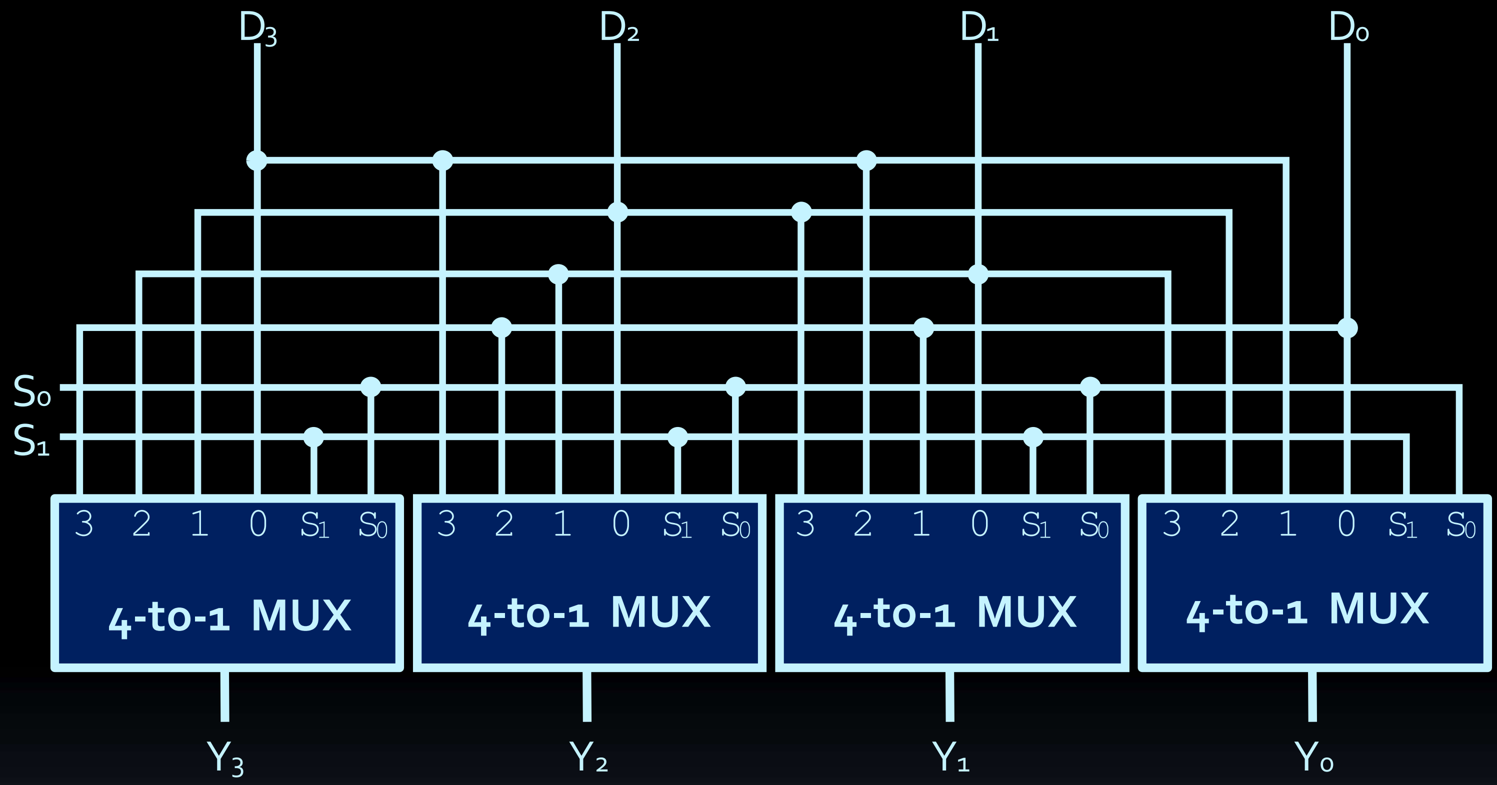
Subtract B from here

Sign extend this before adding

Reflections on multiplication

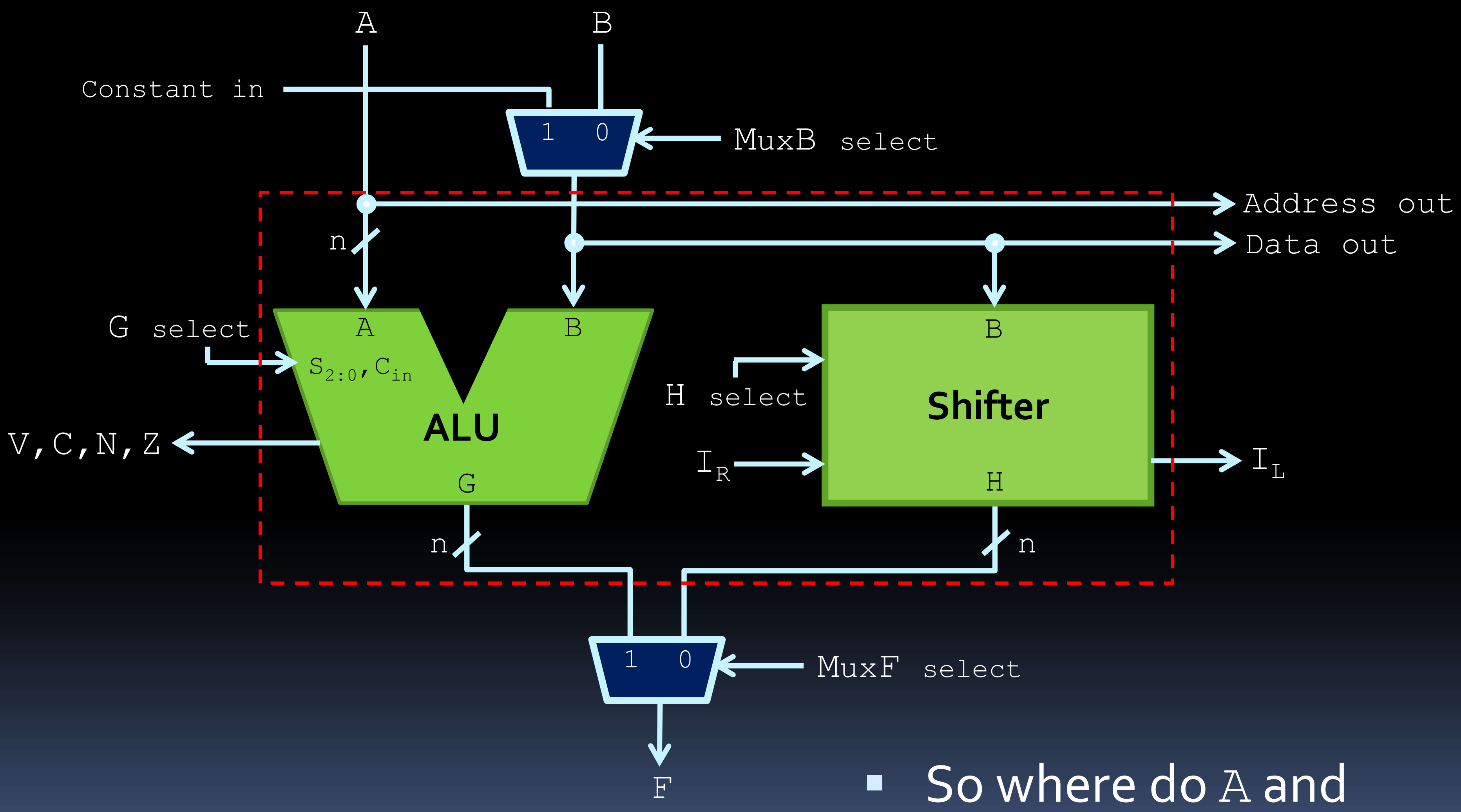
- Multiplication isn't as common an operation as addition or subtraction, but occurs enough that its implementation is handled in the hardware, rather than by the CPU.
- Most common multiplication and division operations are powers of 2. For this, a shift register is used instead of the multiplier circuit.

A Barrel Shifter unit



- This barrel shifter **shifts and rotates** D to the left by S bits.
  - If S_1S_0 is 01 => $Y = D_2D_1D_0D_3$
  - If S_1S_0 is 11 => $Y = D_0D_3D_2D_1$
- This is a purely combinational circuit, unlike the shift registers in the lab.

Expanding our view



- So where do A and B come from?

The Storage Unit

aka: the register file and main memory

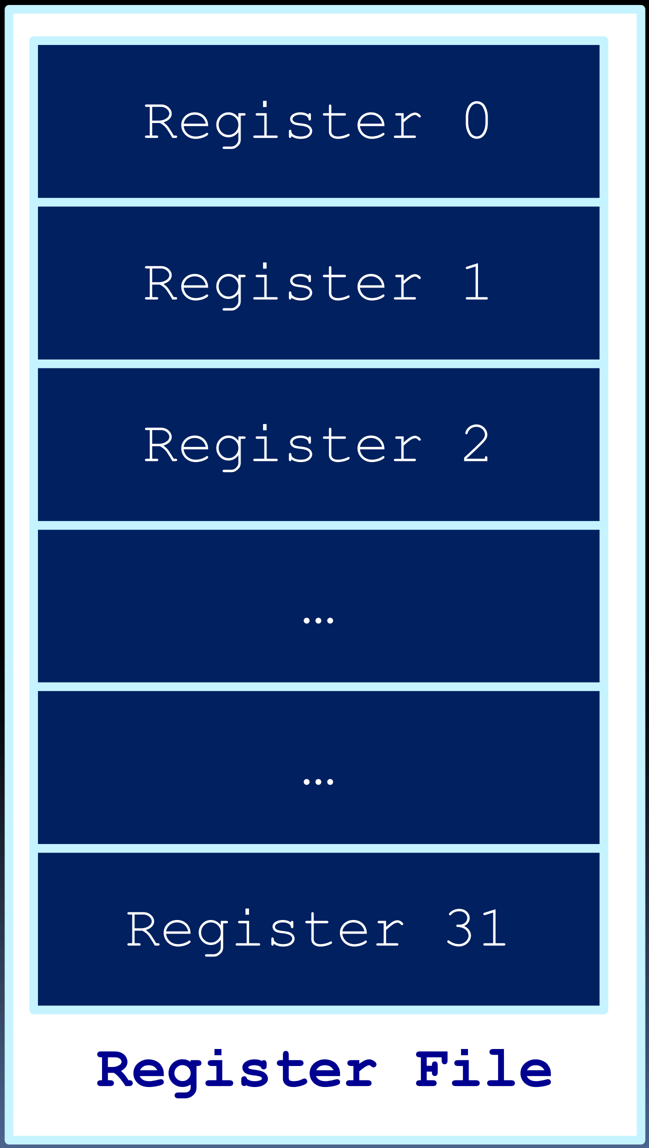
Memory and registers

- The processor has registers that store a single value (program counters, instruction registers, etc.)
- There are also units in the CPU that store large amounts of data for use by the CPU:
    - Register file: Small number of fast memory units that allow multiple values to be read and written simultaneously.
    - Main memory: Larger grid of memory cells that are used to store the main information to be processed by the CPU.

An Analogy

- If registers are books…
- The register file is the pile of books on your desk, small in number but available for quick access.
- Main memory is the library. Larger capacity, but takes time to access.
- Other elements: cache (local library branch), and networks (collections around the world)

The Register File



Register 0

Register 1

Register 2

…

…

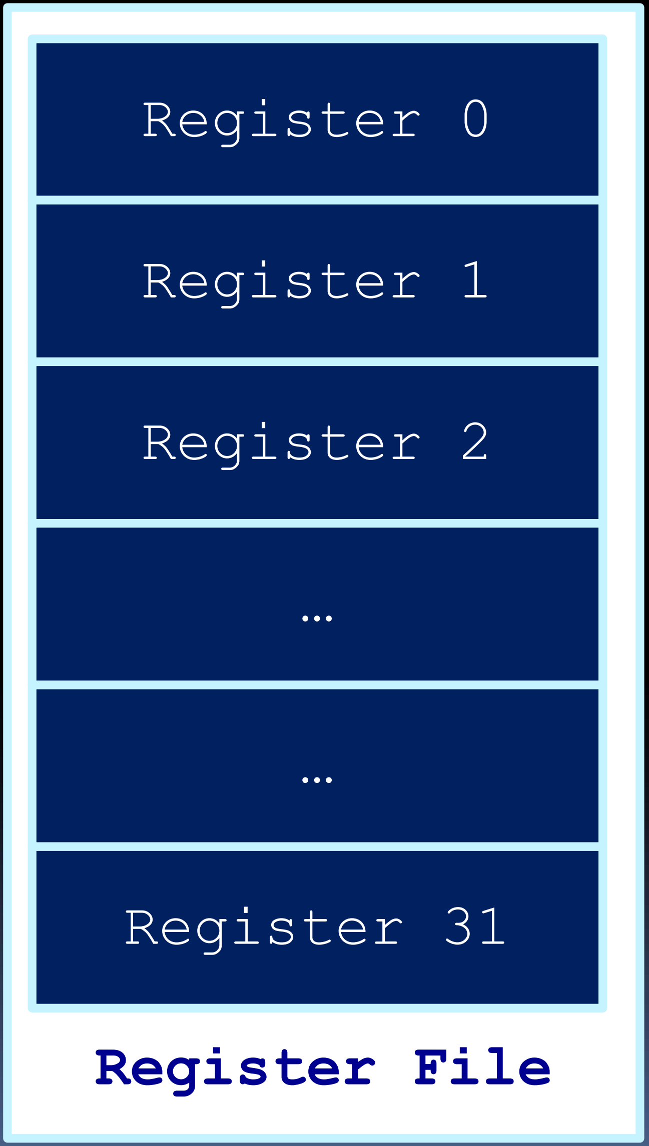
Register 31

Register File

- The register file stores the data that the processor uses for quick calculations.
  - For the example we use in this course, the typical register file contains 32 registers.
- All the registers share a single set of input and output wires.
  - Like an apartment building with a single shared entrance, as opposed to a street of houses.
- So how do we specify a register to read or write?

Writing to the registers

- To write a register, we need to activate the Write signal for the desired register.
  - Assume we have the number (from `0` to `31`, in binary) of the register in this file.
  - This number is also called the address of the register.
- Given the register's address, how do we turn on the write signal for that register?



Register 0

Register 1

Register 2

…

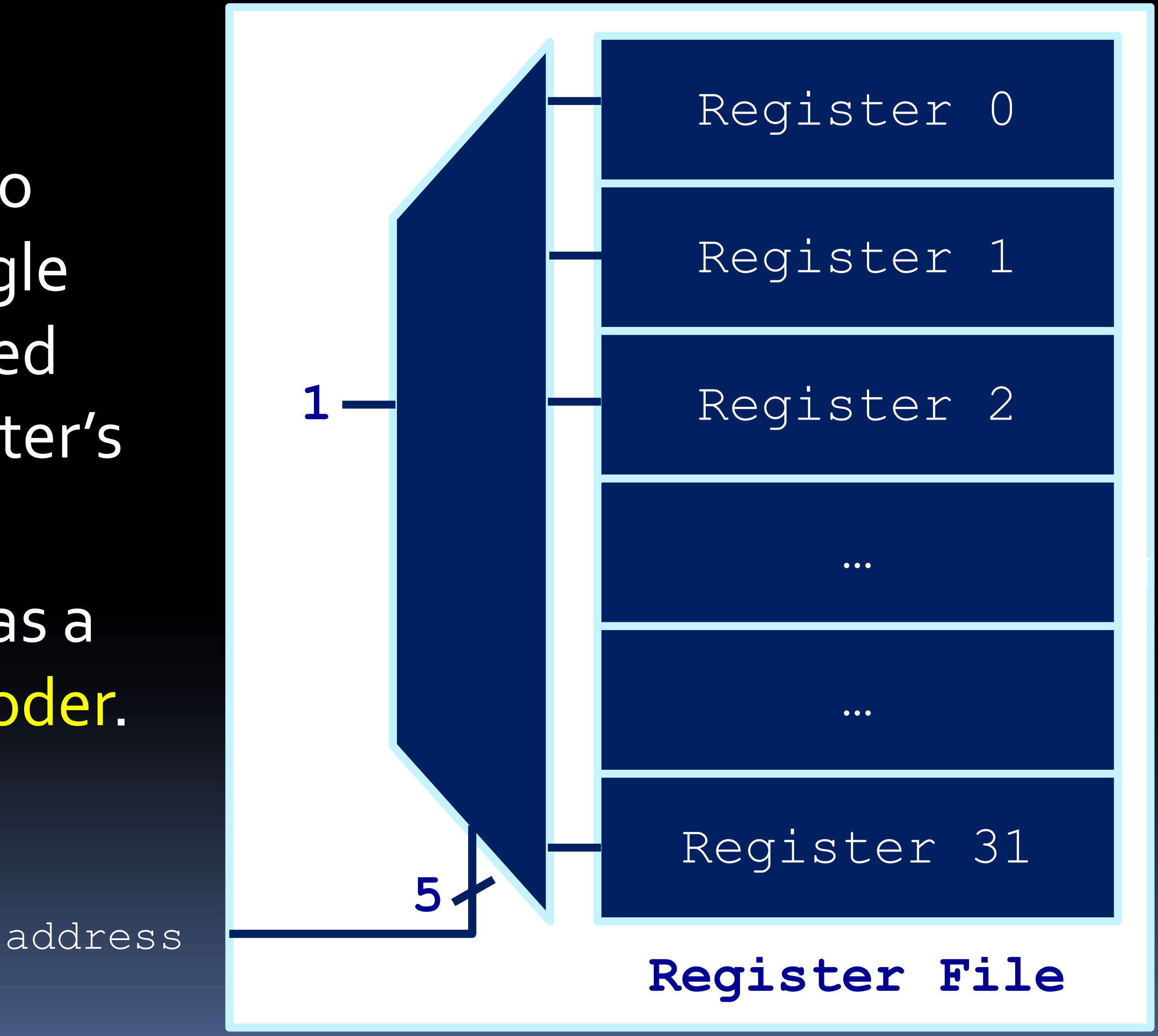
…

Register 31

Register File

Writing to registers

- Solution:
  - Use demux to specify a single register, based on that register's address.
  - Also known as a one-hot decoder.



