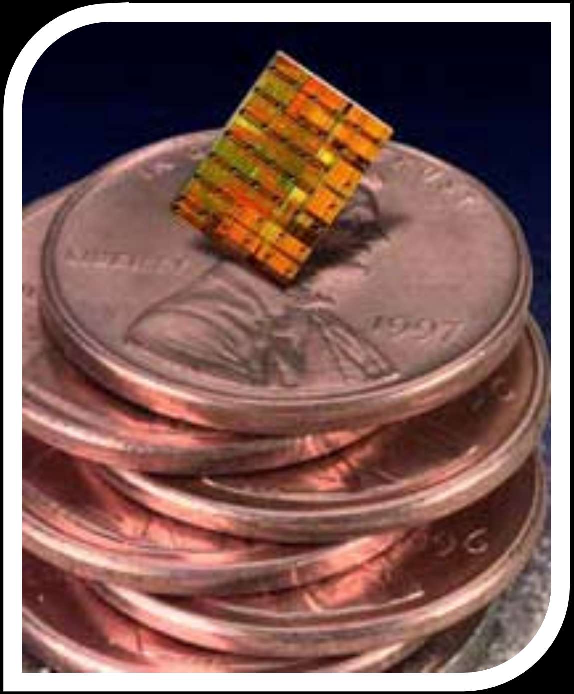
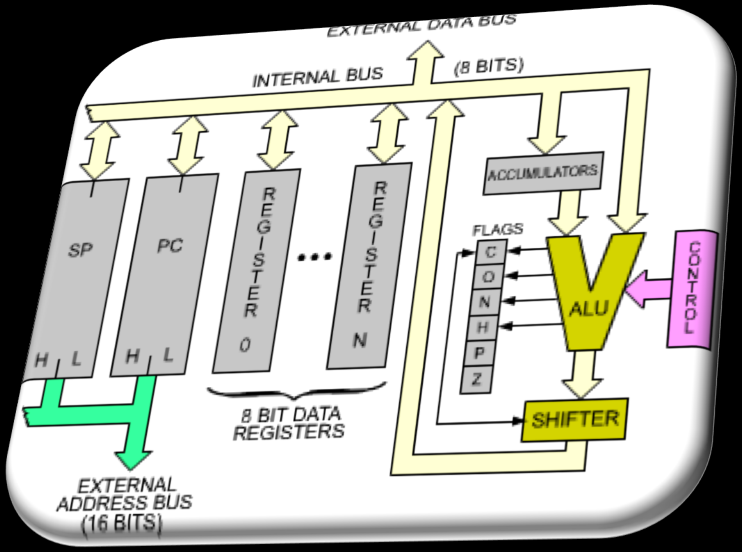
Week 6: Processor Components

Microprocessors



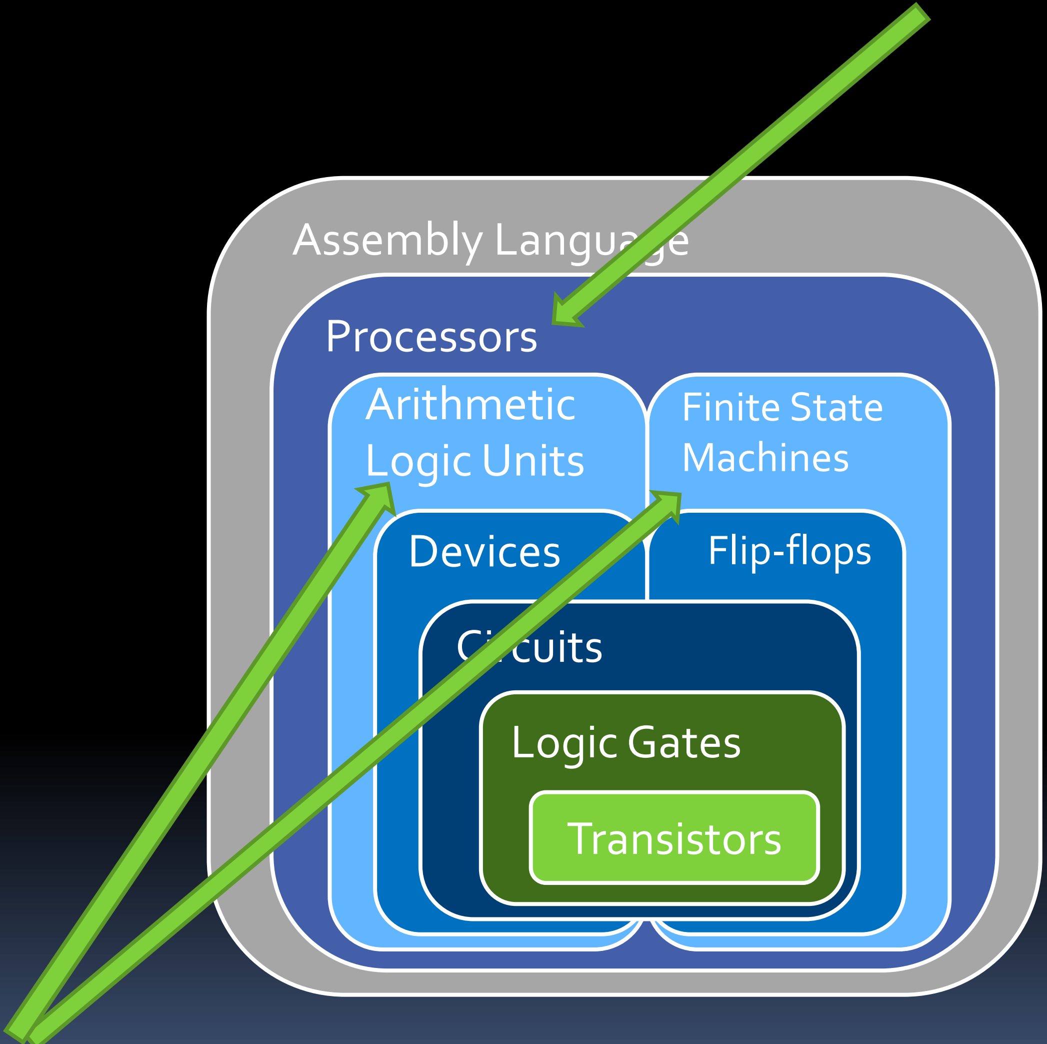
- So far, we've been making devices, such such as adders, counters and registers.
- The ultimate goal is to make a microprocessor, which is a digital device that processes input, can store values and produces output, according to a set of on-board instructions.

Microprocessors



- Microprocessors are a combination of the units that we've discussed so far:
    - Registers to store values.
    - Adders and shifters to process data.
    - Finite state machines to control the process.
- Microprocessors have been the basis of all computing since the 1970's, and can be found in nearly every sort of electronics.