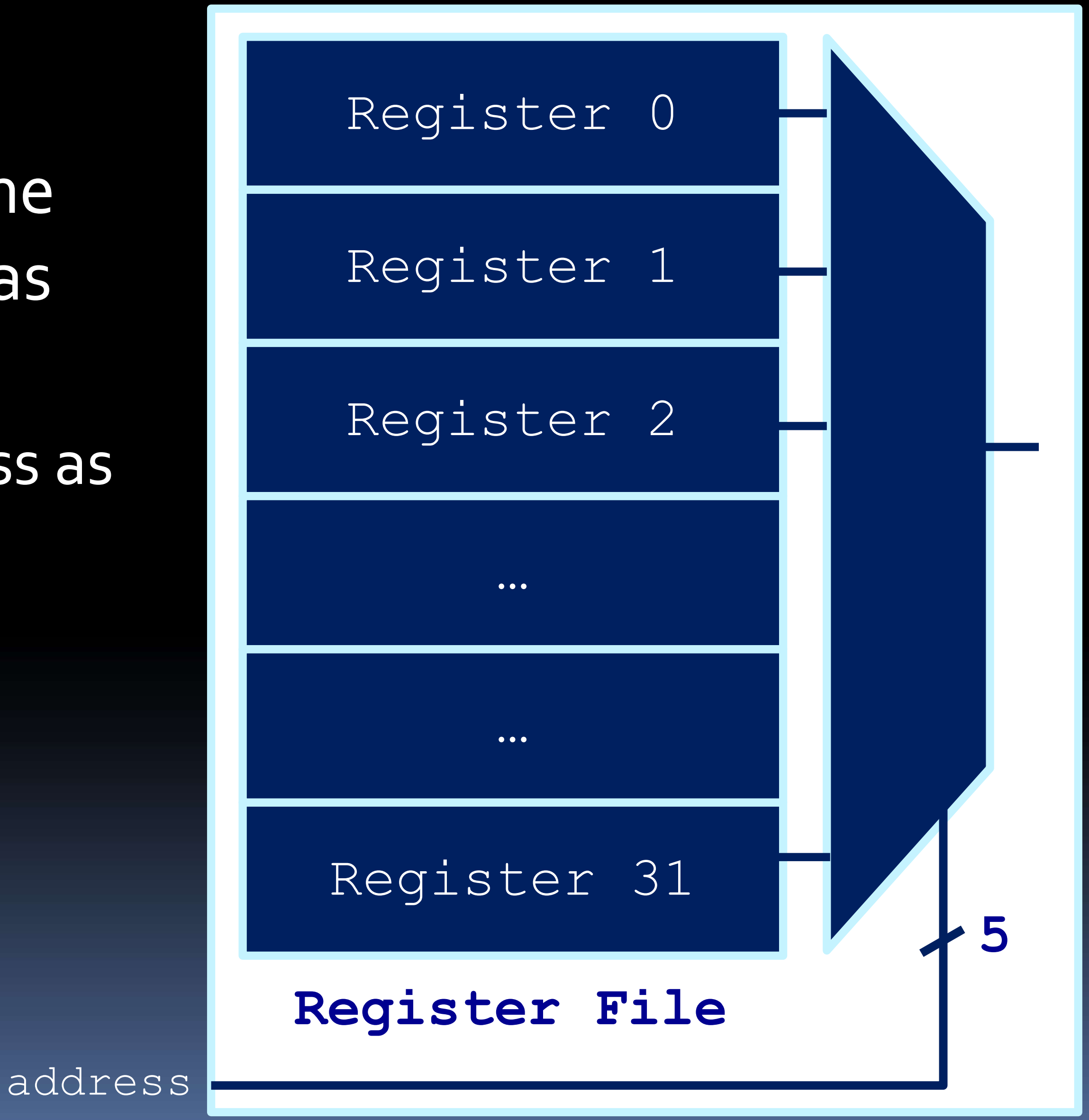
One-hot decoders (for writing)

- The one-hot decoder takes in a m-bit binary address to activate one of the 2^m registers in the register file.

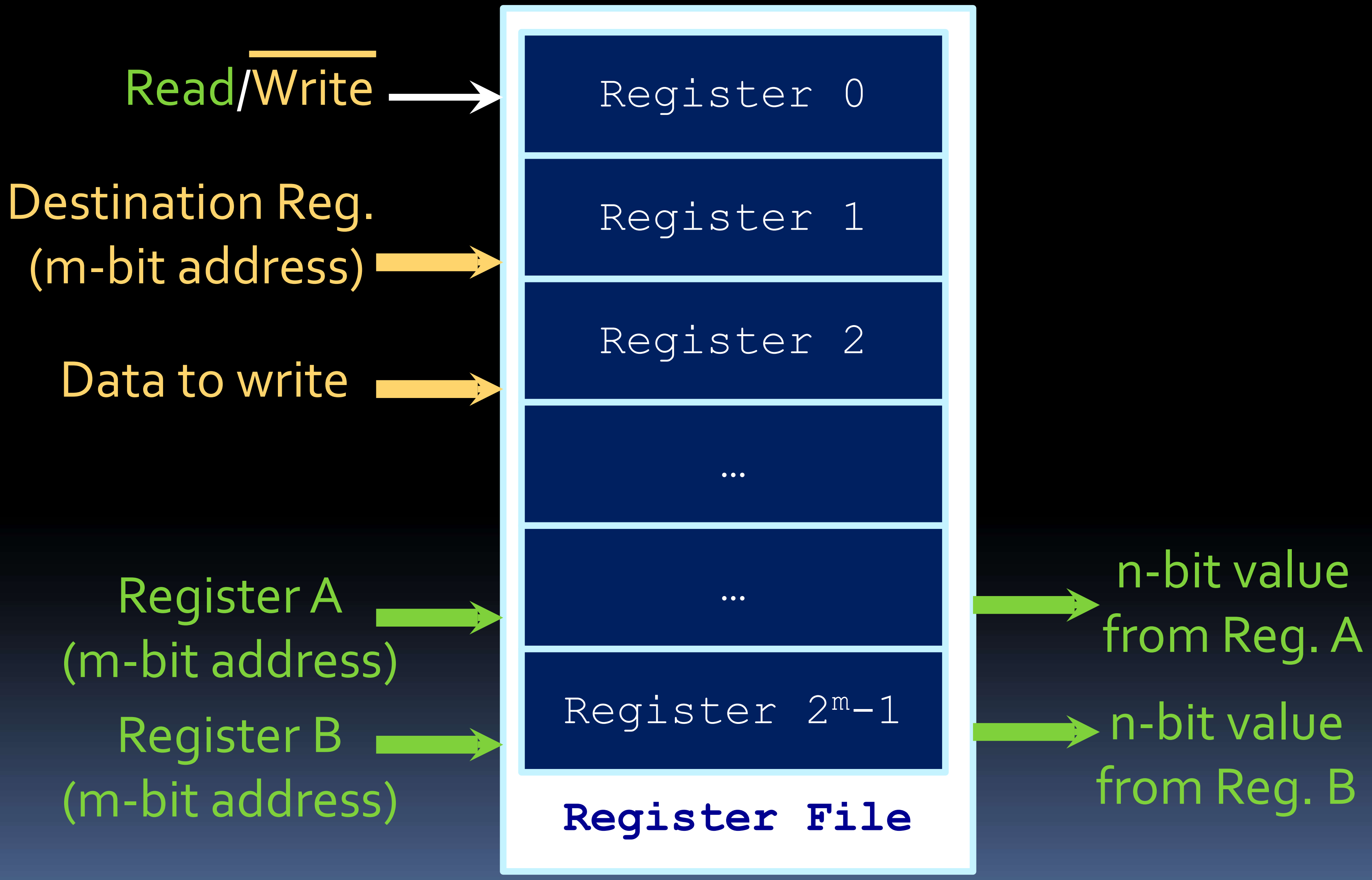
Address bits			One-hot decoder output							
A_2	A_1	A_0	O_7	O_6	O_5	O_4	O_3	O_2	O_1	O_0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
	…					…				
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

Reading from registers

- Solution:
  - Consider the same approach that was used in writing.
    - Mux with address as select bits.



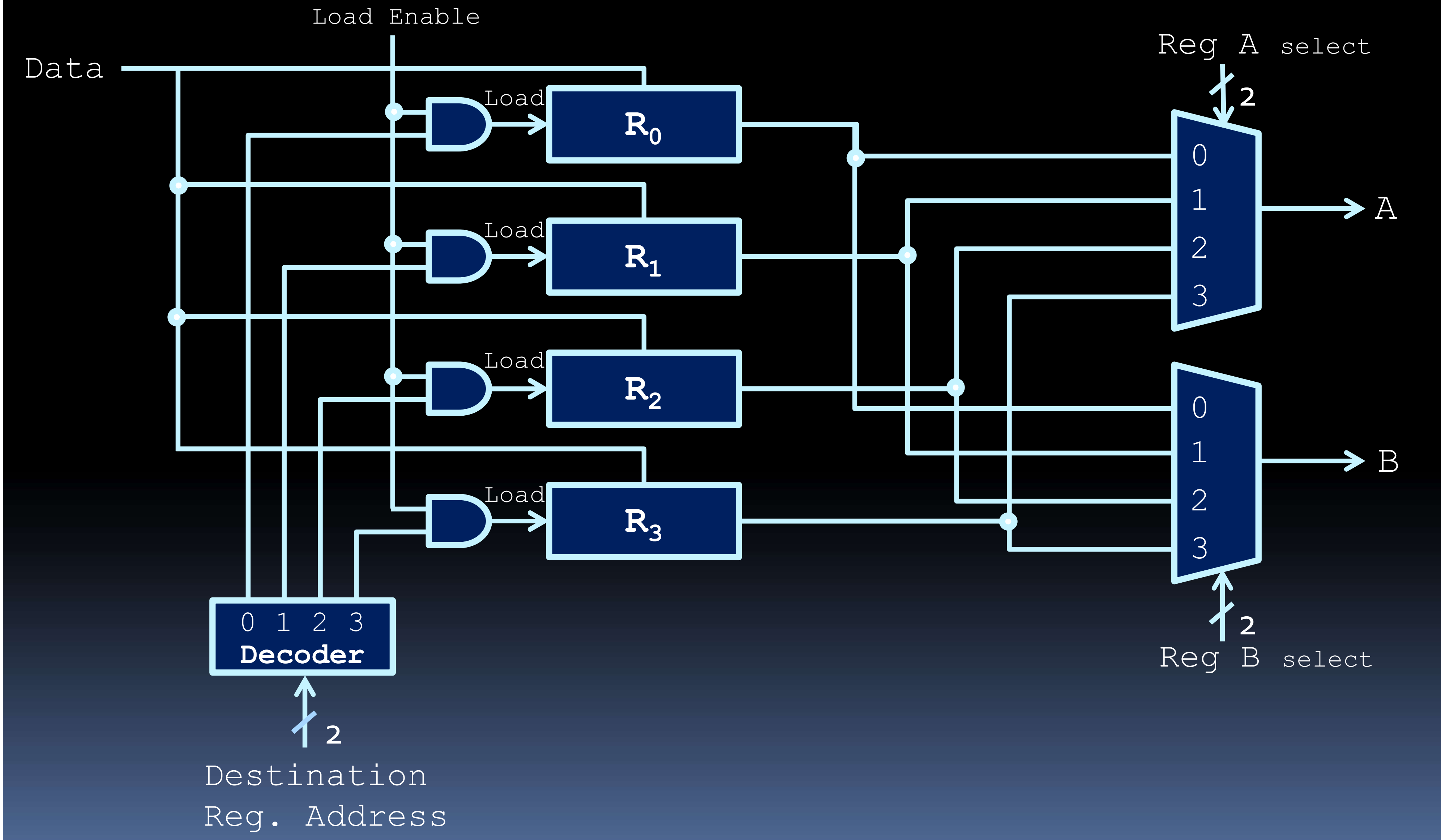
Register File Functionality



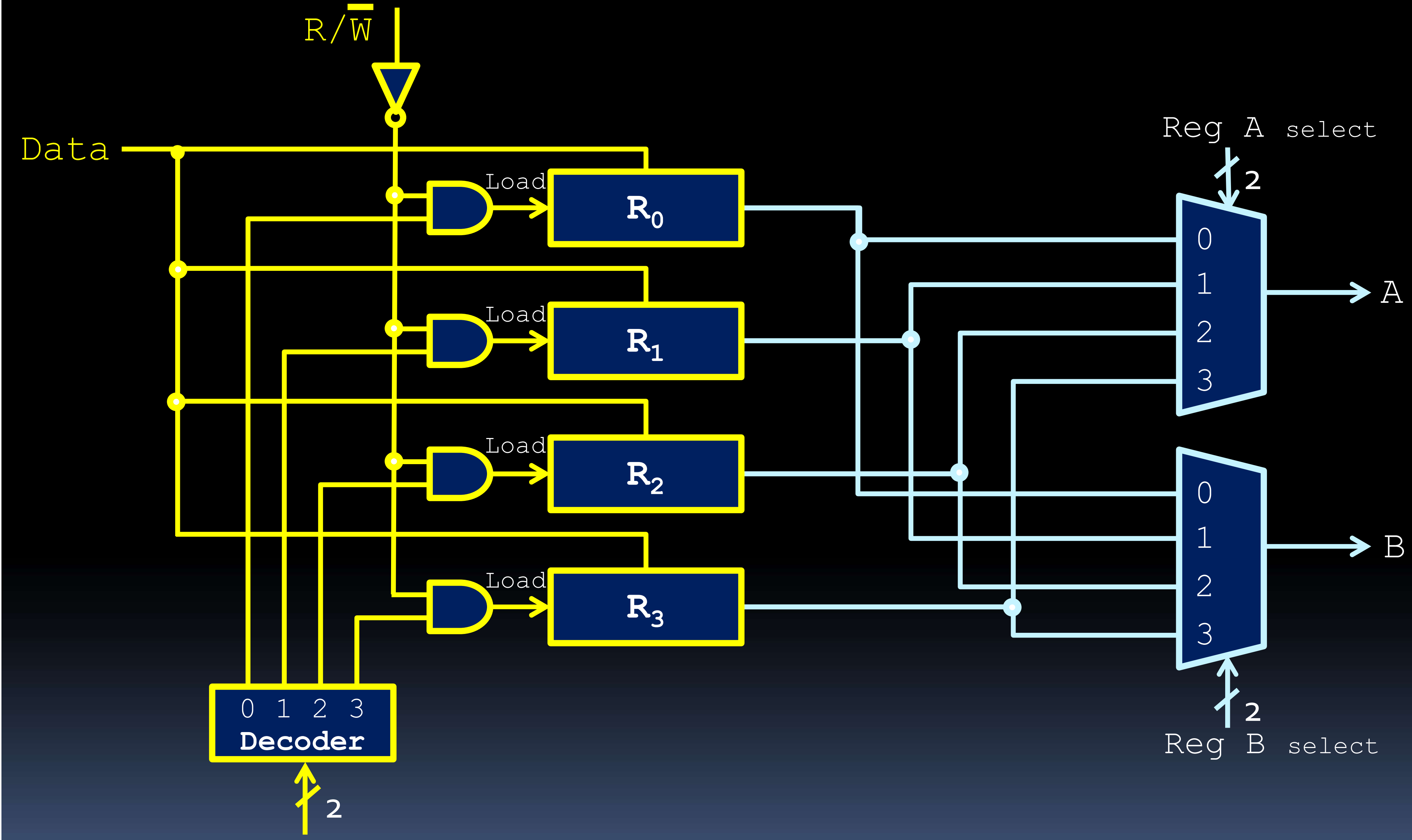
Handling multiple registers

- The register unit in our microprocessor stores 32 registers (each register storing a 32 bit value).
- How do we access or update a single register?
  - Need to specify ALU input A and ALU input B when performing a read operation.
  - Need to specify a register to write to (and the data value to write) when performing a write operation.
- Both of these are done by specifying the address of each register among the 32 available.
  - Each address will be 5 bits ($\log_2 32$).

Register File – Write Operation

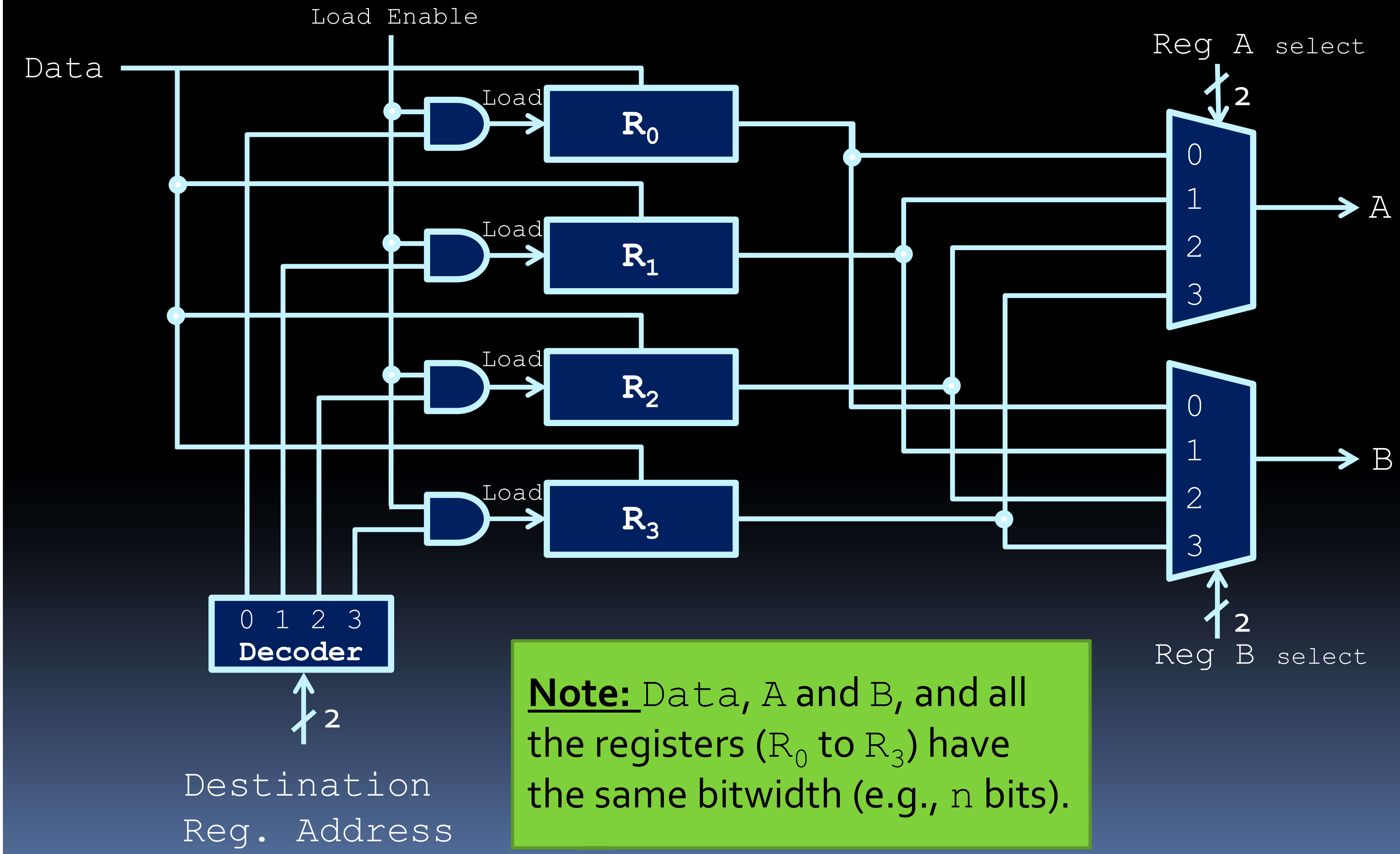


Register File – Write Operation



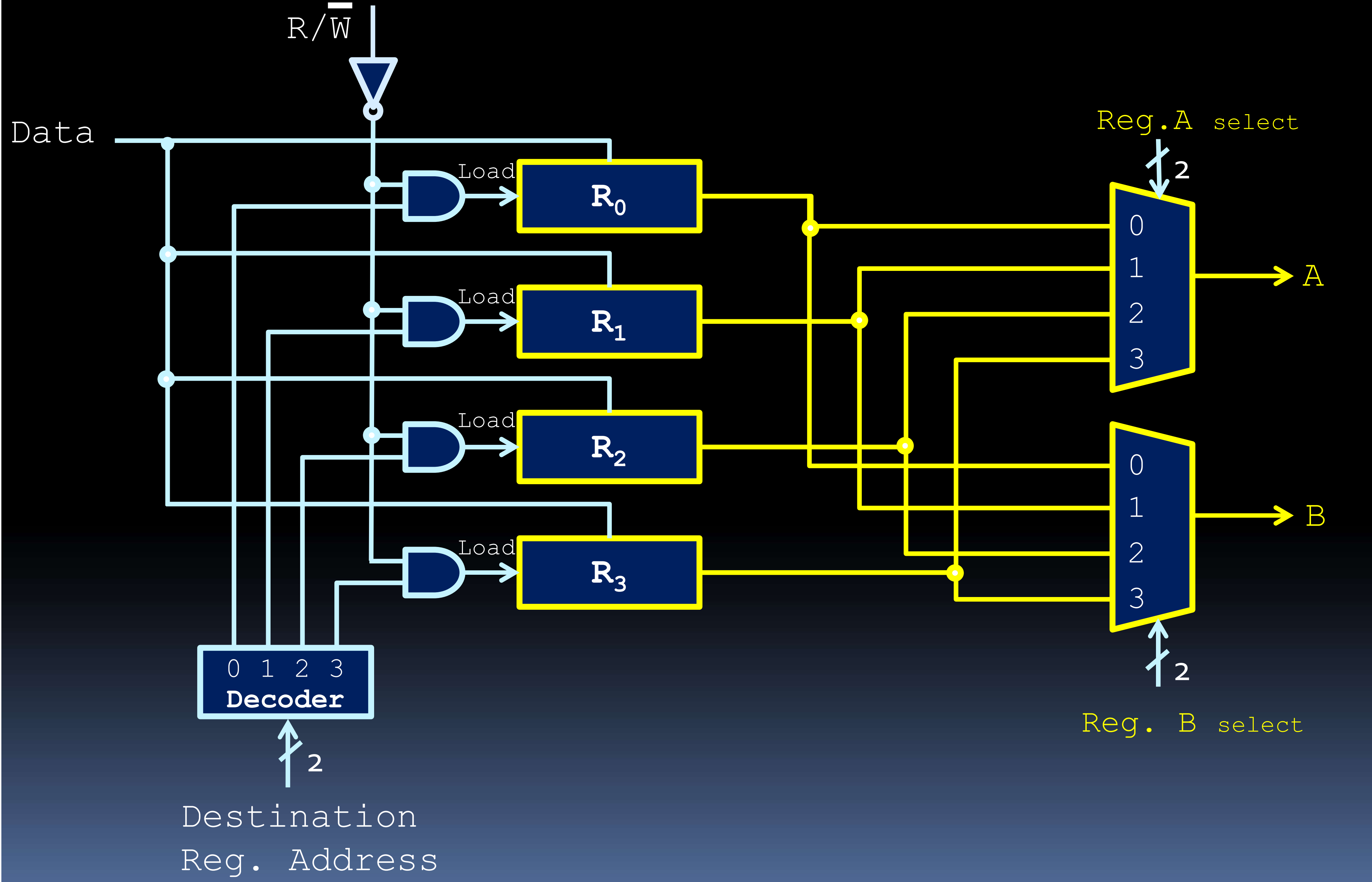
Destination
Reg. Address

Register File – Read Operation



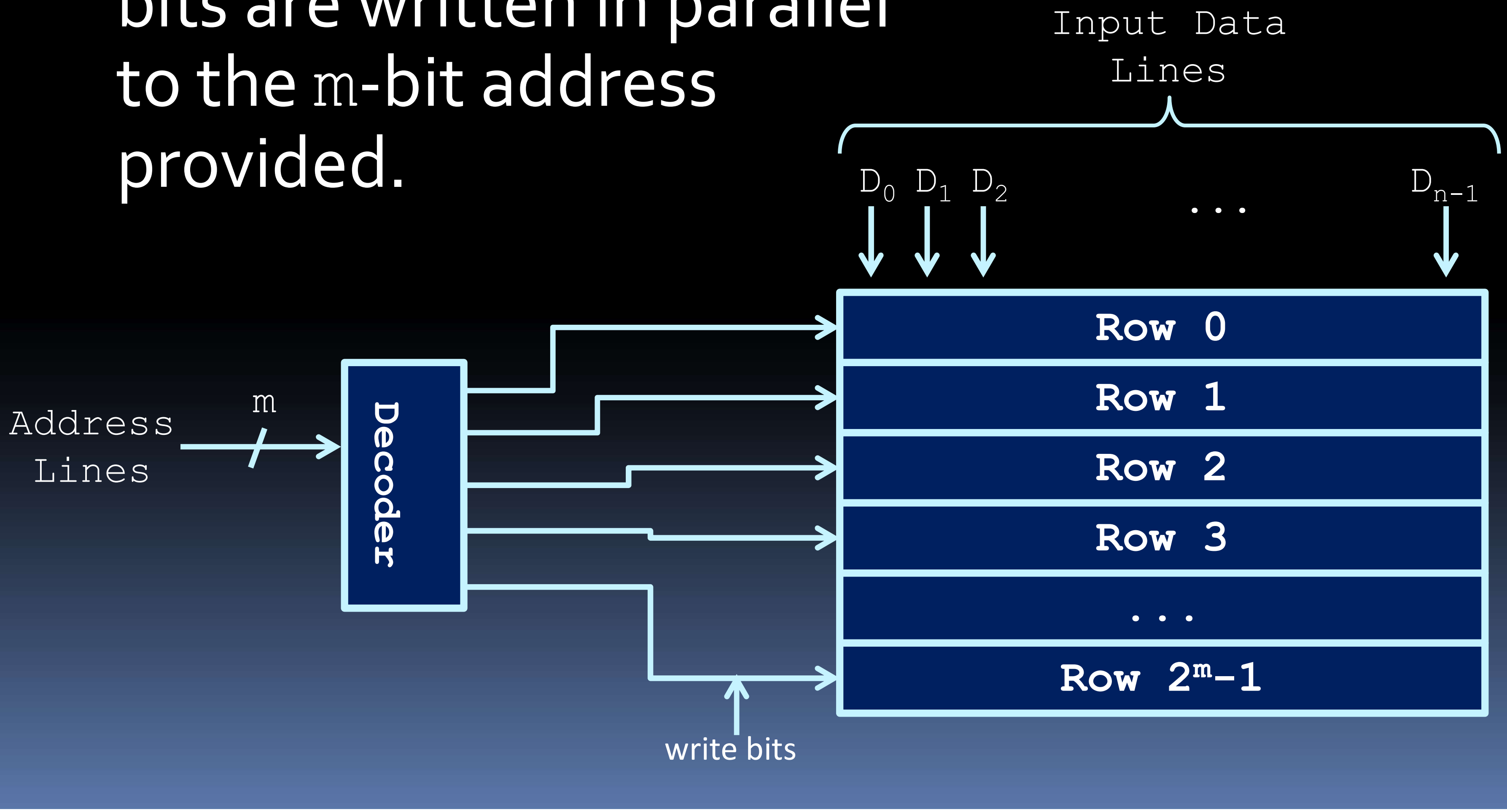
Note: Data, A and B, and all the registers (R_0 to R_3) have the same bitwidth (e.g., n bits).

Register File – Read Operation



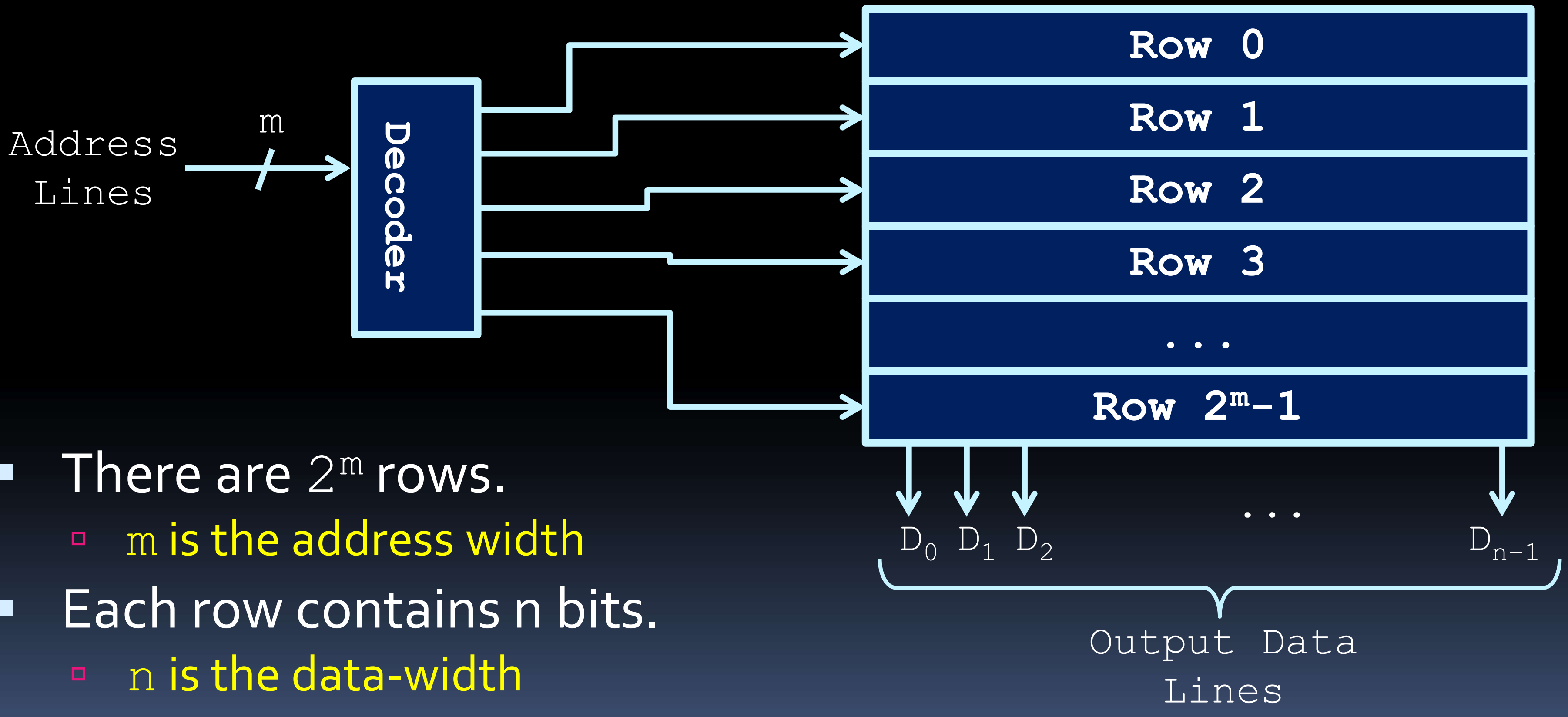
Electronic Memory (RAM)

- Like register files, main memory is made up of a decoder and rows of memory units.
- When the Read/$\overline{\text{Write}}$ signal is low, n data bits are written in parallel to the m-bit address provided.



Electronic Memory (RAM)

- When Read/$\overline{\text{Write}}$ signal is high, n data bits are read from memory at the m-bit address.



Address Lines

m

Decoder

Row 0

Row 1

Row 2

Row 3

. . .

Row 2^m-1

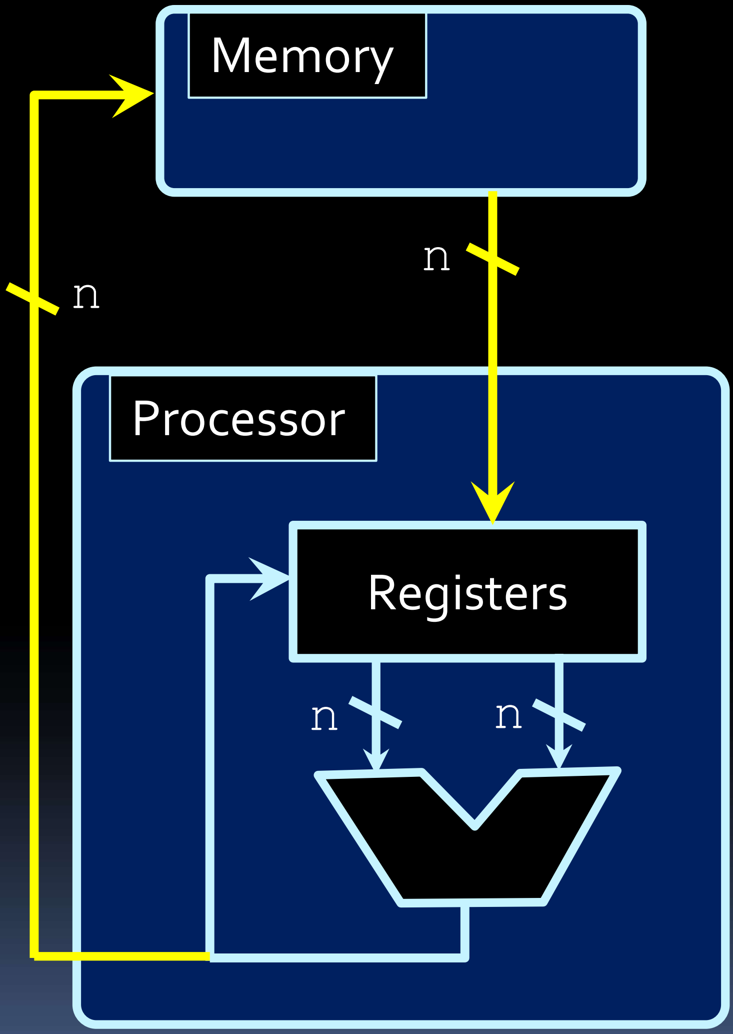
D_0 D_1 D_2 . . . D_{n-1}

Output Data Lines

- There are 2^m rows.
  - m is the address width
- Each row contains n bits.
  - n is the data-width
- What's the size of this memory?
  - 2^m * n bits => 2^m * n / 8 bytes

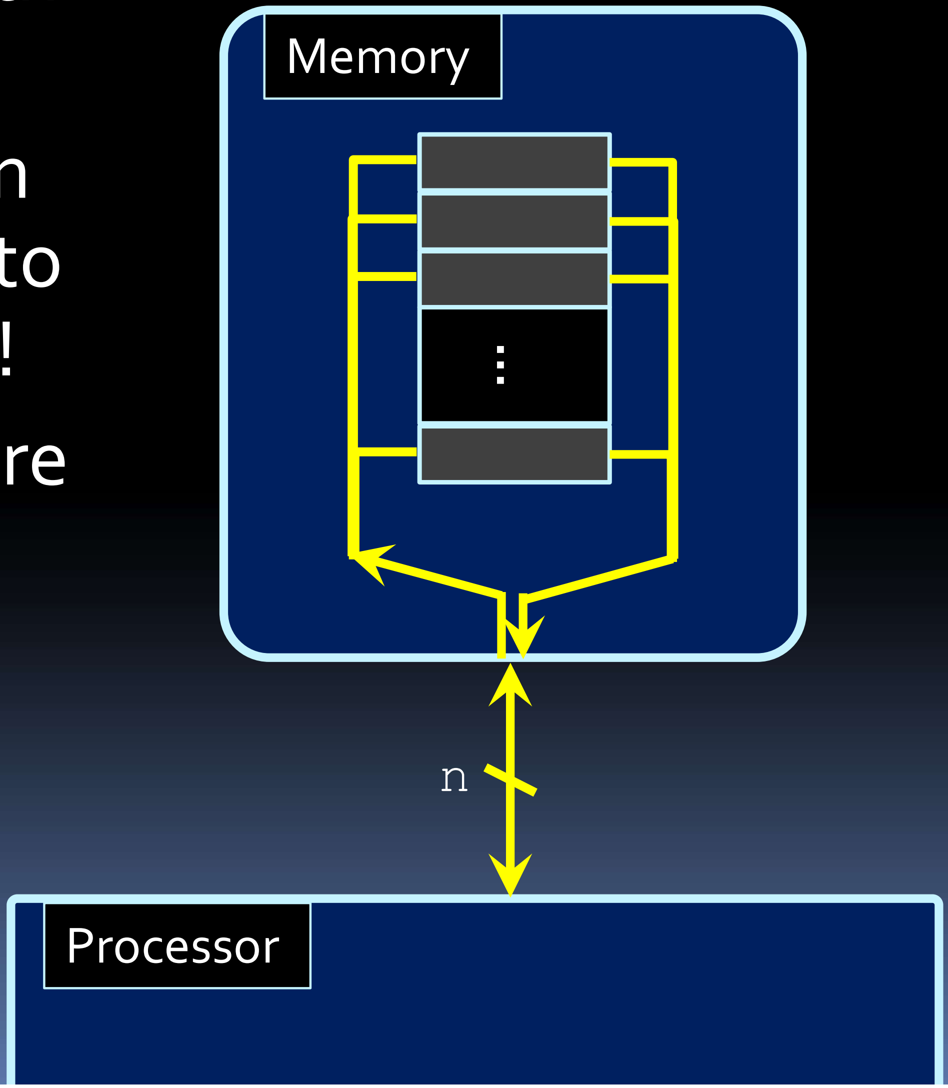
Connecting to memory units

- Memory values are read to the registers and then processed by the ALU.
- Results are eventually sent back to memory.
- Might make you picture an architecture like the one on the right…but….