To get to this



Assembly Language

Processors

Arithmetic Logic Units

Finite State Machines

Devices

Flip-flops

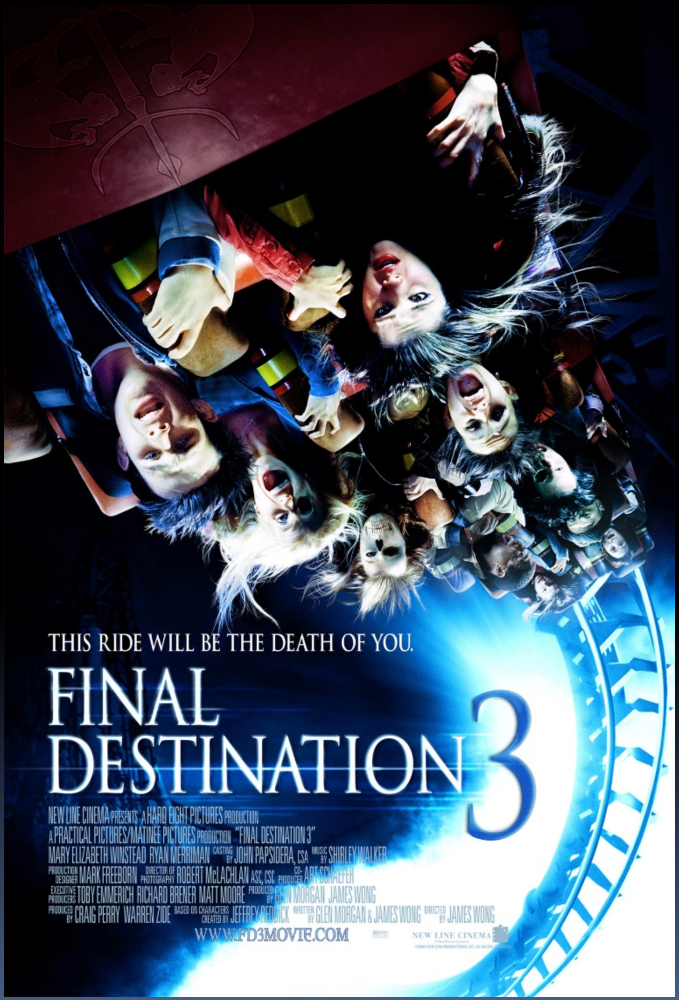
Circuits

Logic Gates

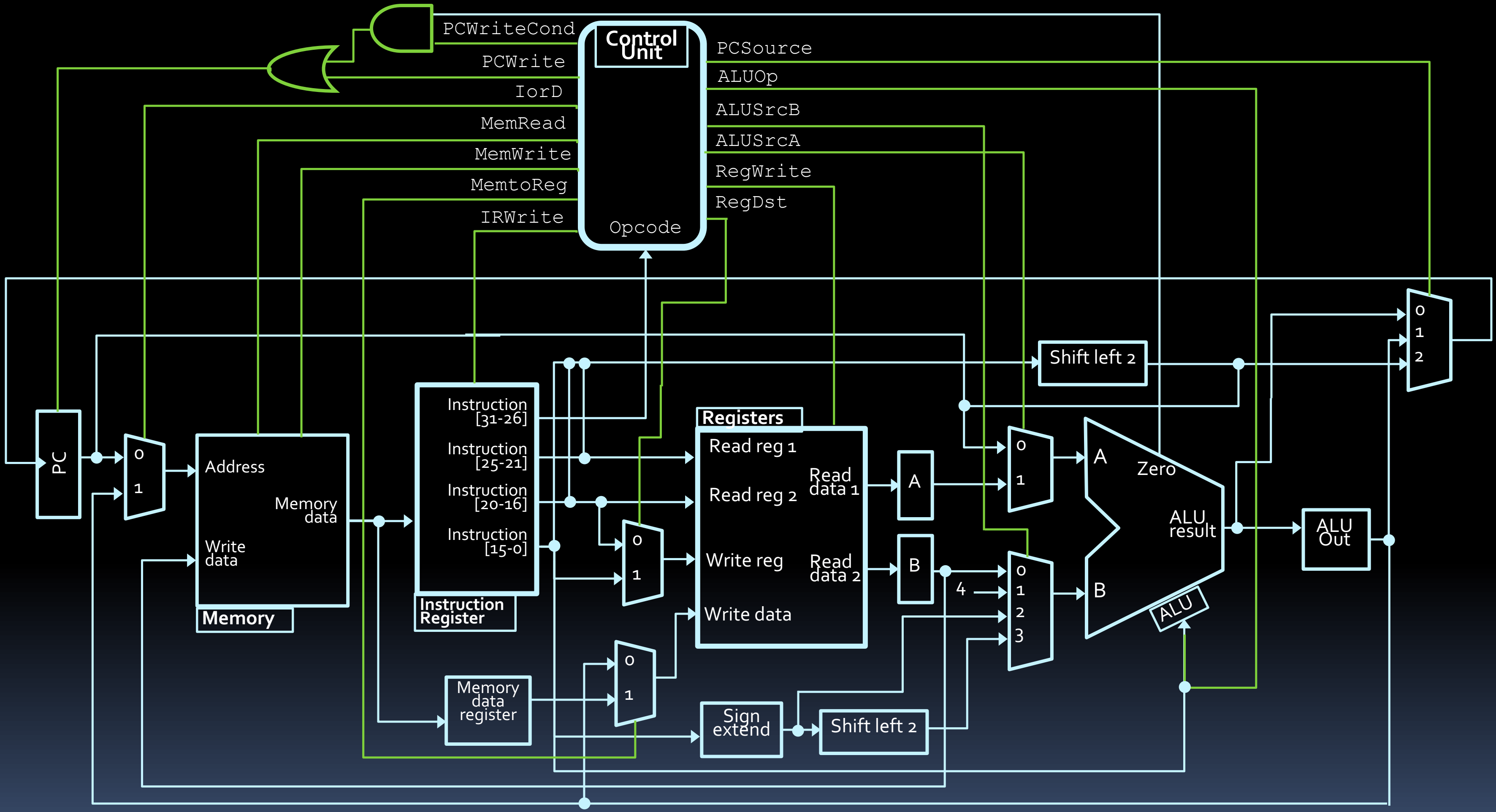
Transistors

We build these

The Final Destination

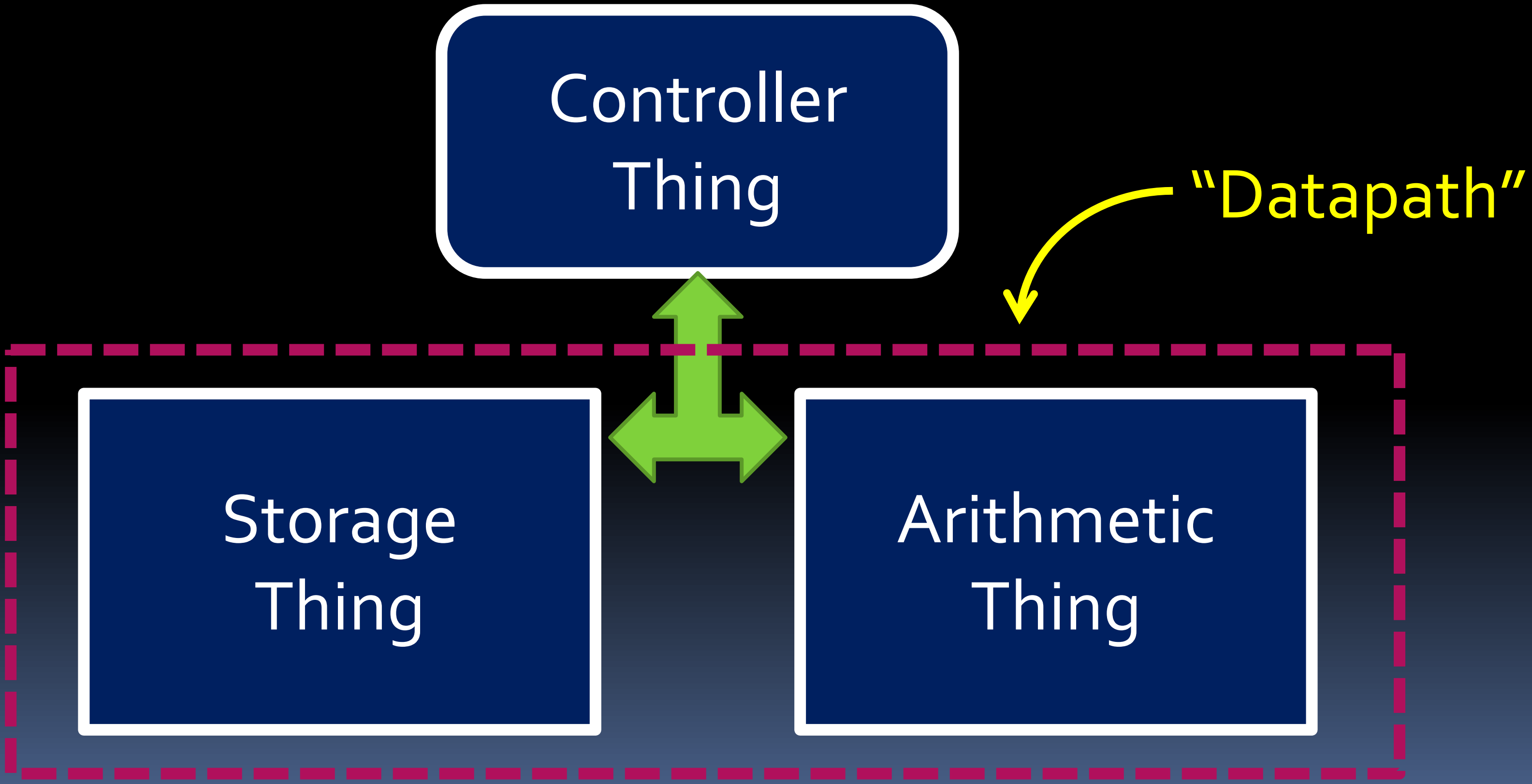


The Final Destination



Deconstructing processors

- Simpler at a high level:



Controller Thing

"Datapath"

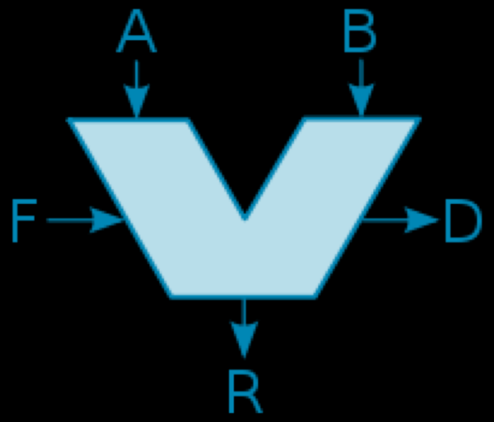
Storage Thing

Arithmetic Thing

The “Arithmetic Thing”