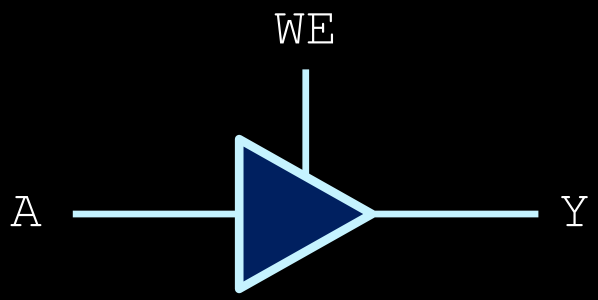
Connecting to memory (cont’d)

- Memory units use the ***same n-bit wires*** to both send and receive data.
- But conflicts arise when multiple sources write to wires at the same time!
- We need a way to ensure that the memory unit doesn’t write to these common wires at the same time that the processor does.



Controlling the flow

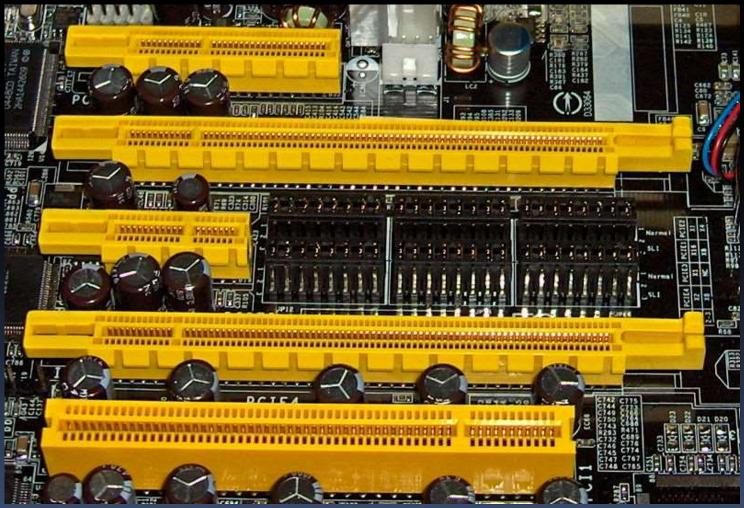
- When reading or writing a memory location, we can use a (sort of) new gate called the tri-state buffer.
  - This sets the output to the input, but only when a third signal (write enable) is high.
- When the WE (write enable) signal is low, the buffer output is a "high impedance" signal.
  - The output is neither connected to high voltage or to the ground (i.e "Z").



WE	A	Y
0	x	z
1	0	0
1	1	1

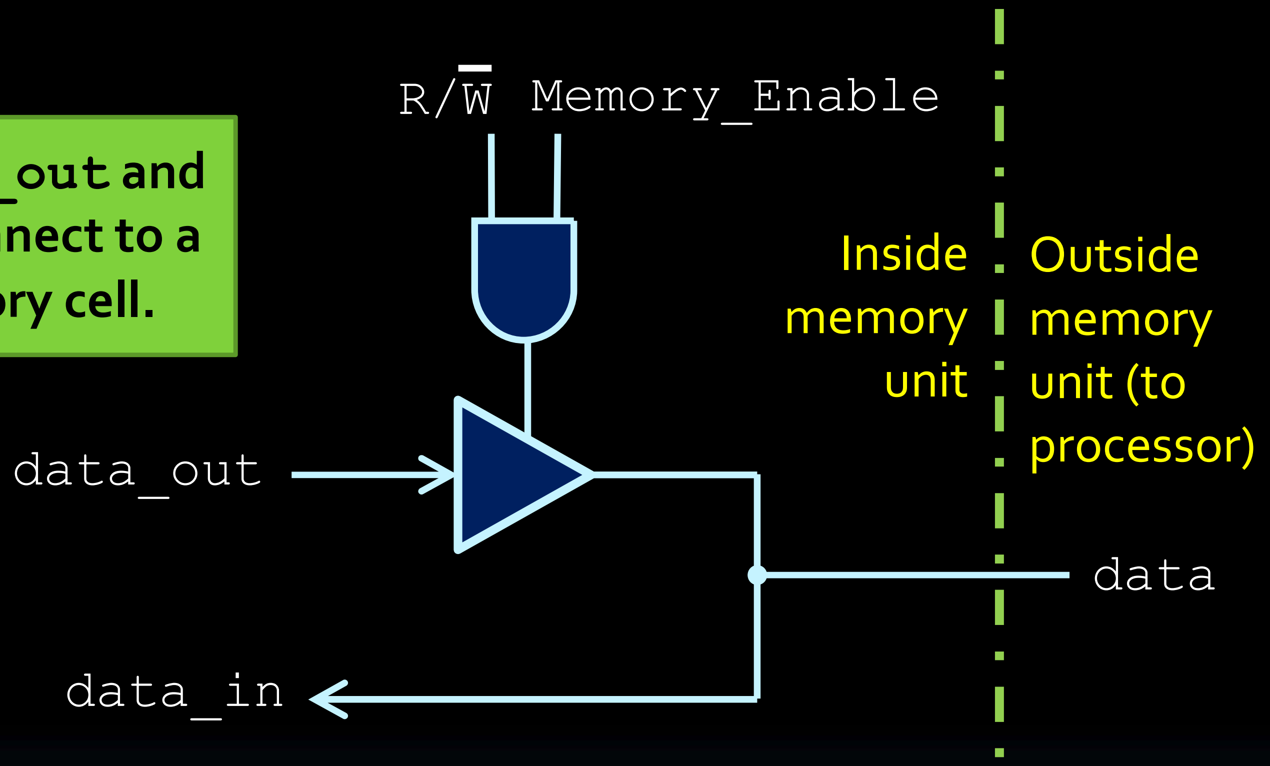
Data Bus

- Tri-state buffers allow us to use a single common of wires called a bus (or data bus) to communicate in both directions between memory and the processor.
- Anything that writes to the bus goes through a tristate buffer. For example, the buffer for a memory location has high impedance when:
  1. The processor is writing to memory.
  2. That memory location is not being accessed.
- When reading from memory, only one location can write to a bus at a time (also called the bus driver). The other memory locations must have their tristate buffers turned off.



Tri-state Buffer Use

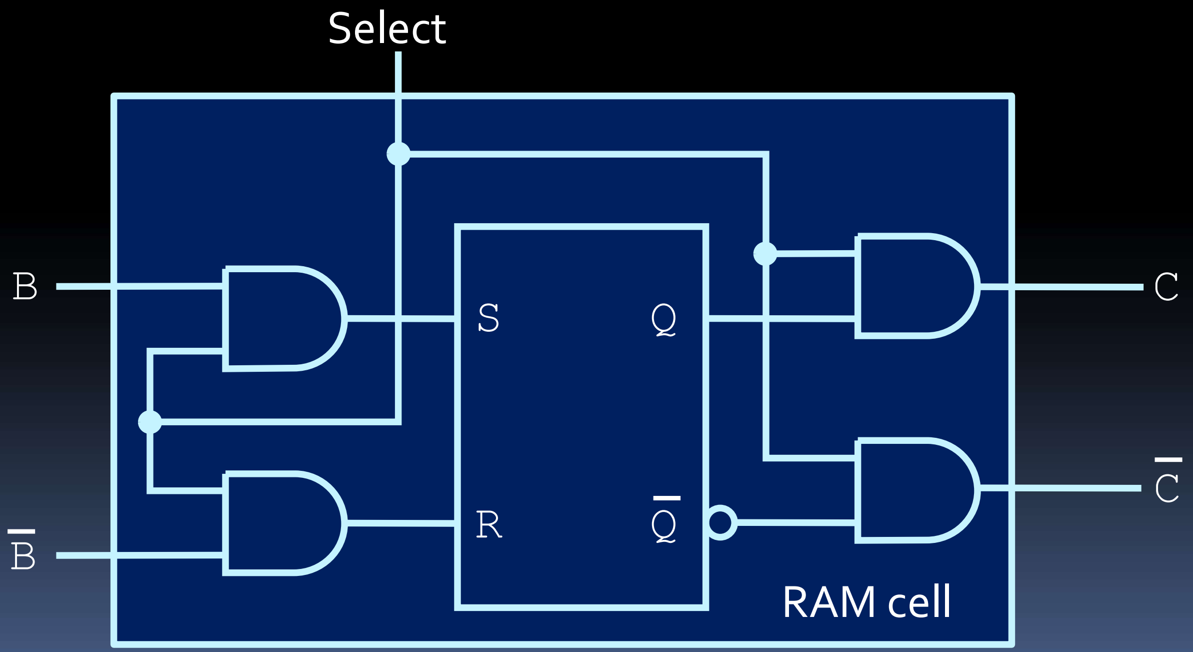
Assume data_out and data_in connect to a single memory cell.

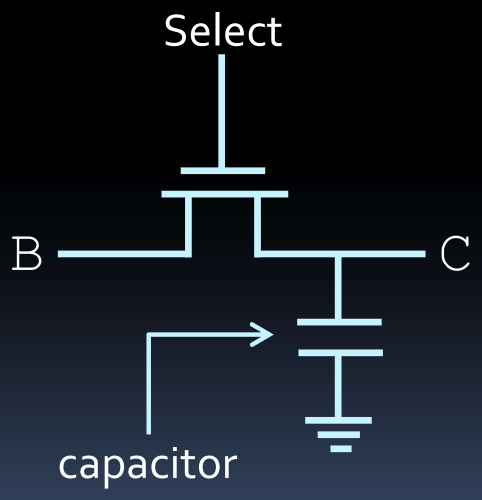


- If $R/\overline{W}$ is 1 (read) & Memory_Enable (for this location) is 1
    - data sends data from memory to processor, through the buffer.
- Otherwise if $R/\overline{W}$ is 0
    - data brings data from processor to memory since the tri-state buffer is disabled (i.e. outputting high impedance).

Storage cells

- Each memory location consists of a row of `n` storage cells (typically a byte = 8 bits).
    - Each cell stores a single bit of information (see below).
- Multiple ways of representing these cells.
    - e.g. RAM cell (know this): DRAM IC cell (just FYI):

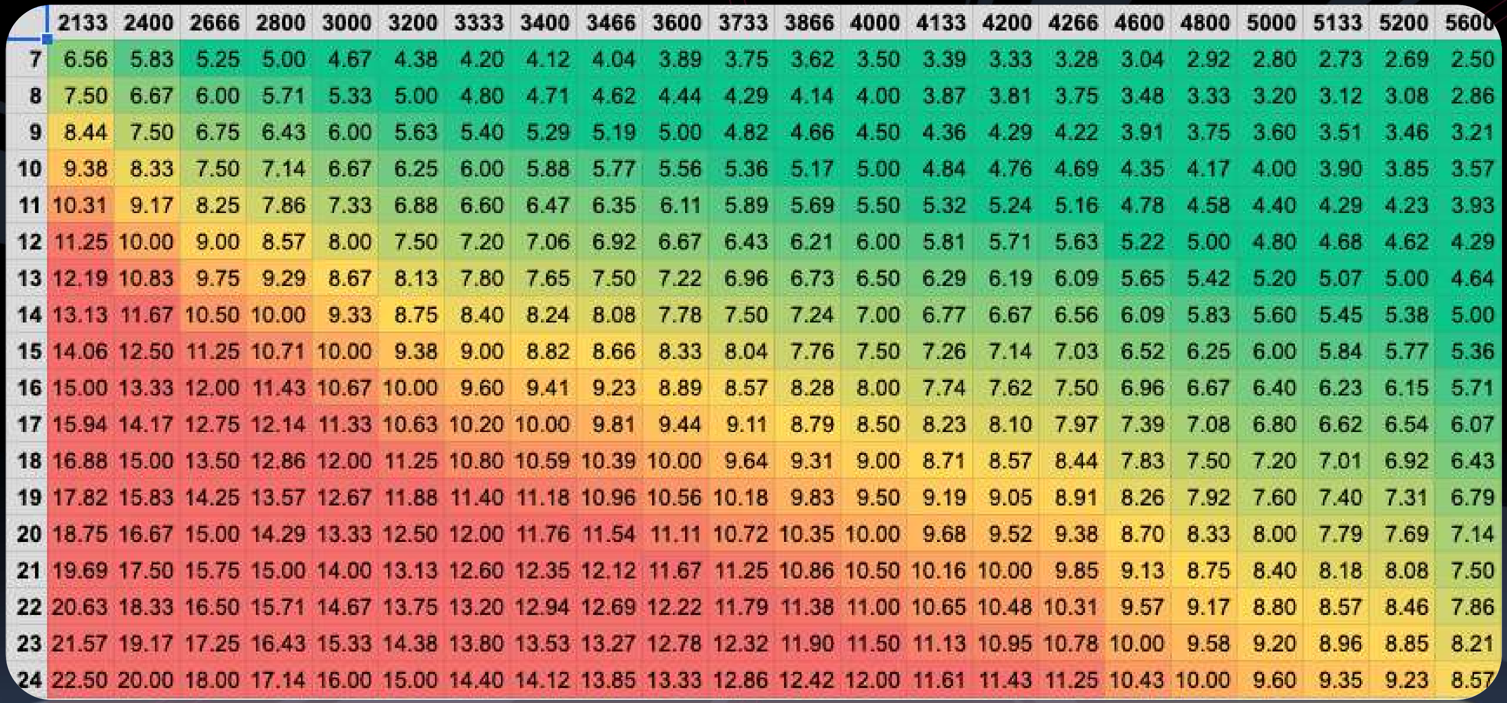




Memory vs registers

- Memory and registers are similar in principle.
  - Both store data values, both use addresses to specify which value to access.
- They are different in two major ways:
  - Usage:
    - Memory is MUCH bigger, and houses most of the long-term data values being used by a program.
    - Registers are local data values, used internally by the processor to perform an operation.
      - Like scrap paper for a calculation, discarded when the calculation is complete (with some notable exceptions).
  - Access:
    - Register access is immediate. Memory units are separate from the main processor, requiring more time for a single access.