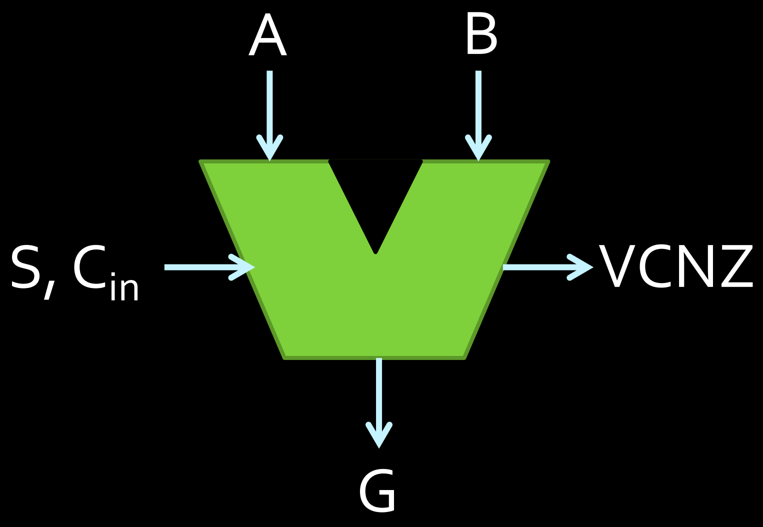
aka: the Arithmetic Logic Unit (ALU)

Arithmetic Logic Unit



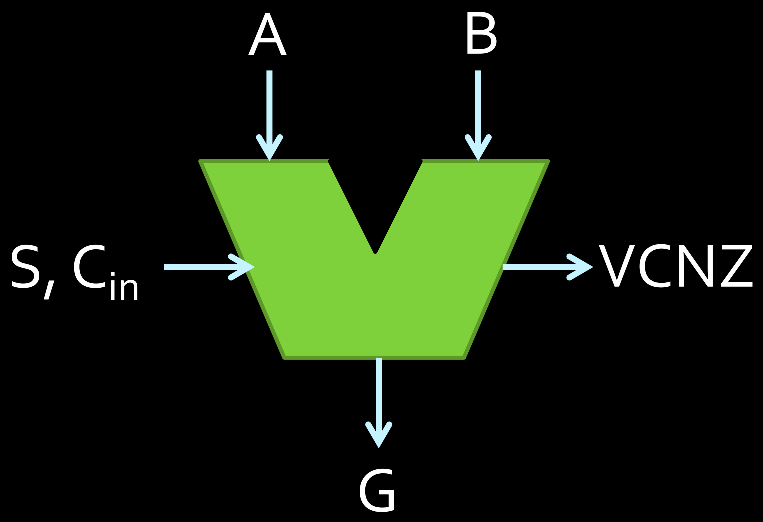
- The first microprocessor applications were calculators.
  - Remember adders and subtractors?
  - These are part of a larger structure called the arithmetic logic unit (ALU).
  - You made a simple one for a lab!
- This larger structure is responsible for the processing of all data values in a basic 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_0) that specify which operation to perform.
    - For example: S2 is a mode select bit, indicating whether the ALU is in arithmetic or logic mode
  - The carry-in bit C_{in} is used in operations such as incrementing an input value or the overall result.

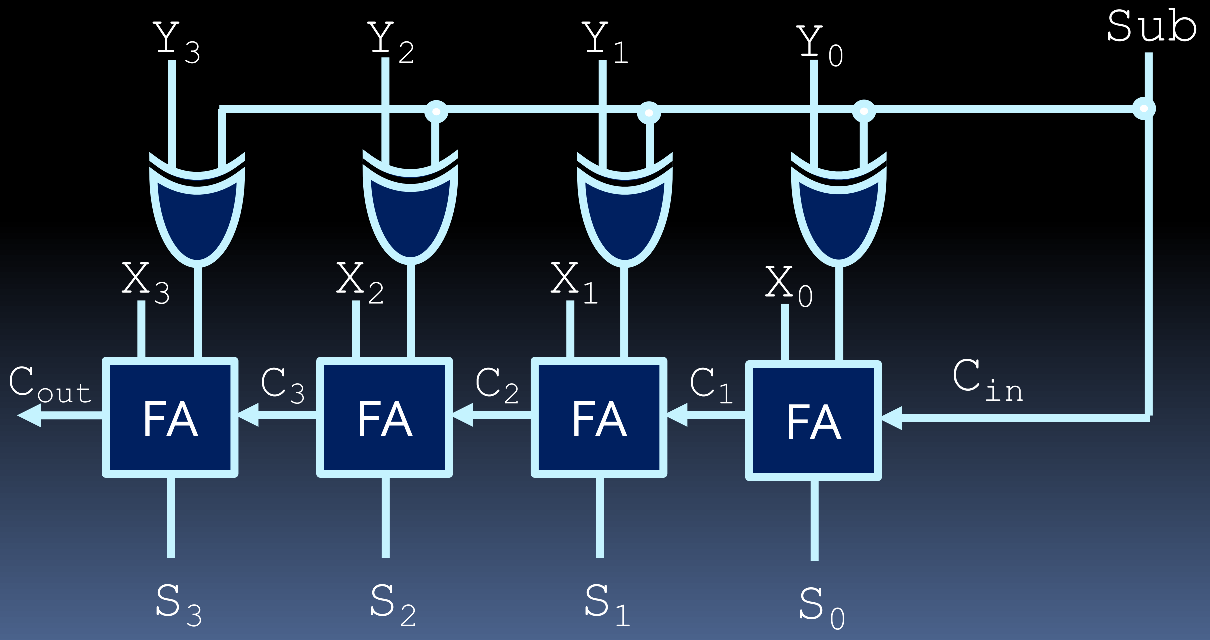
ALU outputs



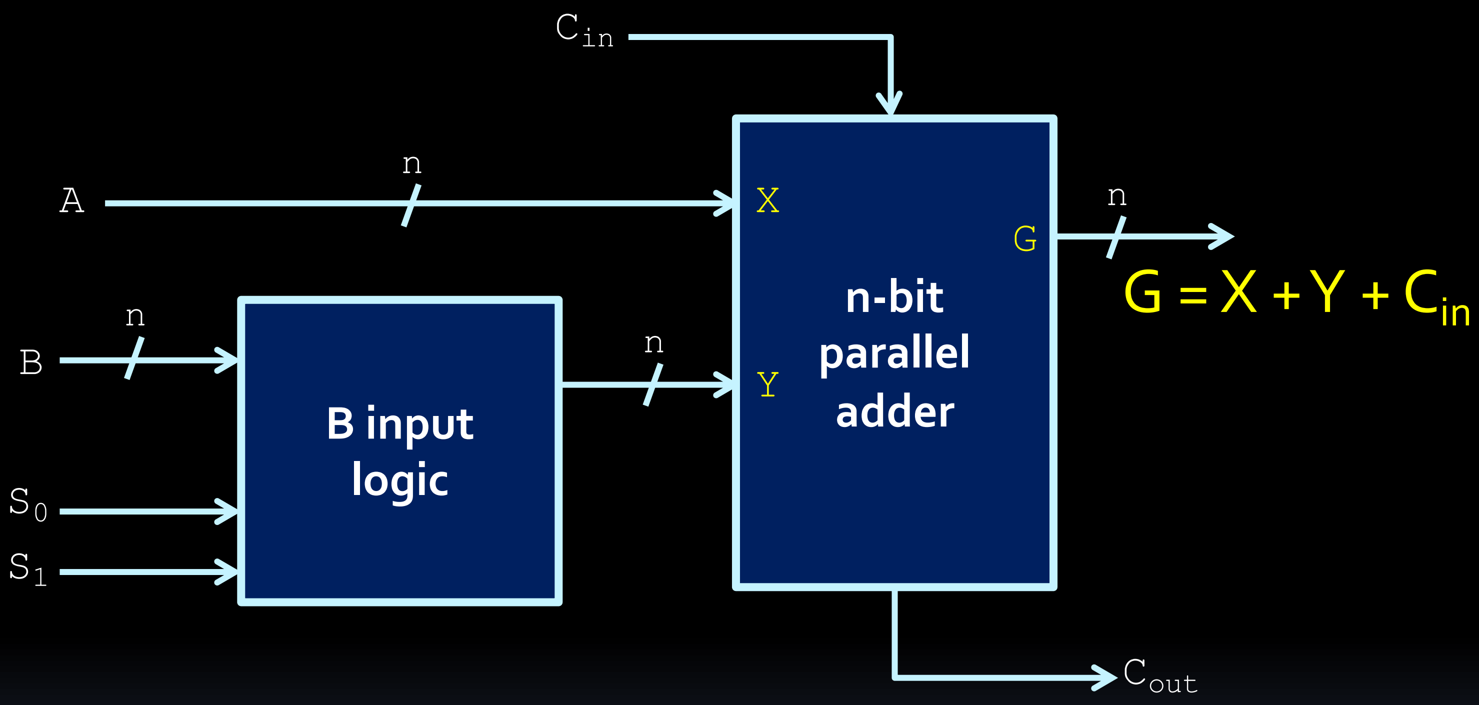
- In addition to the input signals, there are output signals V, C, N & Z which indicate special conditions in the arithmetic result:
  - V: overflow condition
    - The result of the operation could not be stored in the `n` bits of G, meaning that the result is incorrect.
  - C: carry-out bit
  - N: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

The “A” of ALU

- To understand how the ALU does all of these operations, let’s start with the arithmetic side.
- Fundamentally, this side is made of an adder / subtractor unit, which we’ve seen already:



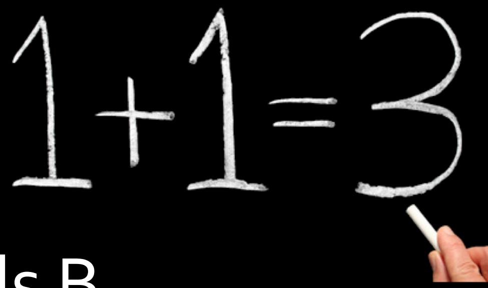
Arithmetic components



$G = X + Y + C_{in}$

- 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 $\overline{B}$?
  - Result of addition operation: $G = A-B-1$
- 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 $G = A + Y$

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-in 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.

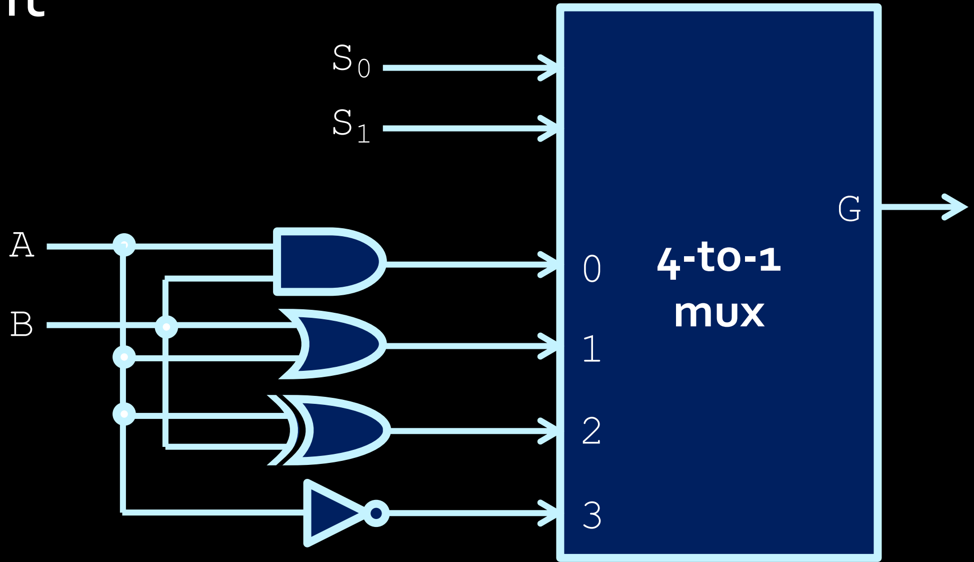
Full operation selection

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`.

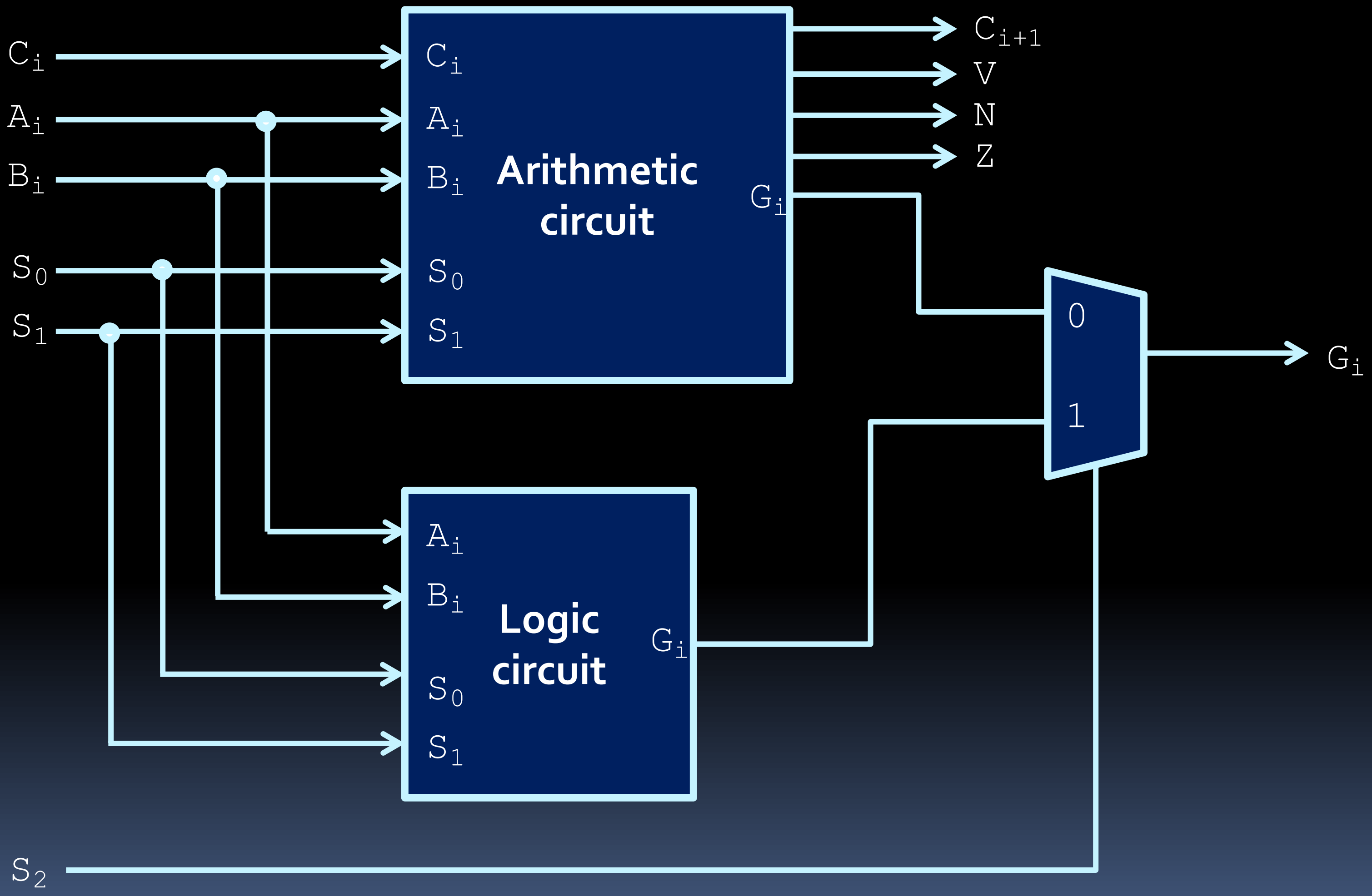
The “L” of ALU

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



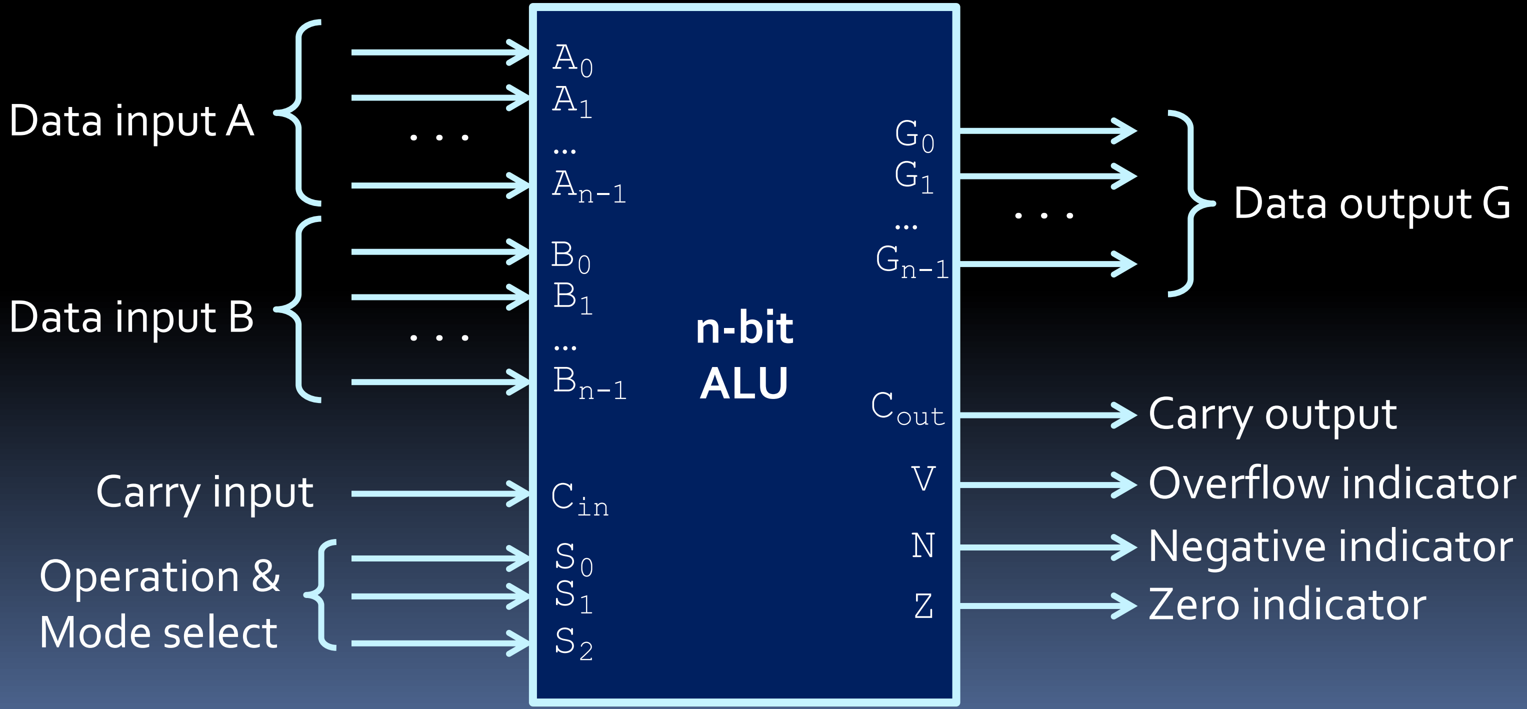
- If $S_2 = 1$, then logic circuit block is activated.
- Multiplexer is used to determine which block (logical or arithmetic) goes to the output.