Memory Timing Issues



	2133	2400	2666	2800	3000	3200	3333	3400	3466	3600	3733	3866	4000	4133	4200	4266	4600	4800	5000	5133	5200	5600
7	6.56	5.83	5.25	5.00	4.67	4.38	4.20	4.12	4.04	3.89	3.75	3.62	3.50	3.39	3.33	3.28	3.04	2.92	2.80	2.73	2.69	2.50
8	7.50	6.67	6.00	5.71	5.33	5.00	4.80	4.71	4.62	4.44	4.29	4.14	4.00	3.87	3.81	3.75	3.48	3.33	3.20	3.12	3.08	2.86
9	8.44	7.50	6.75	6.43	6.00	5.63	5.40	5.29	5.19	5.00	4.82	4.66	4.50	4.36	4.29	4.22	3.91	3.75	3.60	3.51	3.46	3.21
10	9.38	8.33	7.50	7.14	6.67	6.25	6.00	5.88	5.77	5.56	5.36	5.17	5.00	4.84	4.76	4.69	4.35	4.17	4.00	3.90	3.85	3.57
11	10.31	9.17	8.25	7.86	7.33	6.88	6.60	6.47	6.35	6.11	5.89	5.69	5.50	5.32	5.24	5.16	4.78	4.58	4.40	4.29	4.23	3.93
12	11.25	10.00	9.00	8.57	8.00	7.50	7.20	7.06	6.92	6.67	6.43	6.21	6.00	5.81	5.71	5.63	5.22	5.00	4.80	4.68	4.62	4.29
13	12.19	10.83	9.75	9.29	8.67	8.13	7.80	7.65	7.50	7.22	6.96	6.73	6.50	6.29	6.19	6.09	5.65	5.42	5.20	5.07	5.00	4.64
14	13.13	11.67	10.50	10.00	9.33	8.75	8.40	8.24	8.08	7.78	7.50	7.24	7.00	6.77	6.67	6.56	6.09	5.83	5.60	5.45	5.38	5.00
15	14.06	12.50	11.25	10.71	10.00	9.38	9.00	8.82	8.66	8.33	8.04	7.76	7.50	7.26	7.14	7.03	6.52	6.25	6.00	5.84	5.77	5.36
16	15.00	13.33	12.00	11.43	10.67	10.00	9.60	9.41	9.23	8.89	8.57	8.28	8.00	7.74	7.62	7.50	6.96	6.67	6.40	6.23	6.15	5.71
17	15.94	14.17	12.75	12.14	11.33	10.63	10.20	10.00	9.81	9.44	9.11	8.79	8.50	8.23	8.10	7.97	7.39	7.08	6.80	6.62	6.54	6.07
18	16.88	15.00	13.50	12.86	12.00	11.25	10.80	10.59	10.39	10.00	9.64	9.31	9.00	8.71	8.57	8.44	7.83	7.50	7.20	7.01	6.92	6.43
19	17.82	15.83	14.25	13.57	12.67	11.88	11.40	11.18	10.96	10.56	10.18	9.83	9.50	9.19	9.05	8.91	8.26	7.92	7.60	7.40	7.31	6.79
20	18.75	16.67	15.00	14.29	13.33	12.50	12.00	11.76	11.54	11.11	10.72	10.35	10.00	9.68	9.52	9.38	8.70	8.33	8.00	7.79	7.69	7.14
21	19.69	17.50	15.75	15.00	14.00	13.13	12.60	12.35	12.12	11.67	11.25	10.86	10.50	10.16	10.00	9.85	9.13	8.75	8.40	8.18	8.08	7.50
22	20.63	18.33	16.50	15.71	14.67	13.75	13.20	12.94	12.69	12.22	11.79	11.38	11.00	10.65	10.48	10.31	9.57	9.17	8.80	8.57	8.46	7.86
23	21.57	19.17	17.25	16.43	15.33	14.38	13.80	13.53	13.27	12.78	12.32	11.90	11.50	11.13	10.95	10.78	10.00	9.58	9.20	8.96	8.85	8.21
24	22.50	20.00	18.00	17.14	16.00	15.00	14.40	14.12	13.85	13.33	12.86	12.42	12.00	11.61	11.43	11.25	10.43	10.00	9.60	9.35	9.23	8.57

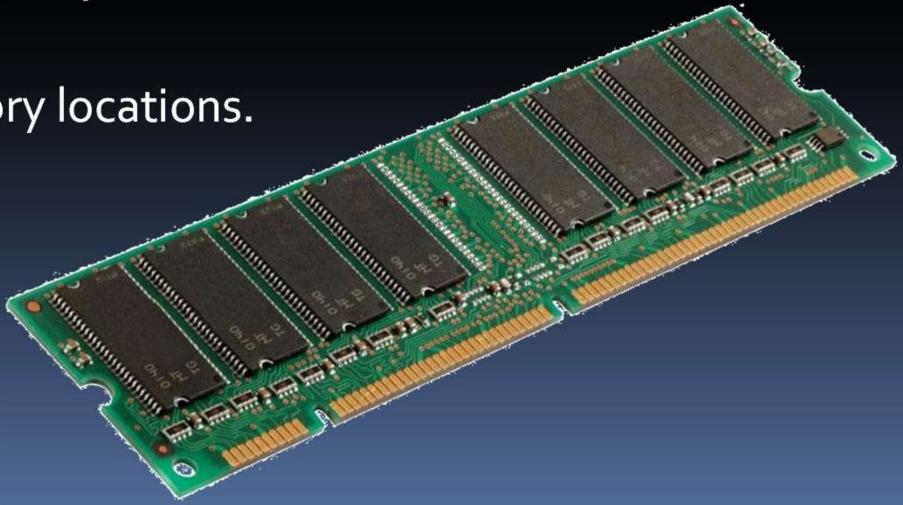
Register vs Memory Access

- Memory access can take multiple clock cycles (compared to a single clock cycle for registers)
- What can cause these extra delays?
  - Time needed to send write signal & data
  - Time to hold data & write signals high
  - Time to hold data signal high after write signal
- To understand this better, let's look at memory read and write operations in more detail.

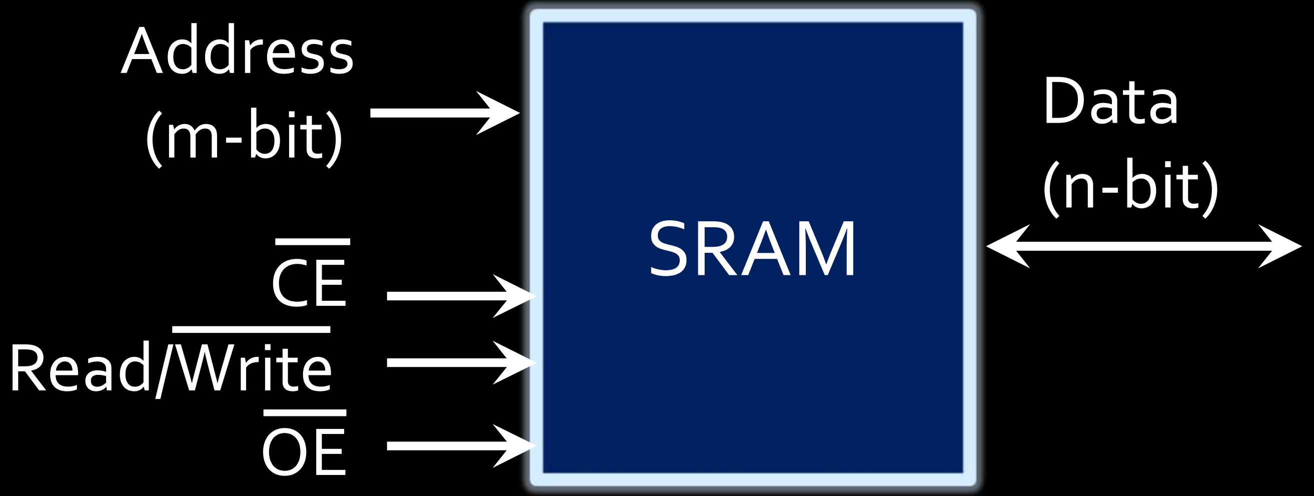
RAM Memory Signals

- Read/$\overline{\text{Write}}$ Enable (or R/$\overline{\text{W}}$, or two separate signals) – Input
  - Memory write: Memory is modified if this signal is low.
  - Memory read: Memory is read if this signal is high.
- Data In – Input
  - The data to write (store in memory) if R/$\overline{\text{W}}$ is low.
- Data Out – Output
  - The data read from memory if R/$\overline{\text{W}}$ is high.

- Additional signals needed for memory units:
  - Address Port - Input
    - Takes in `m` bits to address 2^m memory locations.
  - Chip Enable - Input
    - Activates memory for read or write
  - Output Enable - Input
    - Accompanies data read
    - Activates tri-state buffers



Example: Asynchronous SRAM* Interface

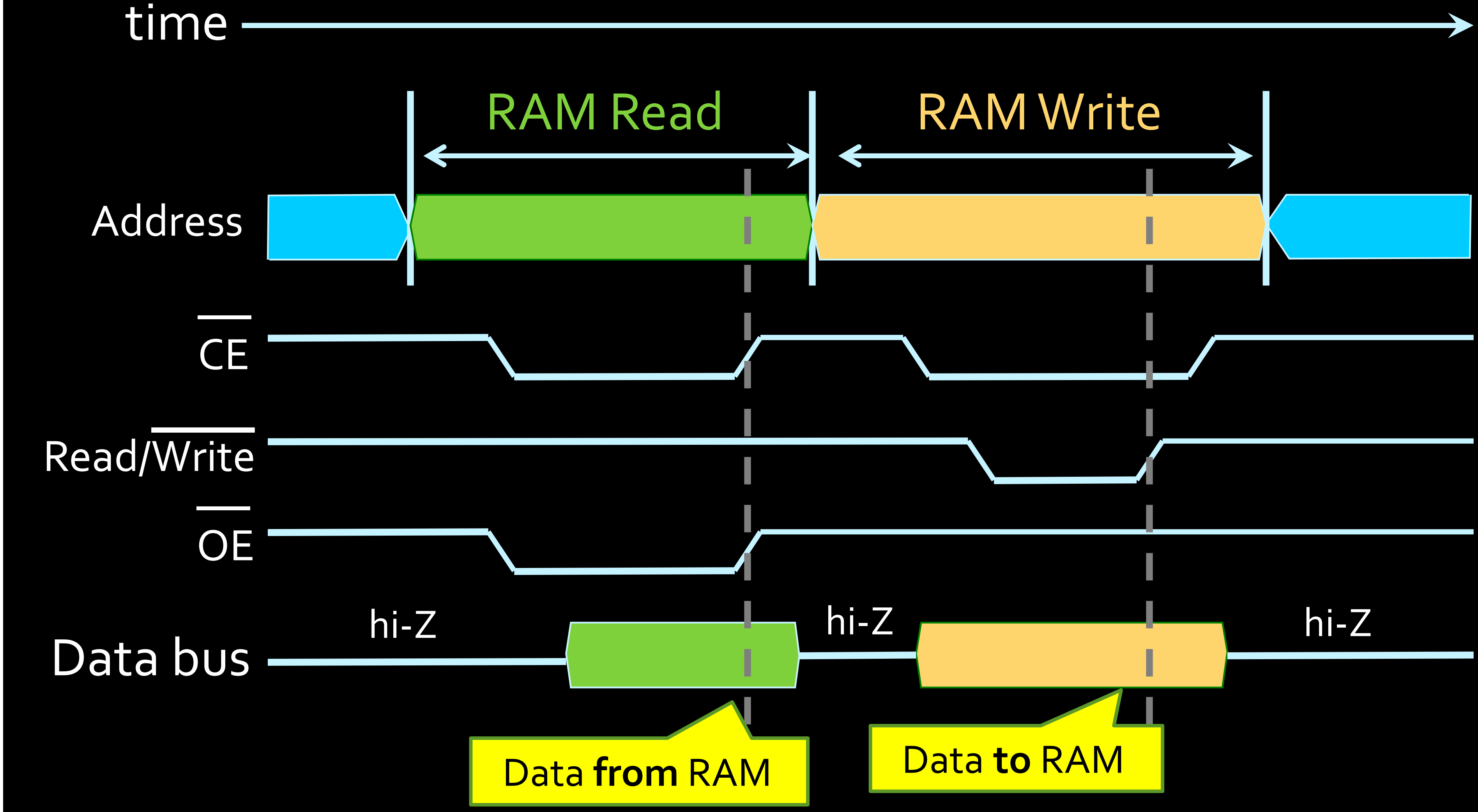


$\overline{\text{Chip Enable}}$ ($\overline{\text{CE}}$)	Read/$\overline{\text{Write}}$	$\overline{\text{Output Enable}}$ ($\overline{\text{OE}}$)	Access Type
0	0	1	SRAM Write
0	1	0	SRAM Read
1	X	X	SRAM not enabled

*SRAM = Static Random Access Memory

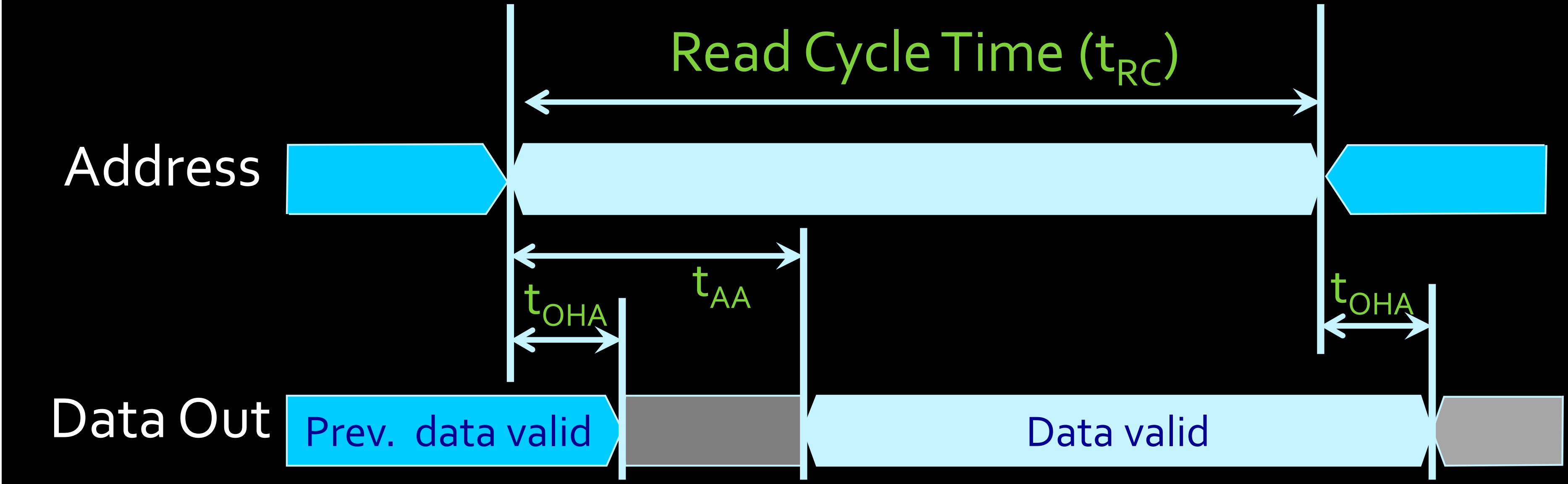
(don't worry too much about RAM vs SRAM vs DRAM)

Asynchronous RAM - Timing waveforms



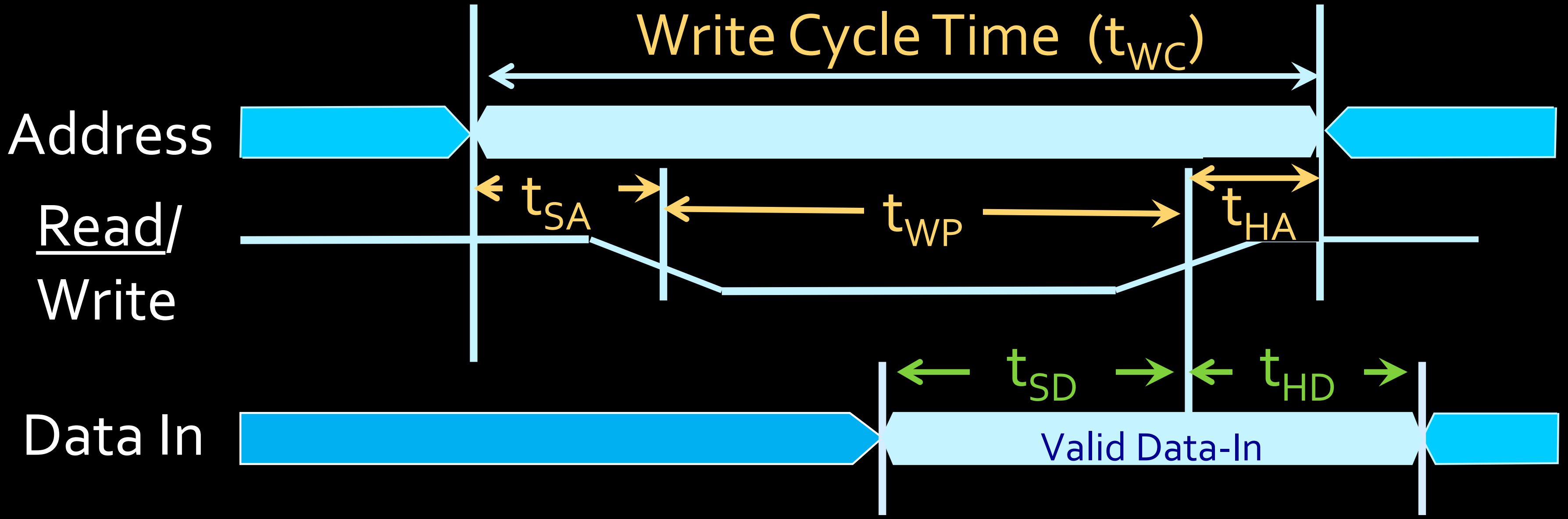
- Each memory read and write is done in stages.
- Each stage takes a certain amount of time.

Reading From Memory – Timing Constraints



- t_{RC} = Read Cycle time
  - Minimum time needed between two read cycles (min 10 ns).
- t_{AA} = Address Access time
  - Time needed for address to be stable before reading data values.
- t_{OHA} = Output Hold time
  - Time output data is held after change of address.
- This is an address-controlled read cycle. ($\overline{OE}$ is activated)

Writing To Memory – Timing Constraints

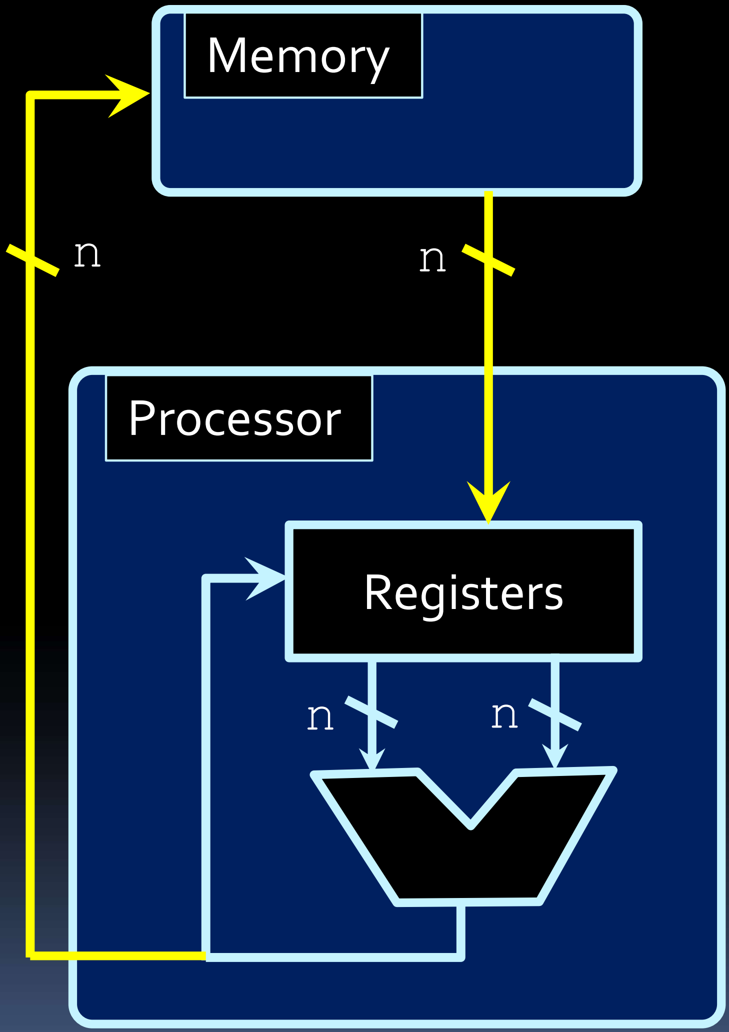


- t_{SA} = Address Setup Time
  - Time for address to be stable before enabling write signal.
- t_{HA} = Address Hold Time
  - Time for address to be stable after enabling write signal.
- t_{WP} = Write Pulse Width

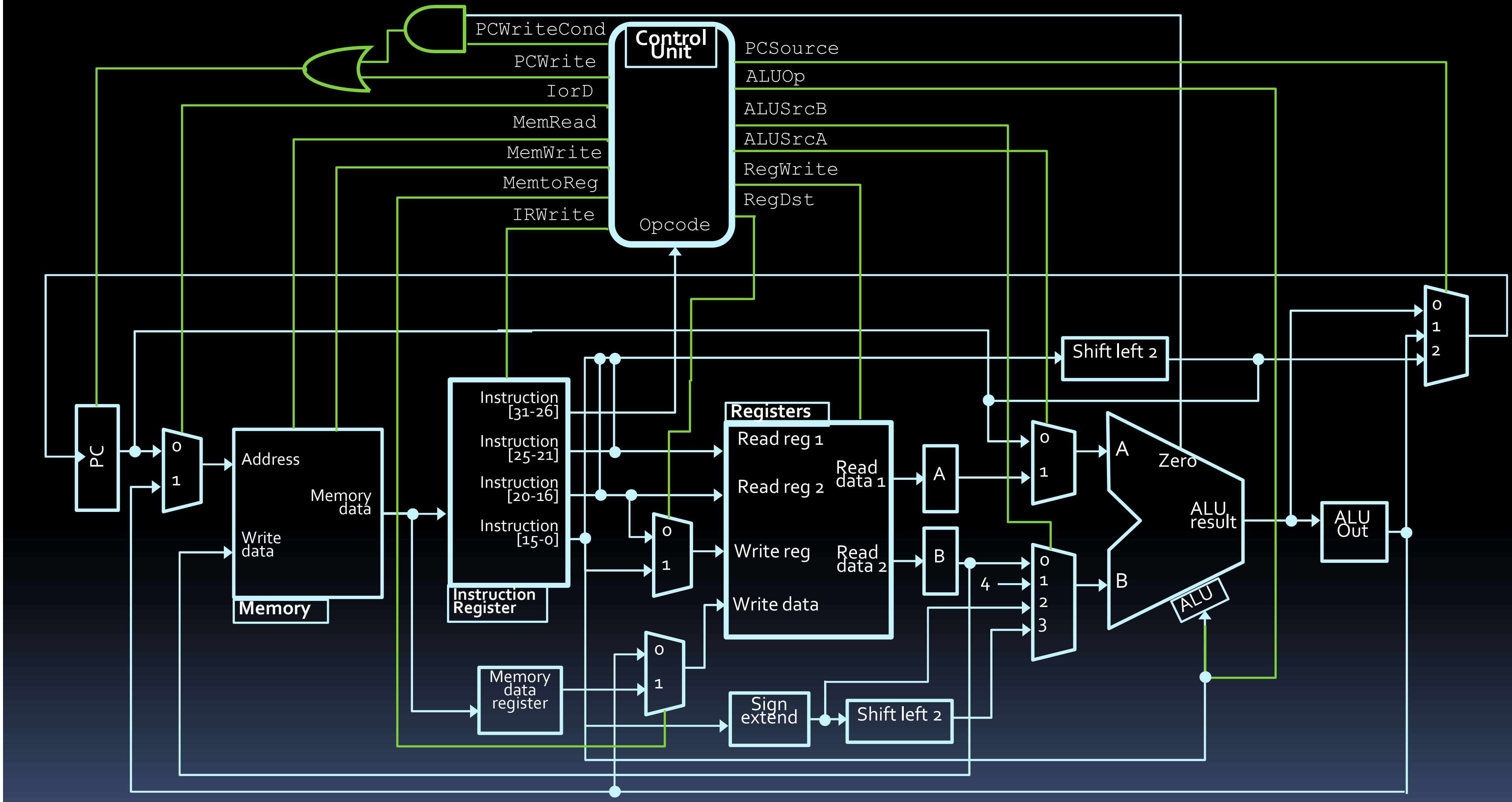
- t_{SD} = Data Setup Time (to Write End)
  - Time for data-in value to be set-up at destination.
- t_{HD} = Data Hold Time (from Write End)
  - Time data-in value should stay unchanged after write signal changes.

Memory Data Transfer

- Getting back to our earlier abstraction of the processor's datapath:
  - We fetch values from memory into the registers,
  - Process these values using the ALU,
  - When our overall calculation is complete, we return values back to memory.
  - Also known as a load-store architecture.
- But this is how it operates. What controls the datapath?



The processor datapath diagram



- What controls the memory, registers and ALU?

The Control Unit

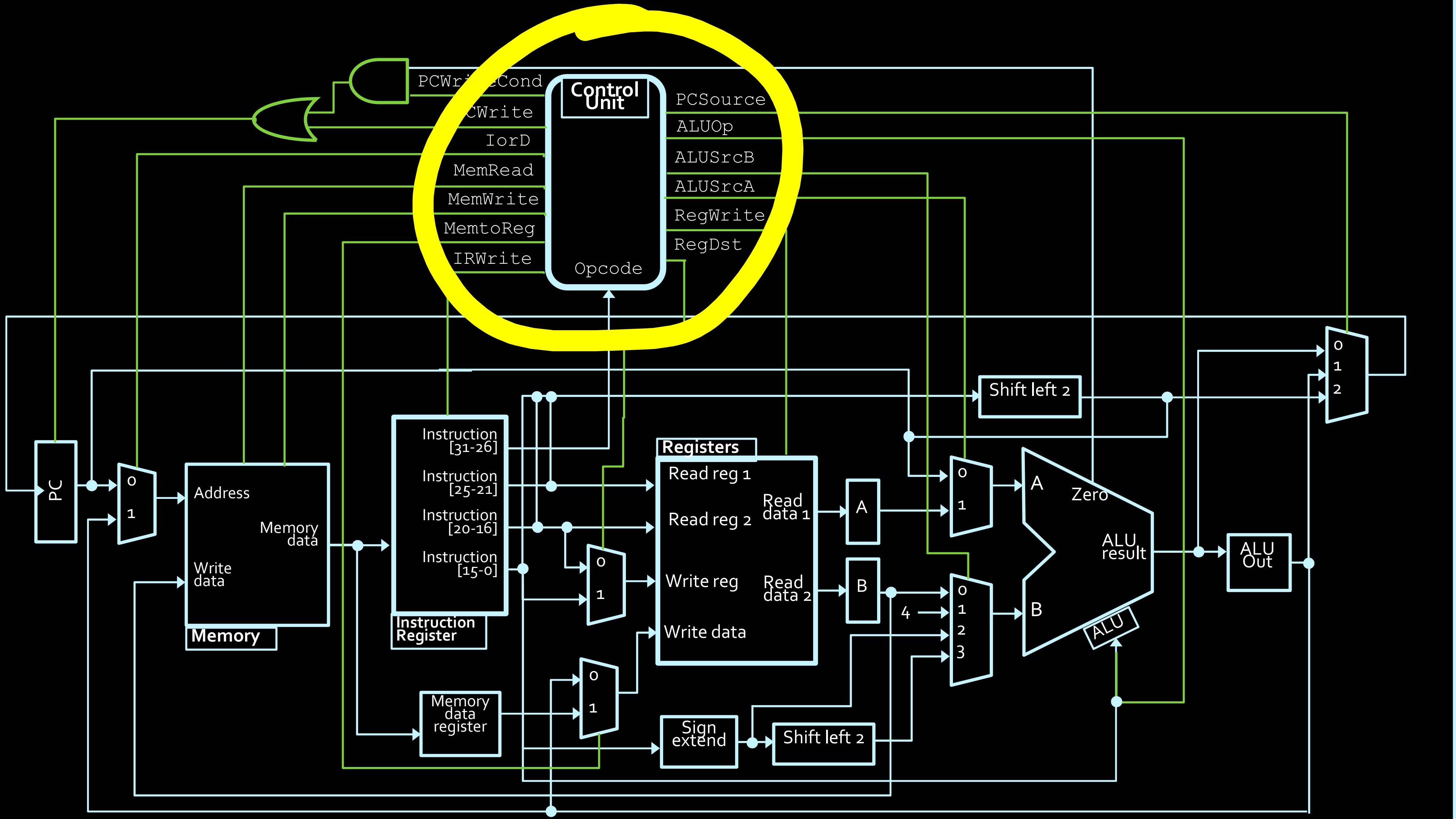
The processor datapath

- Operations are executed by turning various parts of the datapath on and off, to direct the flow of data from the correct source to the correct destination.
- What tells the processor to turn on these various components at the correct times?

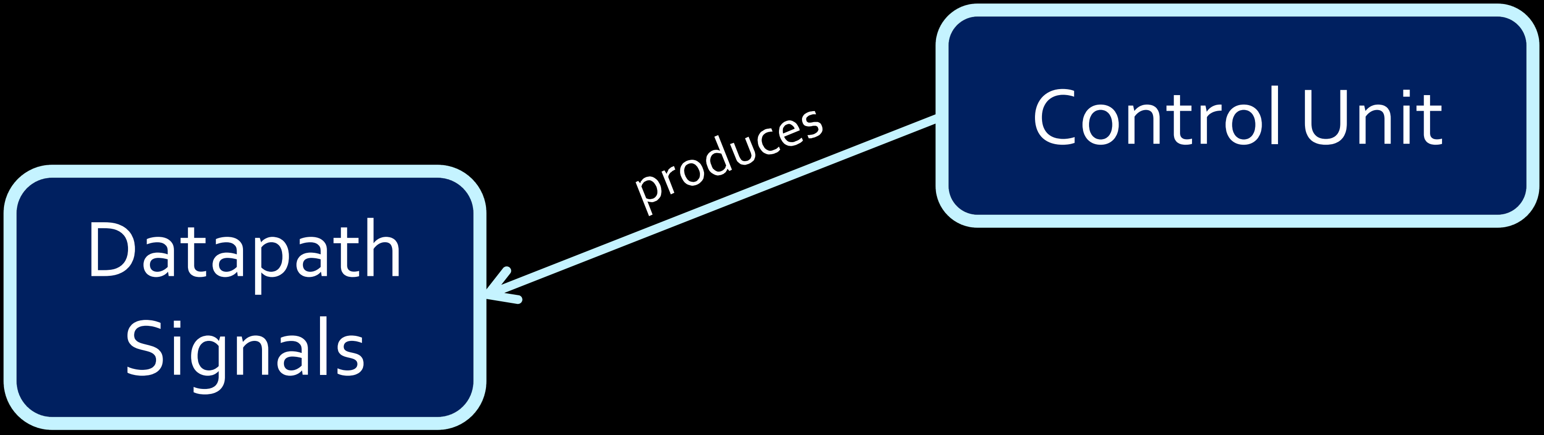
The Control Unit

- The datapath of a processor is a description/ illustration of how the data flows between processor components during the execution of an operation:
  - Where the data is coming from (the source),
  - Where it's going to (the destination), and
  - How the data is being processed (the operation).
- The control unit is an FSM that controls that datapath by sending signals (green lines in the previous schematic) to various processor components to enact all possible operations.
  - Think Lab 6 ☺

- The control unit sends signals (green lines) to various processor components to enact all possible operations.



How things fit together



- What goes into designing this control unit?
- What needs to be different from Lab 6?

Control Unit Design

How is this control unit different from the one you made in Lab 6?

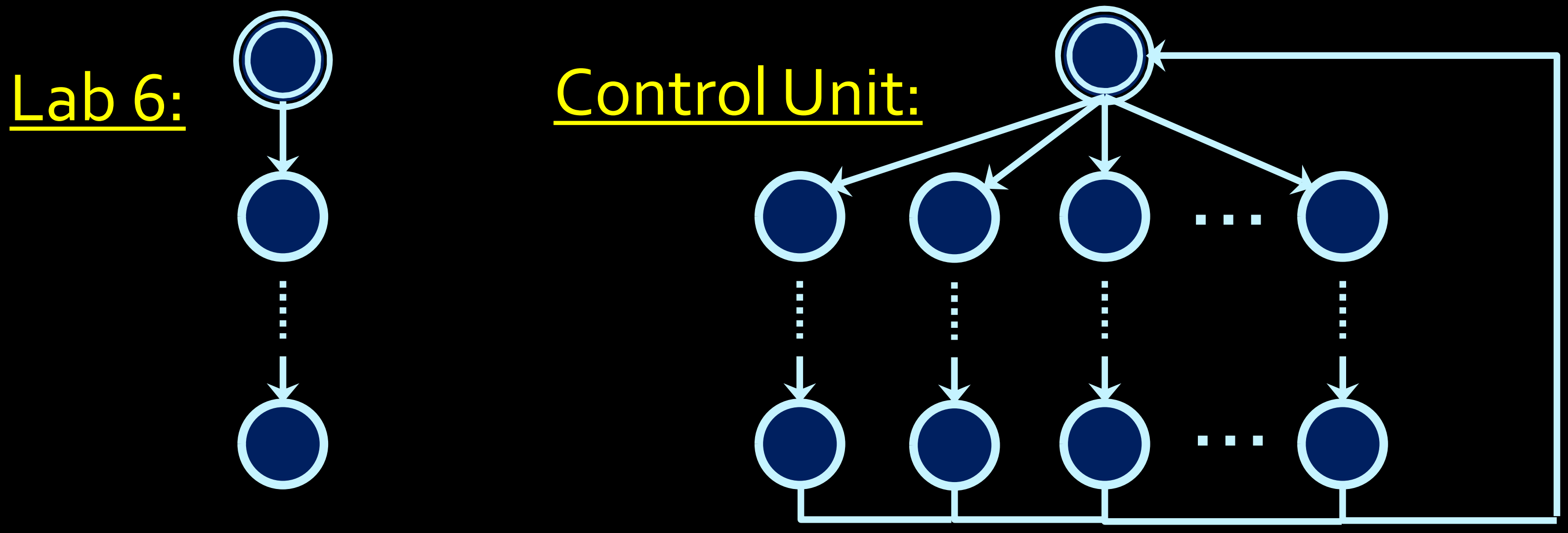
FSM from Lab 6

- Only supports 1 operation.
- Ends when operation is complete.
- No input signals to FSM.
- Only performs ALU + data operations.

Full Datapath FSM

- Supports hundreds of operations.
- Once operation is complete, returns to start state.
- 6-bit “operation code” input.
- Also responsible for setting up next opcode input.

Control Unit Design



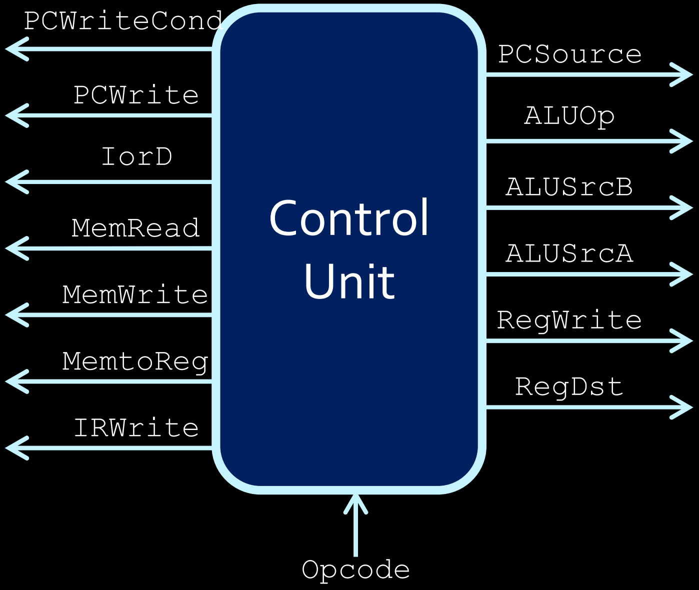
Lab 6:

Control Unit:

- Control unit receives “operation code” input to determine which branch to take from start state.
- The next operation code value (opcode) needs to be ready before the branch terminates and the FSM returns to the start state.

Control unit signals

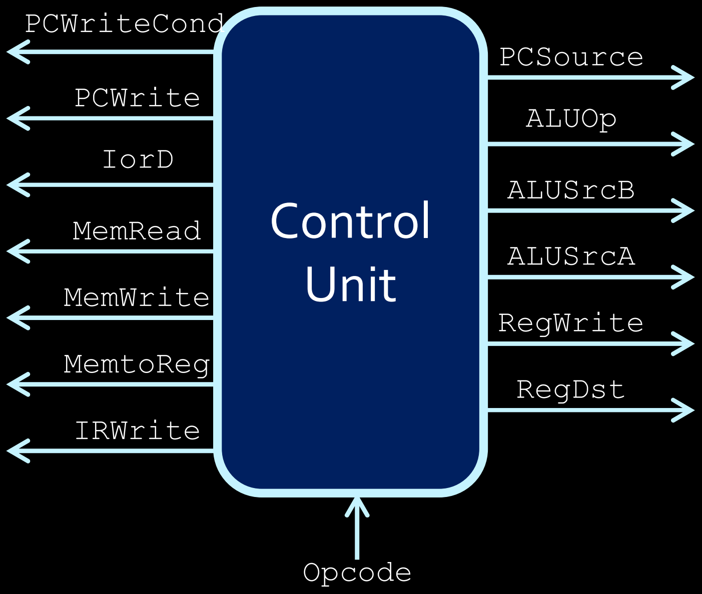
- The control unit takes in the opcode from the current instruction (more on that later), and sends signals to the rest of the processor.



- Within the control unit, the finite state machine can occupy multiple clock cycles for a single instruction.
  - The control unit send out different signals on each clock cycle, to make the overall operation happen.