Single ALU Stage



ALU block diagram

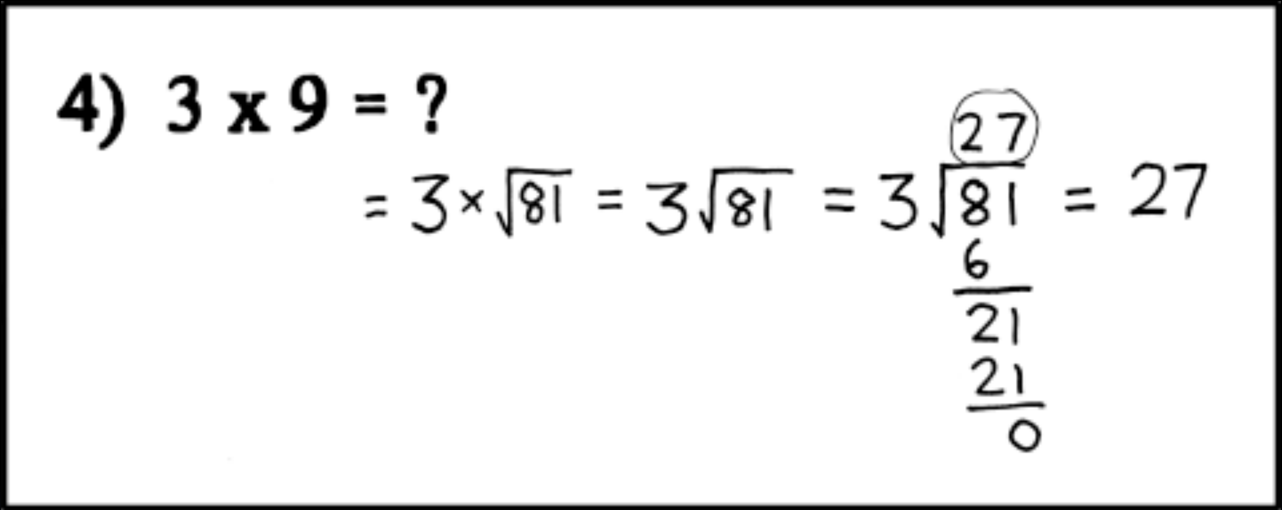
- In addition to data inputs and outputs, this circuit also has:
  - outputs indicating the different conditions,
  - inputs specifying the operation to perform (similar to `Sub`).



What about multiplication?

- Multiplication (and division) operations are 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 with accumulator.
  - Booth’s Algorithm

Break

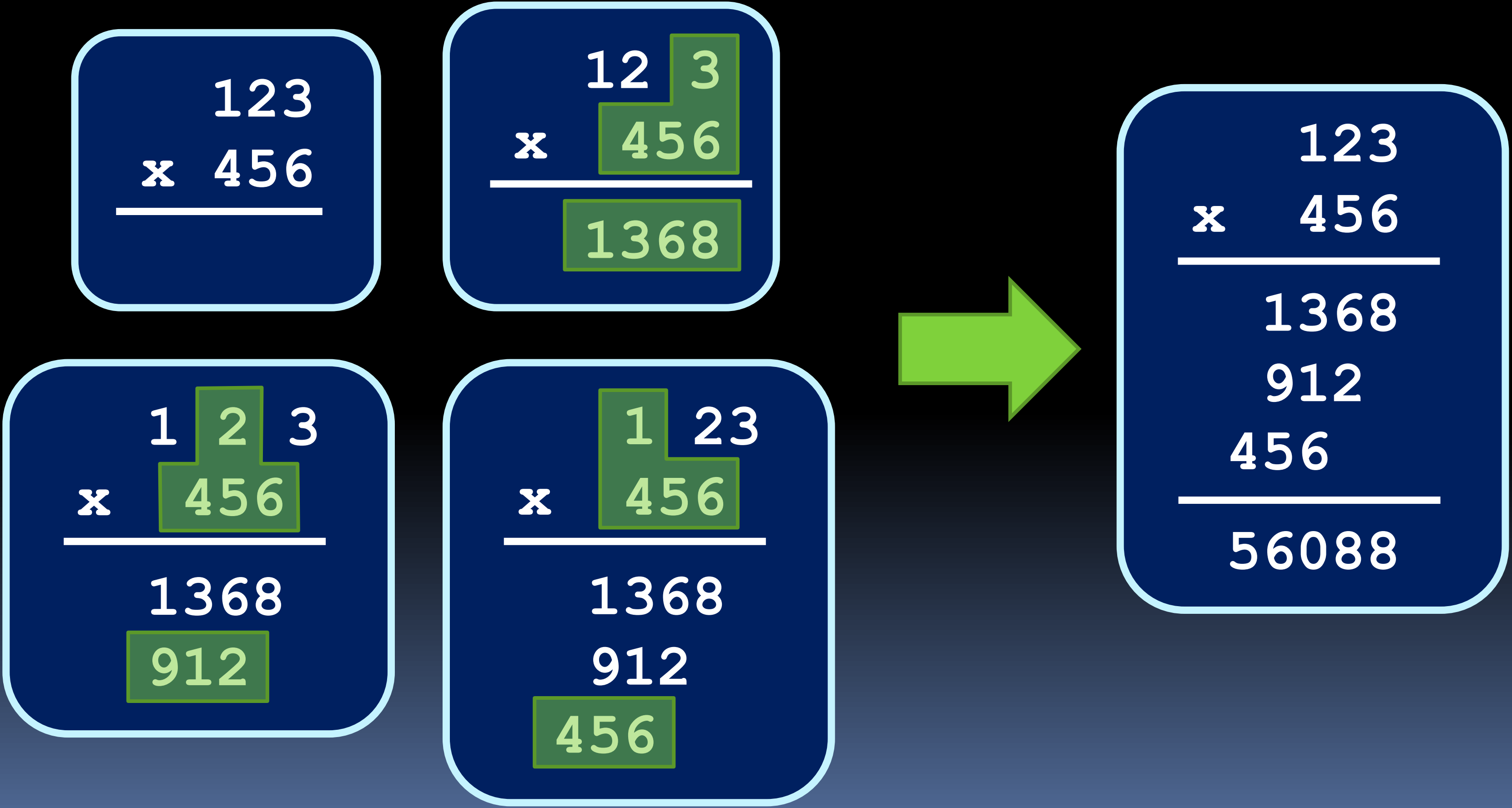


4) 3 x 9 = ?

$$= 3 \times \sqrt{81} = 3\sqrt{81} = \begin{array}{r} \boxed{27} \\ 3\overline{)81} \\ \underline{6} \\ 21 \\ \underline{21} \\ 0 \end{array} = 27$$