Control unit signals

- Of the 13 signals shown here, you should be familiar with:
  - MemRead: Read from memory.
  - MemWrite: Write to memory.
  - MemToReg: Is the register value coming from memory or the ALU?
  - ALUOp (3 wires): ALU operation.
  - ALUSrcA: ALU source A.
  - ALUSrcB: ALU source B.
  - RegWrite: Writing new register value?



Control Unit Input

- Before we discuss the rest of the output signals, let’s return to the opcode input signal.
- How does the control unit use this to know what operation to perform?
  - The opcode comes from an instruction (located in a special location called an instruction register).
  - Encoded in the instruction is the operation to perform and additional information about the operation provided to the rest of the processor.
  - The control unit is responsible for executing this operation and then loading the next instruction to run.

Instruction Architecture

Understanding Instructions

- Instructions are 32-bit binary strings that encode:
  - the operation to perform (first 6 bits, aka the opcode),
  - other details needed to perform it (the remaining 26 bits).
- For 64-bit architectures, instructions are 64 bits long.

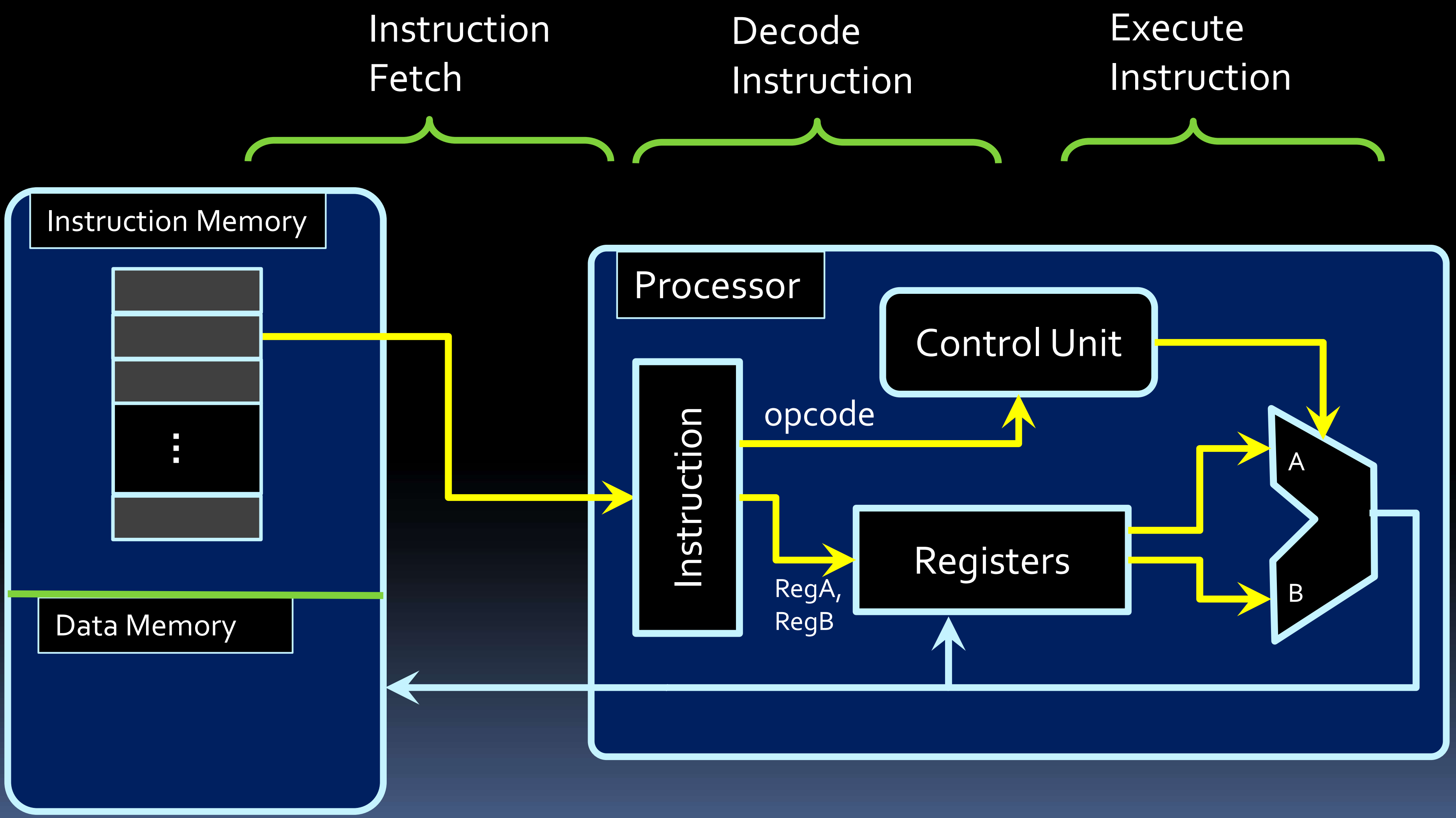
- Other details:
  - Instructions and data values are both stored in main memory.
    - The stack is stored in memory too (but more on that later)
  - Instructions are stored separately from data values.
    - Often identified as the `.text` segment of memory
    - Data values occupy the rest of memory (the `.data` segment)
  - The first instruction to be executed in a program is usually identified with a label `main:`

Instruction Execution

- All programs are translated into a sequence of instructions. To execute them, the control unit continually does the following steps:

1. Instruction Fetch
   - Bring the next instruction from memory and place it into the instruction register.
2. Decode Instruction
   - Based on the instruction’s type, determine what operation to perform.
3. Execute instruction
   - Read the values (contents) of any registers needed from the register file, and perform any computations needed in the ALU.
   - Access memory if we need to read or write data.
   - Write back any data that needs to be stored in memory or registers.
4. Move (or jump) to the next instruction in memory.

Instruction Execution

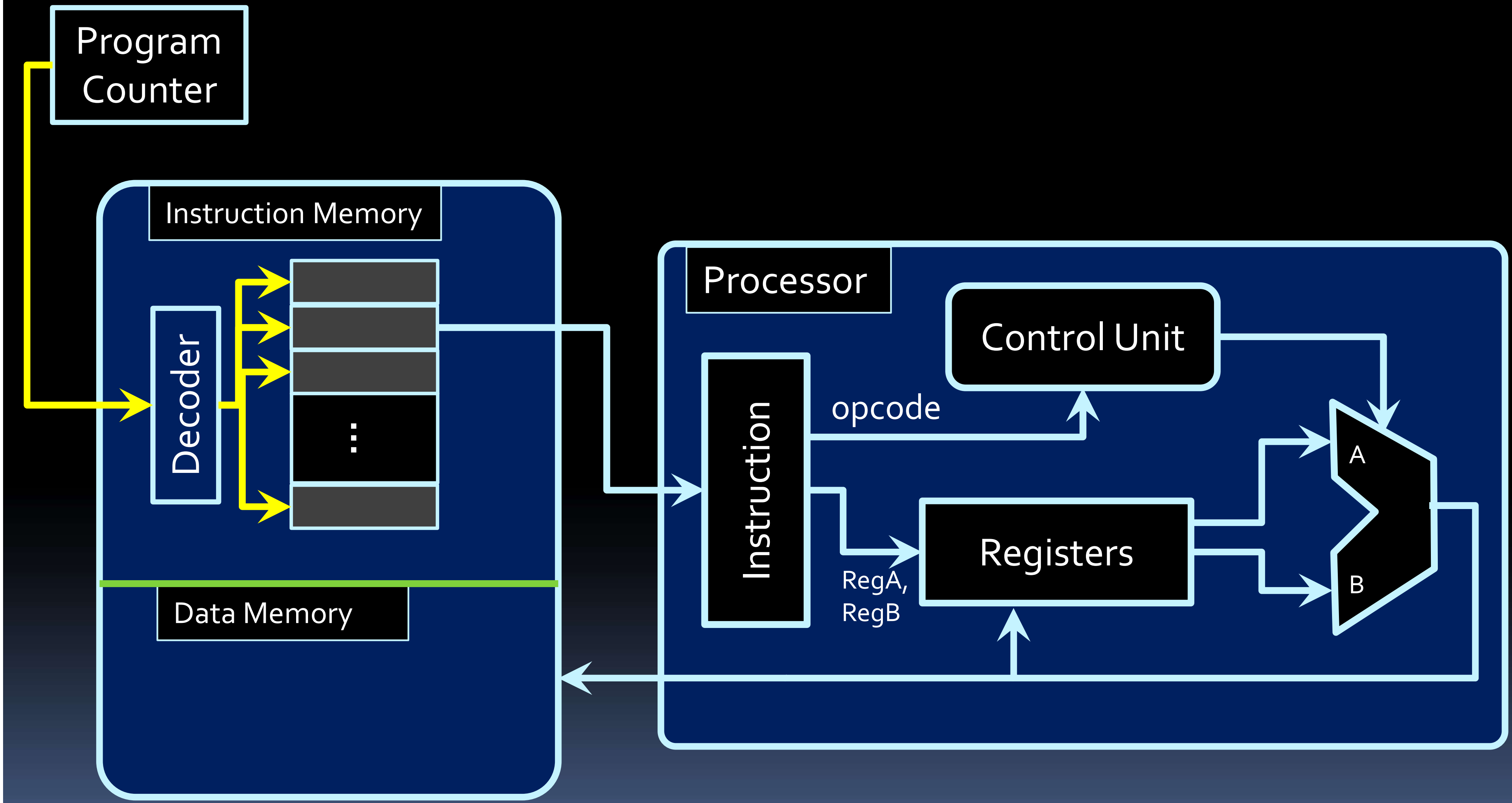


The Program Counter (PC)

- Steps #1 and #4 of instruction execution assume that the control unit knows where to find the current instruction in memory.
    - Makes sense to have a special register for that!

- This special register is called the program counter (or PC), and stores the location (memory address) of the current instruction.

Enter the Program Counter



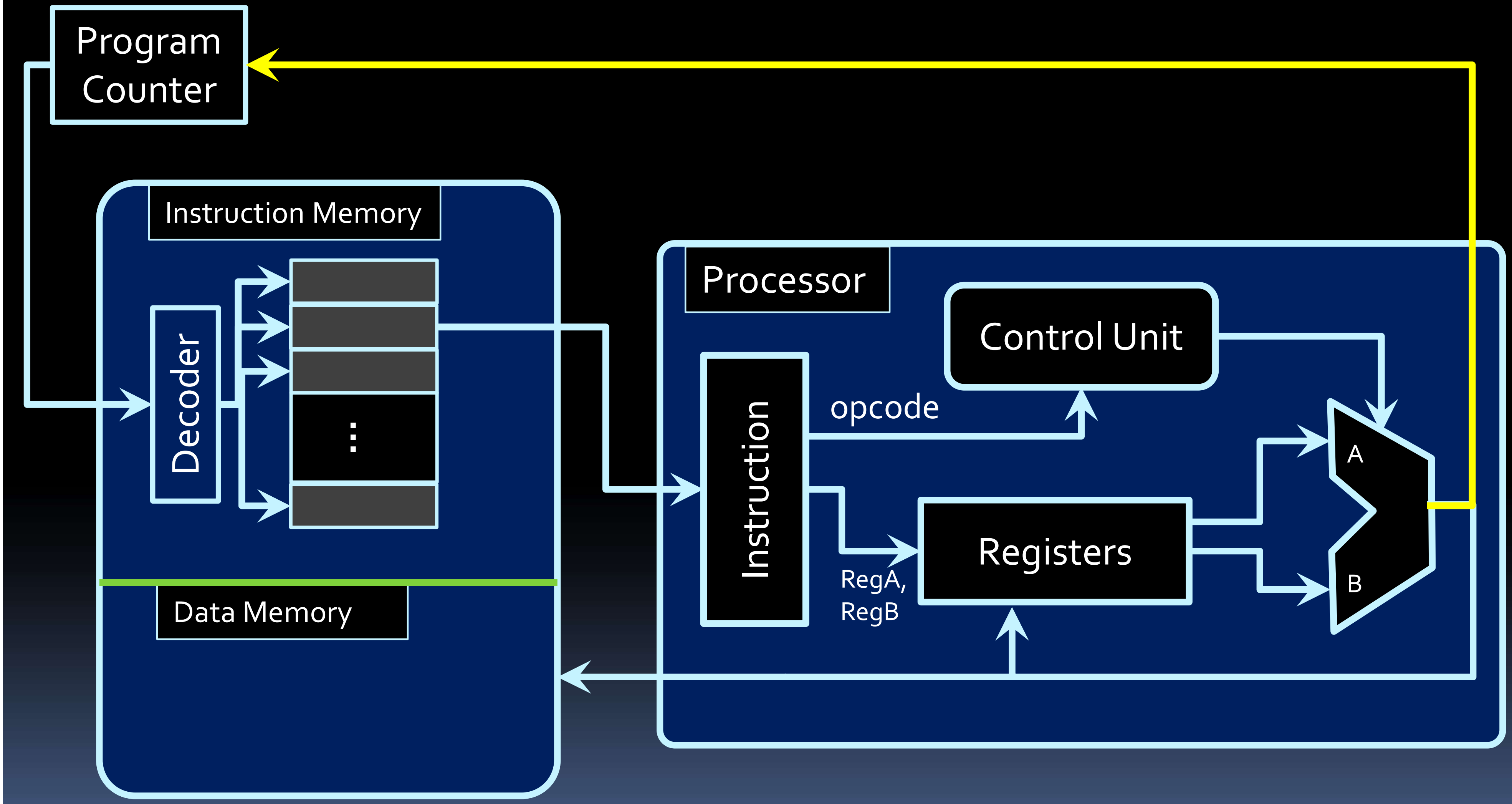
Updating The Program Counter

- How does the Program Counter get updated?
- Usually instructions are executed in sequential order (i.e., one after the other).
  - Doesn't mean that PC is incremented by one each time!
  - Memory locations are *byte-addressable*, meaning that each byte (8 bits) has its own unique address.
  - Since instructions are 32 bits long (i.e. 4 bytes), instructions addresses would be at `0`, `4`, `8`, `12`, `16`, etc.
- Therefore, the PC needs to be incremented by 4 each time it needs to fetch the next instruction.
  - Every instruction ends with the PC update and next instruction fetch.

Updating The Program Counter (cont'd)

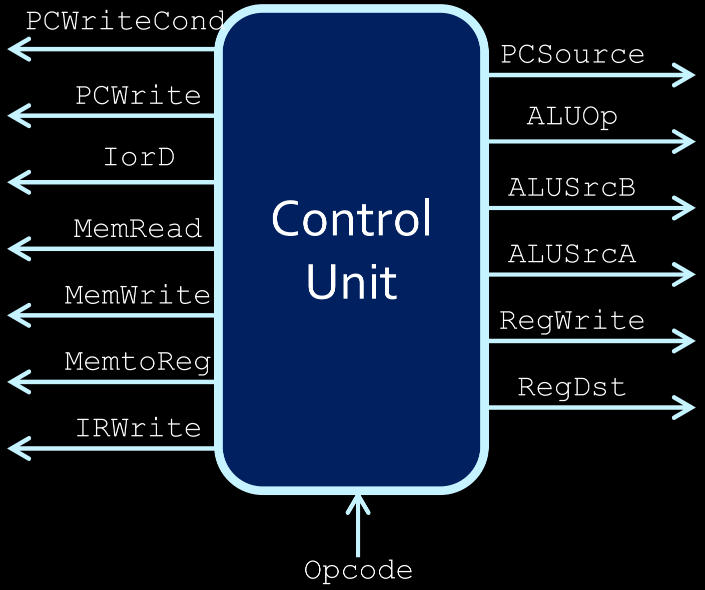
- The exception to the +4 rule:
  - We don't always execute instructions in sequential order (think about if-else statements, loops and function calls).
- Some instructions change the PC differently, by jumping to locations in memory.
  - How? → The ALU needs to calculate the new PC value.
  - Branches, jumps and function calls are executed this way.
- We will come back to these special instructions later, after talking more about decoding instructions.

Program Counter Integration



Control unit signals

- Now these other signals make more sense ☺
  - PCWrite: Write the ALU output to the PC.
  - PCWriteCond: Write the ALU output to the PC if the ALU's Zero condition is true.
  - IorD: Signal whether we're fetching an instruction or data from memory.
  - IRWrite: Load the instruction register.
  - PCSource: Set the source of the PC value.



Control unit signal summary

- PCWrite: Write the ALU output to the PC.
- PCWriteCond: Write the ALU output to the PC, only if the Zero condition has been met.
- IorD: For memory access; short for “Instruction or Data”. Signals whether the memory address is being provided by the PC (for instructions) or an ALU operation (for data).
- MemRead: The processor is reading from memory.
- MemWrite: The processor is writing to memory.
- MemToReg: The register file is receiving data from memory, not from the ALU output.
- IRWrite: The instruction register is being filled with a new instruction from memory.

Control unit signal summary

- PCSource: Signals whether the value of the PC resulting from an jump, or an ALU operation.
- ALUOp (3 wires): Signals the execution of an ALU operation.
- ALUSrcA: Input A into the ALU is coming from the PC (value=0) or the register file (value=1).
- ALUSrcB (2 wires): Input B into the ALU is coming from the register file (value=0), a constant value of 4 (value=1), the instruction register (value=2), or the shifted instruction register (value=3).
- RegWrite: The processor is writing to the register file.
- RegDst: Which part of the instruction is providing the destination address for a register write (rt versus rd).