Multiplication

- Revisiting grade 3 math…



123
x 456

12 3
x 456
1368

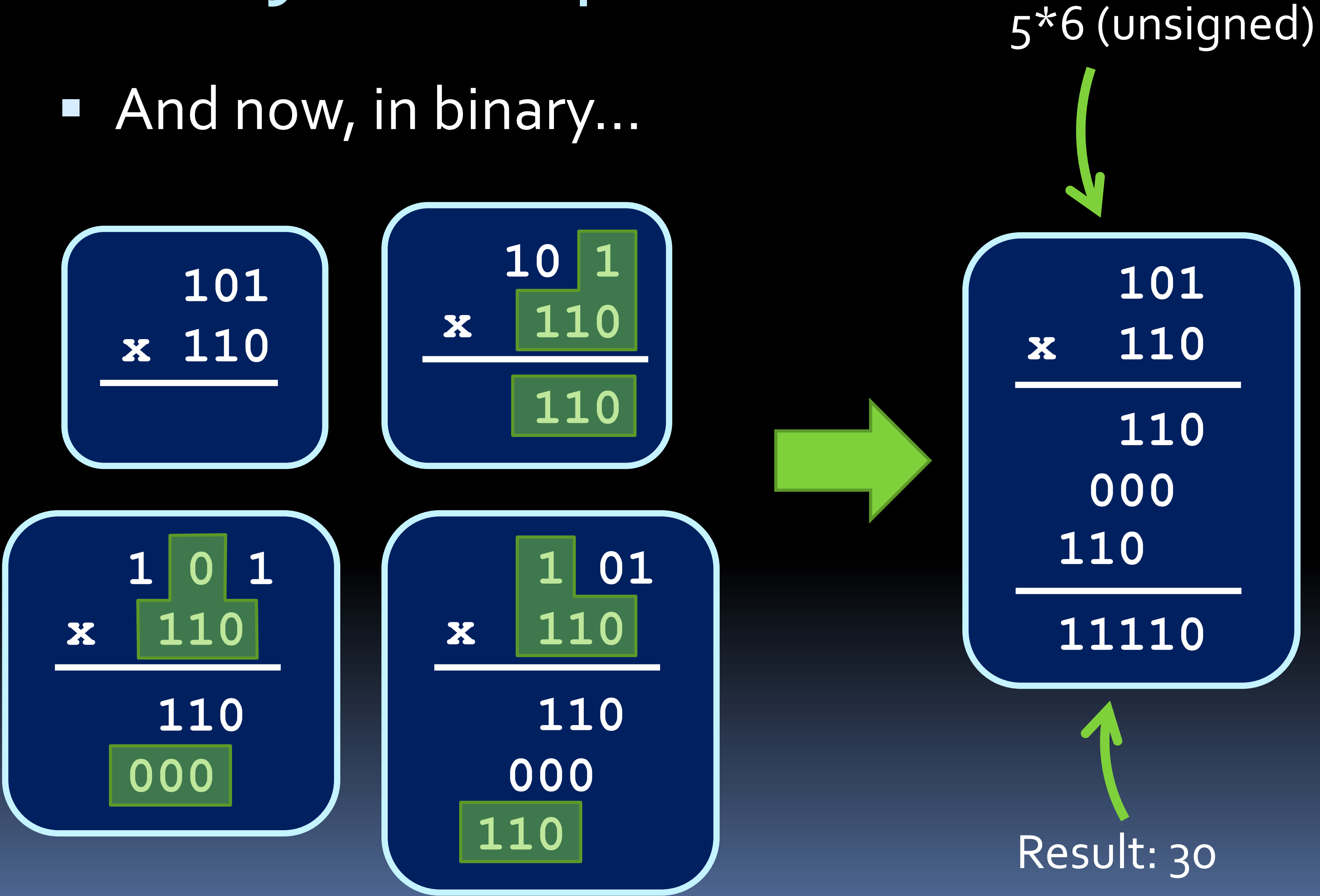
1 2 3
x 456
1368
912

1 23
x 456
1368
912
456

123
x 456
1368
912
456
56088

Binary Multiplication

- And now, in binary…



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

```
     10 1
  x   110
  -------
      110
```

```
   1 0 1
  x  110
  ------
     110
    000
```

```
     1 01
  x   110
  -------
      110
     000
    110
```

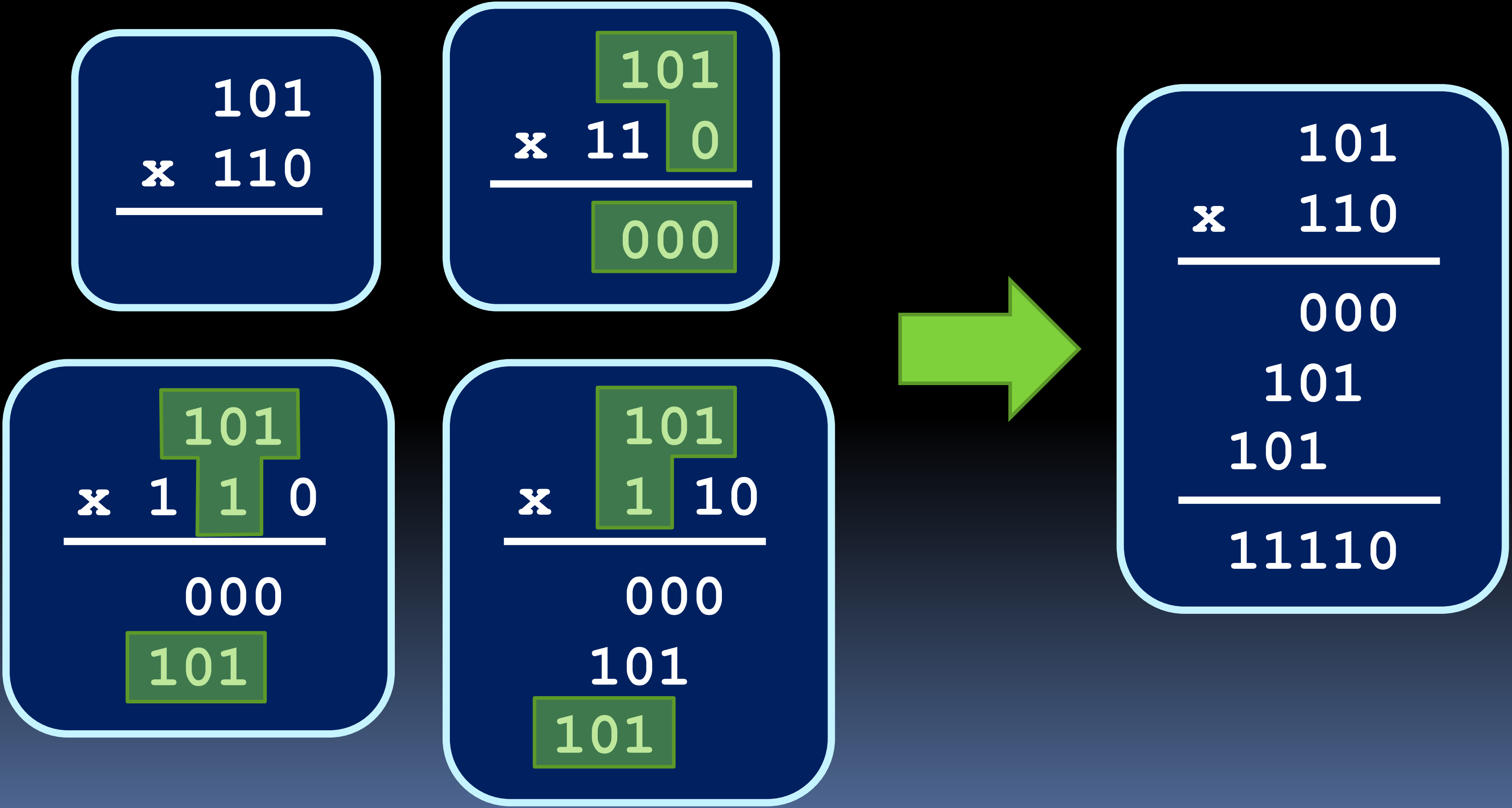
5*6 (unsigned)

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

Result: 30

Binary Multiplication

- Or seen another way….



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

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

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

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

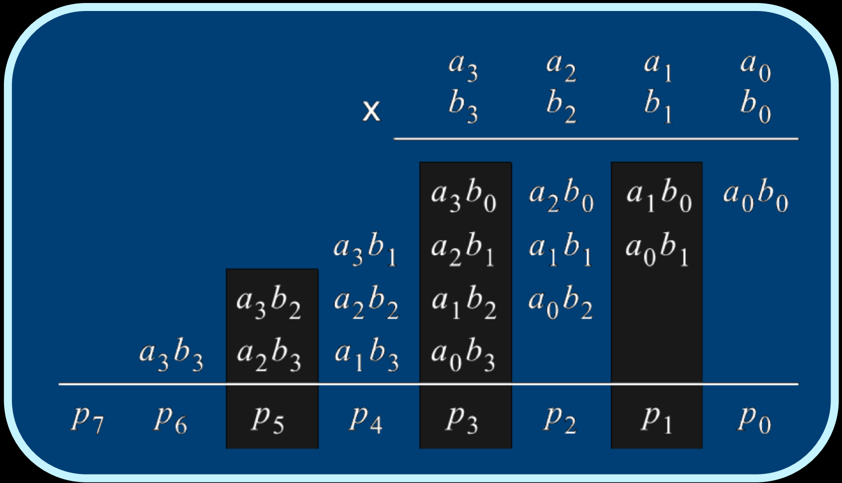
```
    101
x   110
-------
    000
   101
  101
-------
  11110
```

Binary Multiplication

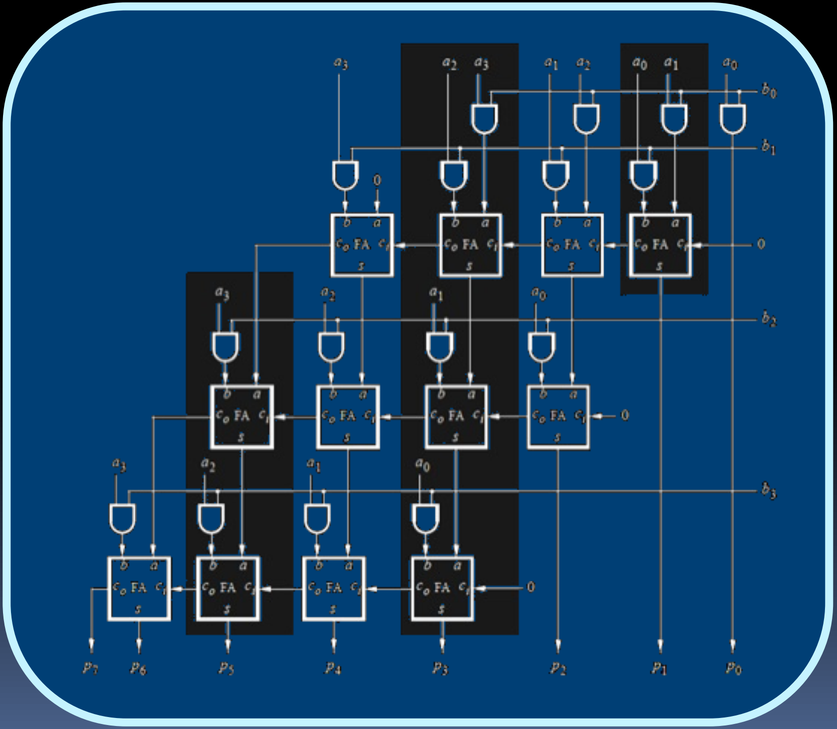
$$
\begin{array}{cccccccc}
 & & & & a_3 & a_2 & a_1 & a_0 \\
 & & & \times & b_3 & b_2 & b_1 & b_0 \\
\hline
 & & & & a_3b_0 & a_2b_0 & a_1b_0 & a_0b_0 \\
 & & & a_3b_1 & a_2b_1 & a_1b_1 & a_0b_1 & \\
 & & a_3b_2 & a_2b_2 & a_1b_2 & a_0b_2 & & \\
 & a_3b_3 & a_2b_3 & a_1b_3 & a_0b_3 & & & \\
\hline
p_7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

Implementation

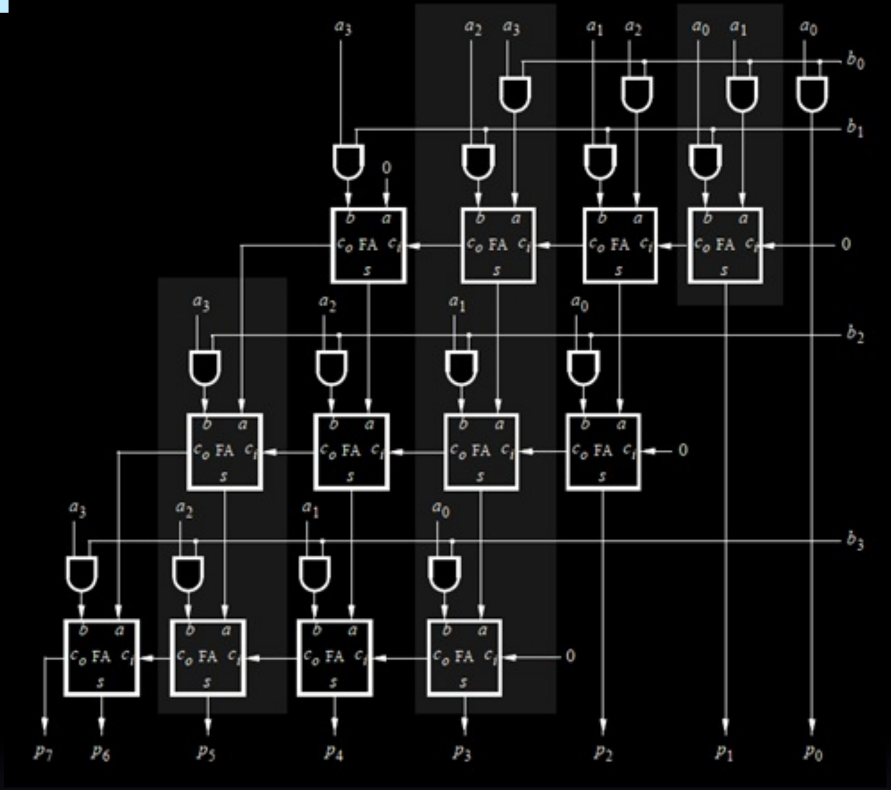
- Implementing this in circuitry involves the summation of several AND terms.
    - AND gates combine input signals.
    - Adders combine the outputs of the AND gates.



				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



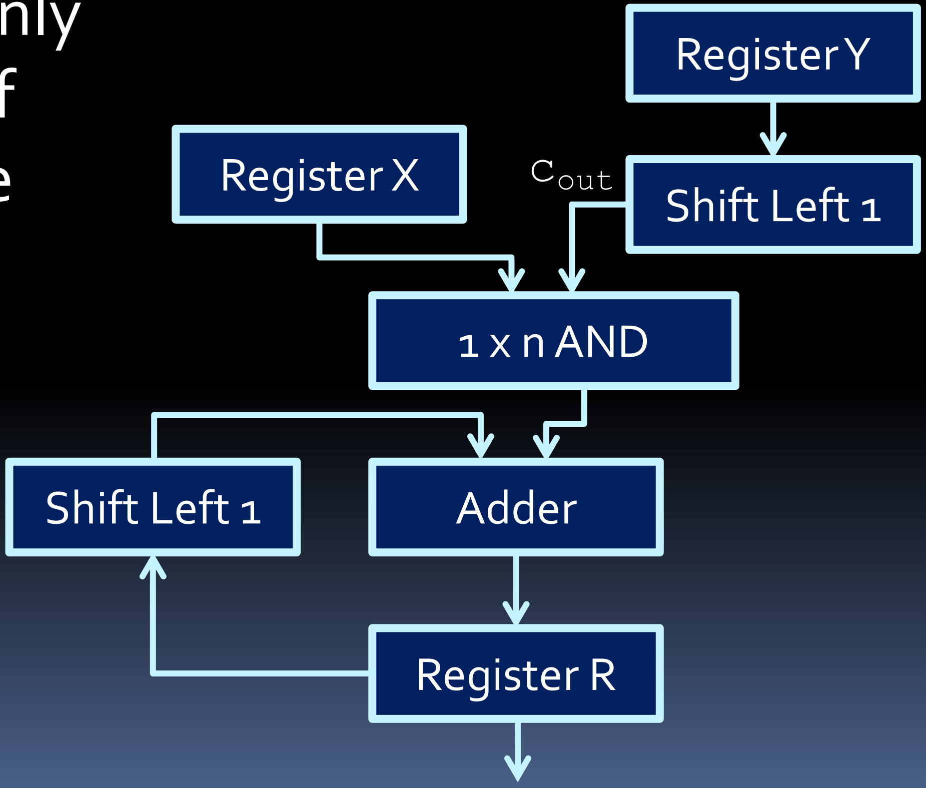
Multiplication



- 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) clock cycles, but O(N²) 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?



Sign Extension

- To subtract 4-bit number from 8-bit number….
- How do we convert a 4-bit two's complement number to 8-bit?
- Sign extend: replicate most significant bit

0101 → 0000 0101

(5) (still 5)

1001 → 1111 1001

(-7) (still -7)

- Arithmetic shift right: shift right and replicate sign bit (you saw this in lab!)

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 Example in Decimal

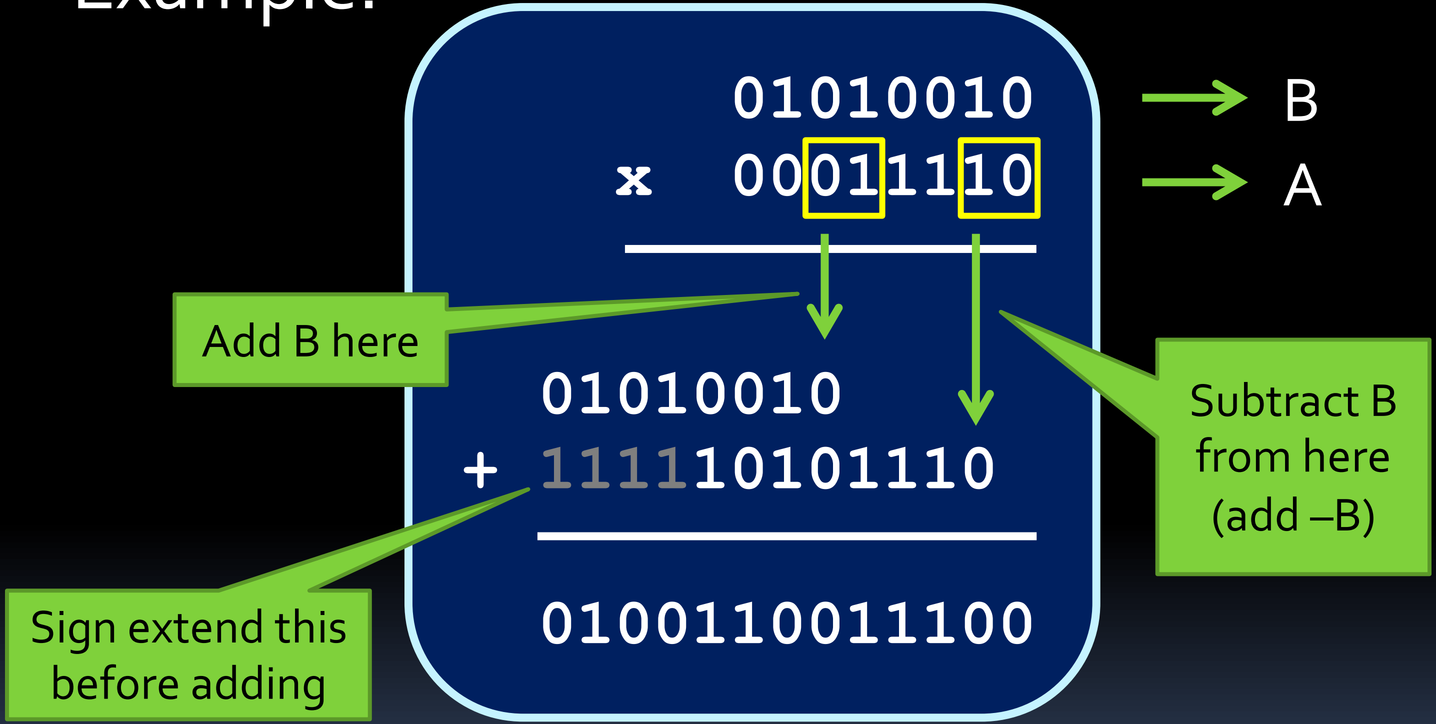
- Compute 999 x 5 →
    - 1000 x 5 – 1 x 5 → 5,000 – 5 = 4,995
- Compute 99,900 x 5 →
    - 100,000 x 5 – 100 x 5 = 500,000 – 500 = 499,500
- Compute 999,099 x 5 →
    - 1,000,000 x 5 – 1,000 x 5 → 5,000,000 – 5,000 = 4,995,000
    - 100 x 5 – 1 x 5 → 500 – 5 = 495
    - 4,995,000 + 495 = **4,995,495**