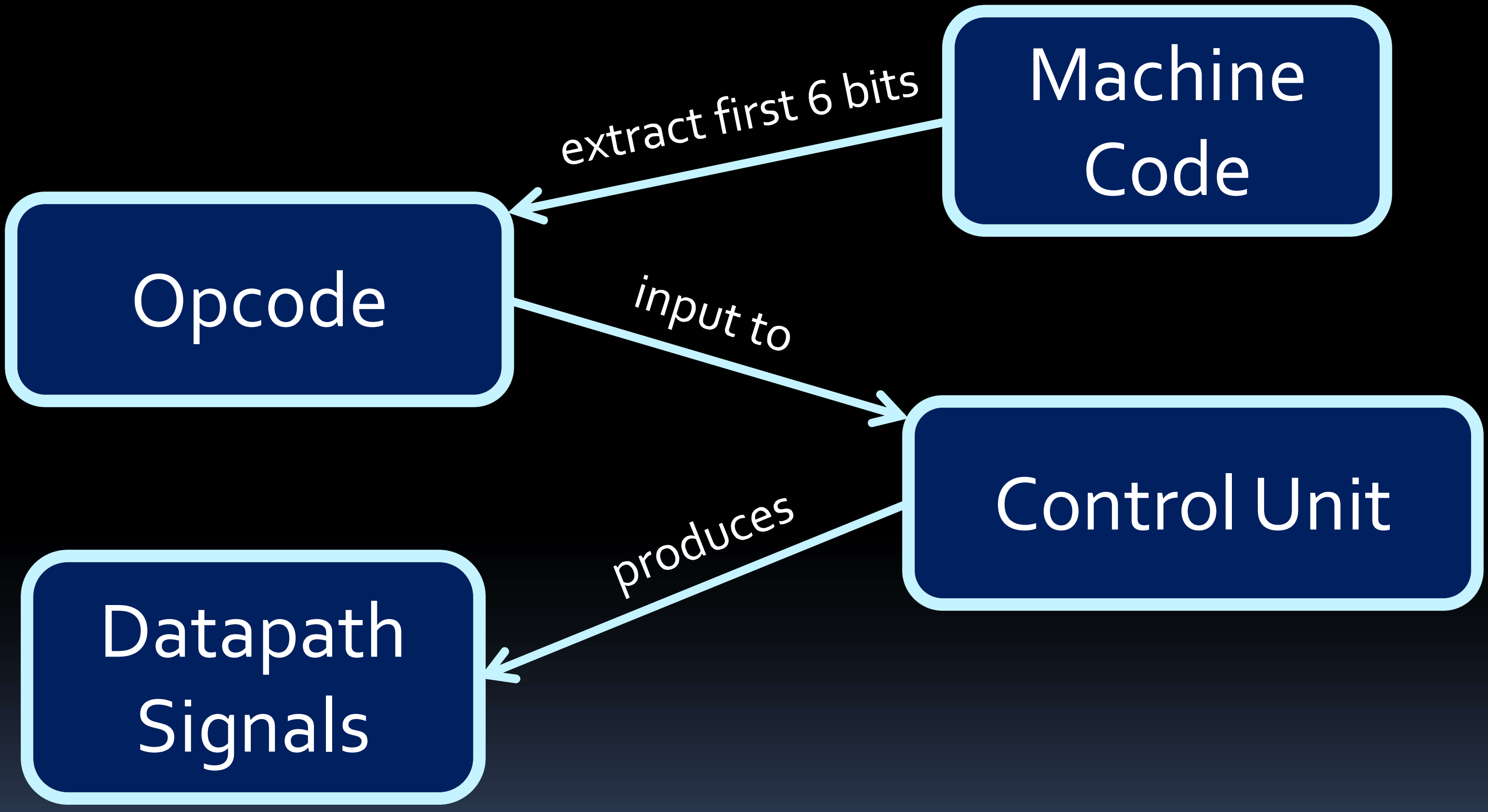
Example control unit signals

- **addi $t7, $t0, 42**

  (set Reg #15 = Reg #8 + 42)

  - PCWrite = 0
  - PCWriteCond = 0
  - IorD = X
  - MemWrite = 0
  - MemRead = 0
  - MemToReg = 0
  - IRWrite = 0
  - PCSource = X
  - ALUOp = Add
  - ALUSrcA = 1
  - ALUSrcB = 10
  - RegWrite = 1
  - RegDst = 0

How things fit together now



Machine Code

extract first 6 bits

Opcode

input to

Control Unit

produces

Datapath Signals

Intro to Machine Code

- Machine code instructions are 32 bits long, made of a sequence of `0`s and `1`s that don’t make intuitive sense to us, but make sense to the processor.
    - You could construct these instructions by hand, but most are created when programs are compiled.
- As soon as the next instruction is fetched from memory, the rest of the processor starts executing that instruction.
    - The control unit receives the opcode it needs,
    - Registers receive source and destination addresses,
    - The ALU receives the operation to perform (sometimes)

Decoding Instructions

- Let’s say we’ve fetched this 4 byte (32-bit) instruction:

```
00000000 00000001 00111000 00100011
```

…what is it telling us to do?

- This is specified (among other things) in the Instruction Set Architecture (ISA) that is implemented by a given processor.
  - Note: there are different ways to implement a given ISA in hardware (that’s called processor microarchitecture!)
  - We will be using the MIPS ISA in our lectures (more onwhat MIPS is later).

Instruction decoding

- Each instruction (also known as control words) can be broken down into sections that contain all the information needed to execute the operation.

- Example: unsigned subtraction (subu $d,$s,$t)

00000000 00000001 00111000 00100011

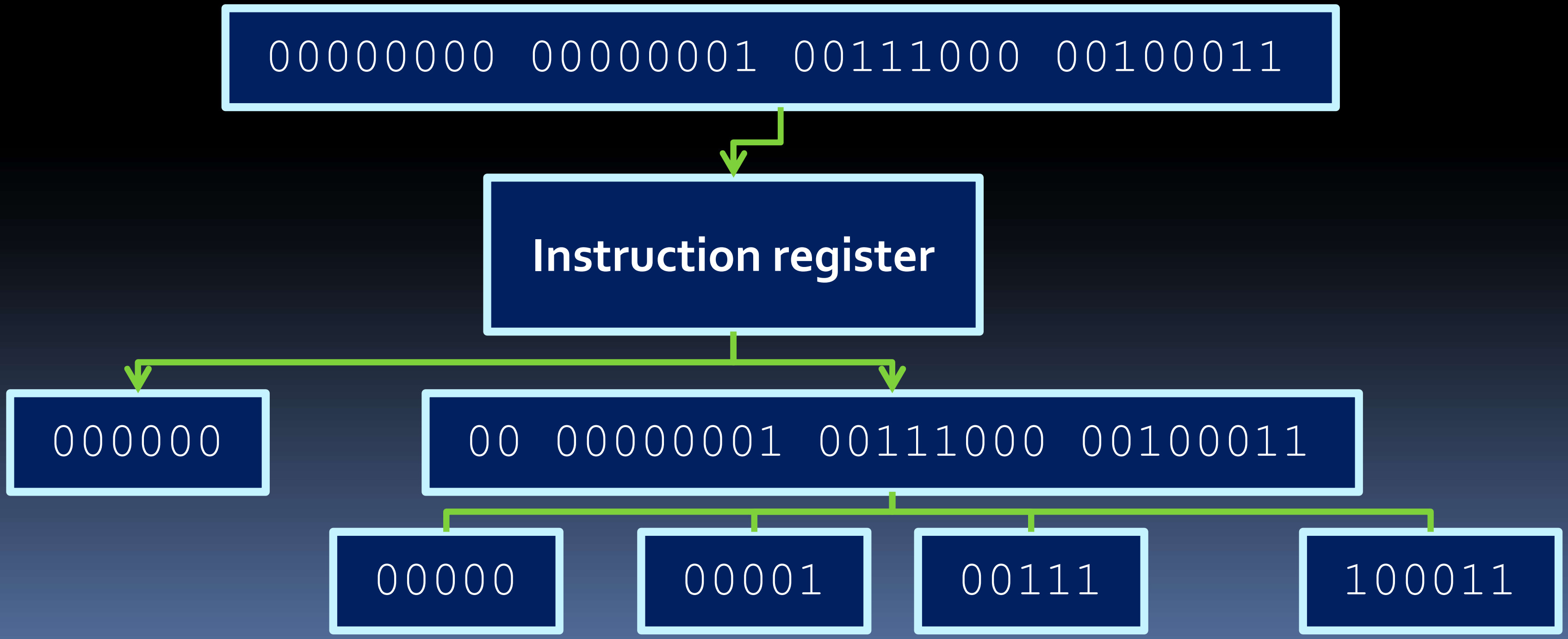
000000ss sssttttt ddddd~~000 00~~100011

Register 7 = Register 0 – Register 1

- Instruction length is usually constrained by the bus width (e.g. 32-bit architecture, 64-bit architecture).

Instruction registers

- The instruction register stores in the 32-bit instruction fetched from memory.
  - The first 6 bits (known as the opcode) specify the operation type, and how to decompose the rest.



00000000 00000001 00111000 00100011

Instruction register

000000

00 00000001 00111000 00100011

00000

00001

00111

100011

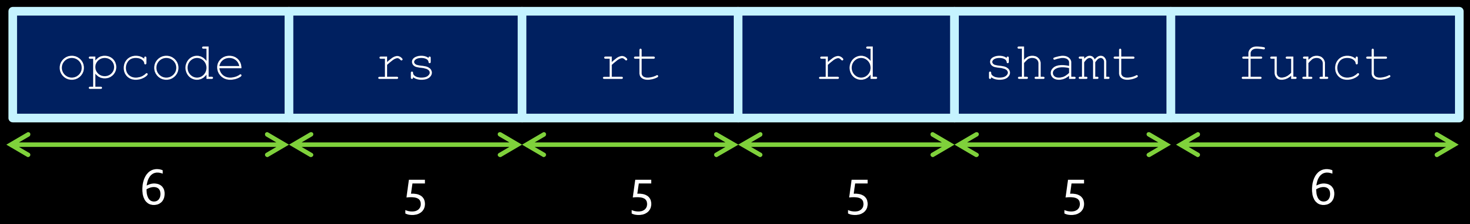
Opcode List

- This table shows the MIPS instructions and the opcode for each instruction.
- Yellow codes indicate "R-type" instructions
  - In these cases, the opcode is actually `000000` and the six digits listed in this table specify the function code.
- What does "R-type" instruction mean?

Instruction	Op/Func	Instruction	Op/Func
add	100000	srav	000111
addu	100001	srl	000010
addi	001000	srlv	000110
addiu	001001	beq	000100
div	011010	bgtz	000111
divu	011011	blez	000110
mult	011000	bne	000101
multu	011001	j	000010
sub	100010	jal	000011
subu	100011	jalr	001001
and	100100	jr	001000
andi	001100	lb	100000
nor	100111	lbu	100100
or	100101	lh	100001
ori	001101	lhu	100101
xor	100110	lw	100011
xori	001110	sb	101000
sll	000000	sh	101001
sllv	000100	sw	101011
sra	000011	mflo	010010

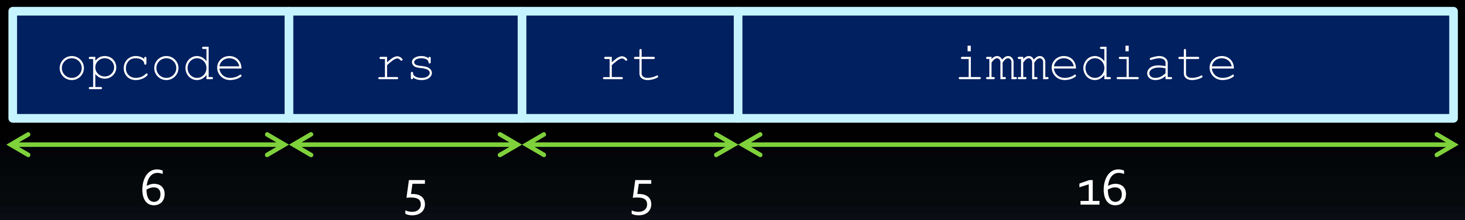
MIPS instruction types

- R-type:



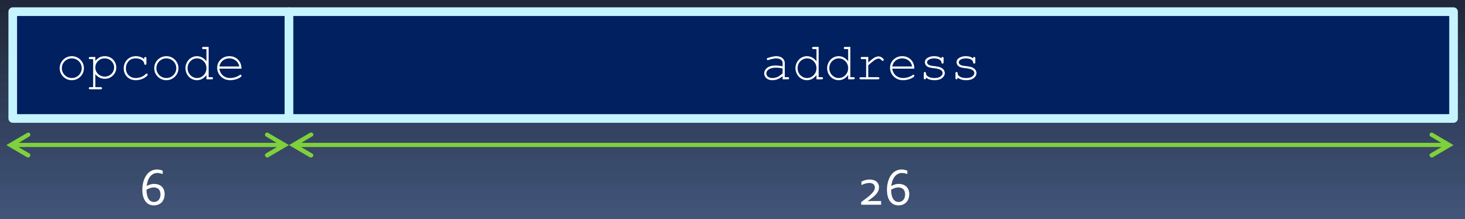
opcode	rs	rt	rd	shamt	funct
6	5	5	5	5	6

- I-type:



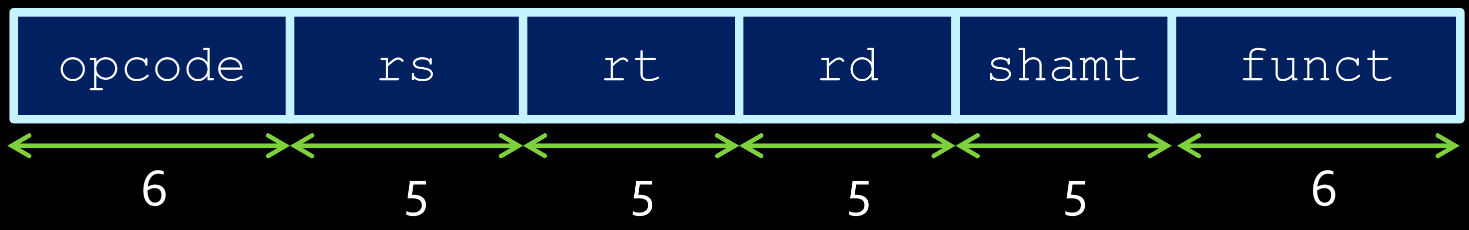
opcode	rs	rt	immediate
6	5	5	16

- J-type:



opcode	address
6	26

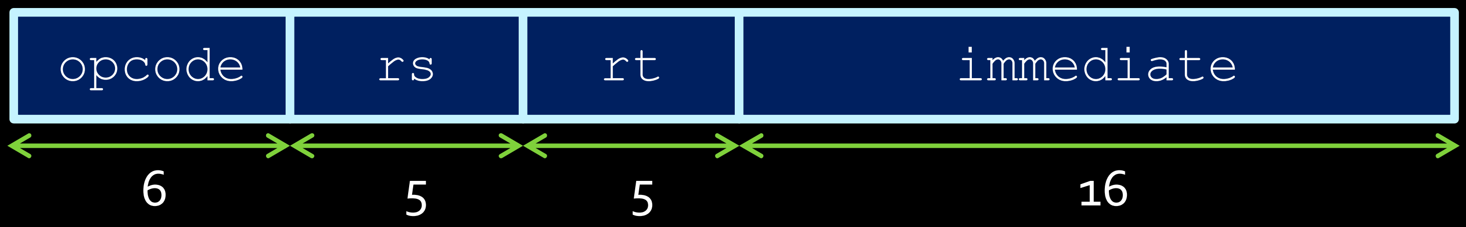
R-type instructions



opcode	rs	rt	rd	shamt	funct
6	5	5	5	5	6

- Short for “register-type” instructions.
  - Because they operate on the registers, naturally.
- These instructions have fields for specifying up to three registers and a shift amount.
  - Three registers: two source registers (`rs` & `rt`) and one destination register (`rd`).
  - A field is usually coded with all `0` bits when not being used.
- The opcode for all R-type instructions is `000000`.
- The function field specifies the type of operation being performed (add, sub, and, etc).

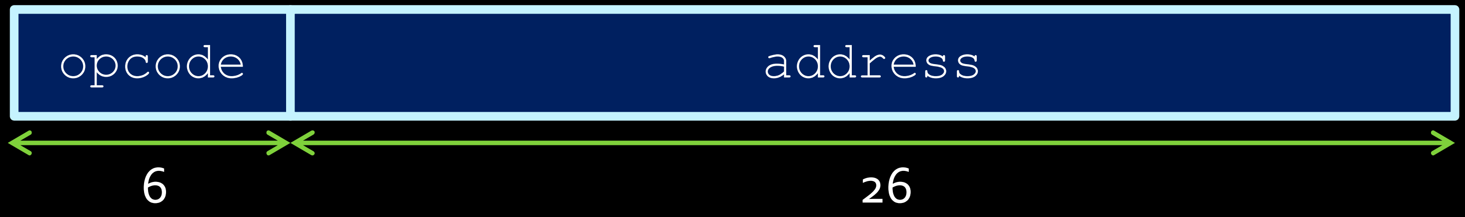
I-type instructions



opcode	rs	rt	immediate
6	5	5	16

- These instructions have a 16-bit immediate field.
- This field a constant value, which is used for:
  - an immediate operand,
  - a branch target offset, or
  - a displacement for a memory operand.
- For branch target offset operations, the immediate field contains the signed difference between the current address stored in the PC and the address of the target instruction.
  - This offset is stored with the two low order bits dropped. The dropped bits are always 0 since instructions are word-aligned.

J-type instructions



- Only two J-type instructions:
  - jump (`j`)
  - jump and link (`jal`)
- These instructions use the 26-bit coded address field to specify the target of the jump.
  - The first four bits of the destination address are the same as the current bits in the program counter.
  - The bits in positions `27` to `2` in the address are the 26 bits provided in the instruction.
  - The bits at positions `1` and `0` are always `0` since instructions are word-aligned.

MIPS ISA Attributes

- R-type MIPS instructions have 3 –operands:
  - 2 source registers
    - acting as data inputs for that instruction
  - 1 destination register
    - acting as output as in the result of the operation applied on the two source operands will be stored (written) into that destination register.

- It's a load-store architecture
  - There are only specific instructions that allow memory access (loads and stores).
  - You can't add a value stored in a register with a value stored in memory. Instead, you need to load that value from memory into a register first (with an earlier instruction).

Brainstorming!

- If you were to create your own low-level language that did a few basic logic operations, what would you do?
  - Which operations would you choose to include?
  - How would you name them?
  - How many source operands would each instruction have?
  - What would the format of the instruction be?
- More on this, coming soon….☺