Booth's Algorithm

- This idea is triggered on cases where two neighboring digits in an operand are different.
- Go through digits from n-1 to 0
  - 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 (add –B)

Sign extend this before adding

Booth's Algorithm

- We need to make this work in hardware.
  - Option #1: Have hardware set up to compare neighbouring bits at every position in `A`, with adders in place for when the bits don't match.
  - Problem: This is a lot of hardware, which Booth's Algorithm is trying to avoid.
  - Option #2: Have hardware set up to compare two neighbouring bits, and have them move down through `A`, looking for mismatched pairs.
  - Problem: Hardware doesn't move like that. Oops.

Booth's Algorithm

- Still need to make this work in hardware…
  - Option #3: Have hardware set up to compare two neighbouring bits in the lowest position of `A`, and looking for mismatched pairs in `A` by shifting `A` to the right one bit at a time.
  - Solution! This could work, but the accumulated solution `P` would have to shift one bit at a time as well, so that when `B` is added or subtracted, it's from the correct position.

Booth's Algorithm

Note: unlike the accumulator, the bits here are being shifted to the right!

- Steps in Booth's Algorithm:
  1. Designate the two multiplicands as A & B, and the result as some product P.
  2. Add an extra zero bit to the right-most side of A.
  3. Repeat the following for each original bit in A:
     a) If the last two bits of A are the same, do nothing.
     b) If the last two bits of A are `01`, then add B to the highest bits of P.
     c) If the last two bits of A are `10`, then subtract B from the highest bits of P.
     d) Perform one-digit arithmetic right-shift on both P and A.
  4. The result in P is the product of A and B.

Booth's Algorithm Example

- Example: `(-5) * 2`

- Steps #1 & #2:
  - A = -5 → 11011
    - Add extra zero to the right → A = 11011 0
  - B = 2 → 00010
  - -B = -2 → 11110
  - P = 0 → 00000 00000

Booth's Algorithm Example

```
A =  11011 0
P =  00000 00000
```

- Step #3 (repeat 5 times):
    - Check last two digits of A:

      1101 10

    - Since digits are 10, subtract B from the most significant digits of P:

      ```
       P     00000 00000
      -B    +11110
      ------------------
       P'    11110 00000
      ```

    - Arithmetic shift P and A one bit to the right:
        - A = 111011 P = 11111 00000

Booth's Algorithm Example

- Step #3 (repeat 4 more times):
  - Check last two digits of A:

    1110 11

  - Since digits are 11, do nothing to P.
  - Arithmetic shift P and A one bit to the right:
    - A = 111101 P = 11111 10000

A = 11101 1

P = 11111 00000

Booth’s Algorithm Example

```
A =  11110 1
P =  11111 10000
```

- Step #3 (repeat 3 more times):
  - Check last two digits of A:

    1111 01

  - Since digits are 01, add B to the most significant digits of P:

```
  P      11111 10000
 +B     +00010
 ____________________
  P’     00001 10000
```

  - Arithmetic shift P and A one bit to the right:
    - A = 111110 P = 00000 11000

Booth’s Algorithm Example

```
A =  11111 0
P =  00000 11000
```

- Step #3 (repeat 2 more times):
  - Check last two digits of A:

```
1111 10
```

  - Since digits are 10, subtract B from the most significant digits of P:

```
 P     00000 11000
-B    +11110
------------------
 P’    11110 11000
```

  - Arithmetic shift P and A one bit to the right:
    - A = 111111   P = 11111 01100

Booth’s Algorithm Example

A = 11111 1
P = 11111 01100

- Step #3 (final time):
  - Check last two digits of A:

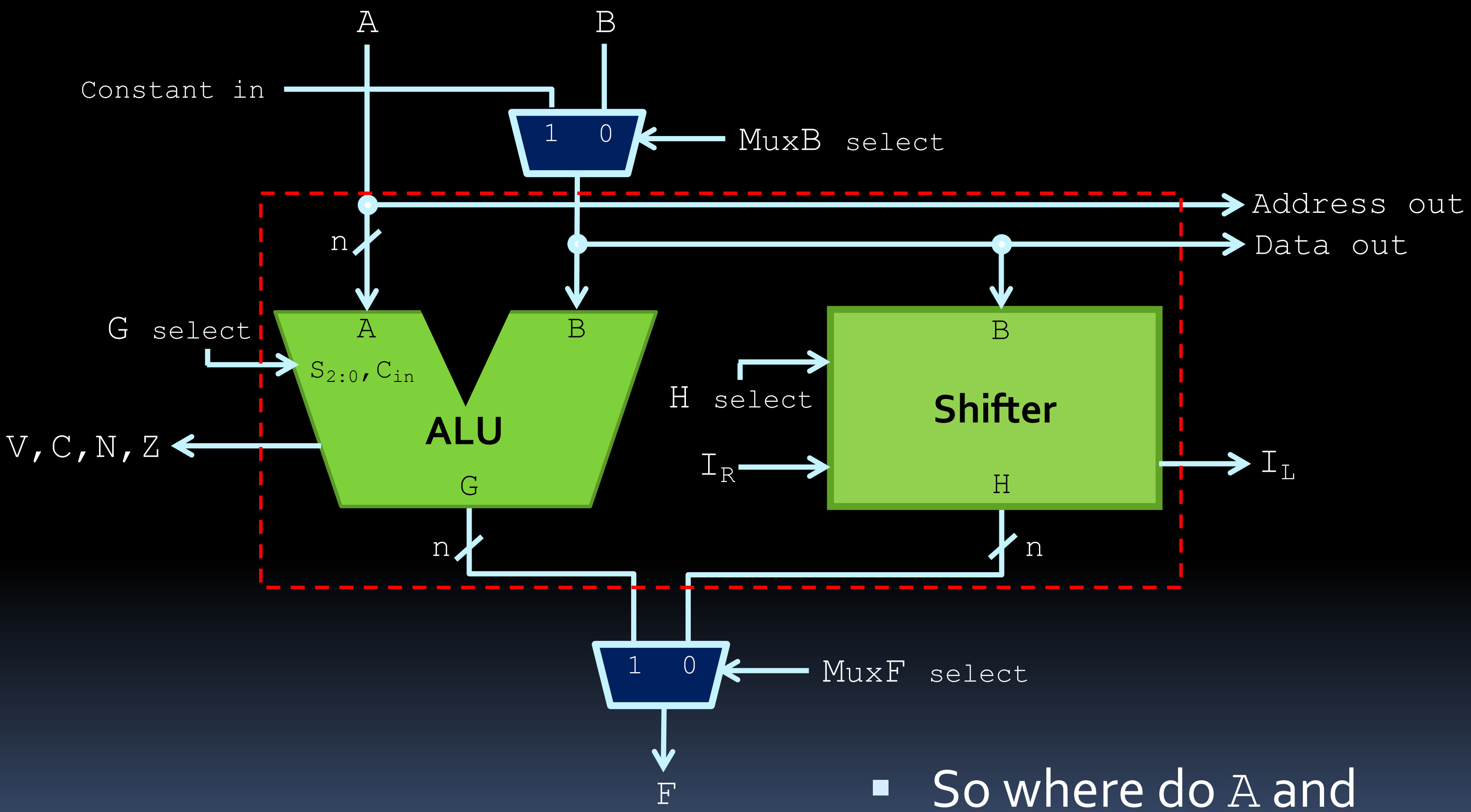
    1111 11
  - Since digits are 11, do nothing to P:
  - Arithmetic shift P and A one bit to the right:
    - A = 111111 P = 11111 10110
- Final product:

P = 111110110
= -10

Reflections on multiplication

- A popular version of this algorithm involves copying A into the lower bits of P, so that the testing and shifting only takes place in P.
    - Also good for maintaining the original value of A.
- 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, the shift register is used instead of the multiplier circuit.

Function Unit



- So where do A and B come from?

The “Storage Thing”

aka: the register file and main memory

More on this next time