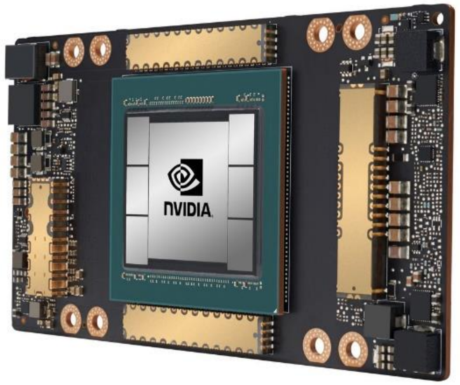
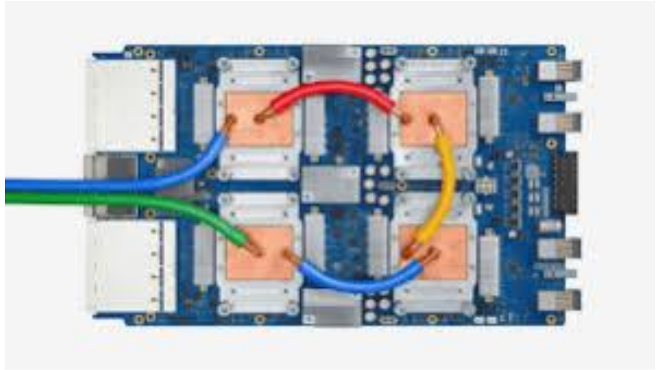
CSCB58:
Computer Organization



Prof. Gennady Pekhimenko

University of Toronto

Fall 2020



The content of this lecture is adapted from the lectures of Larry Zheng and Steve Engels

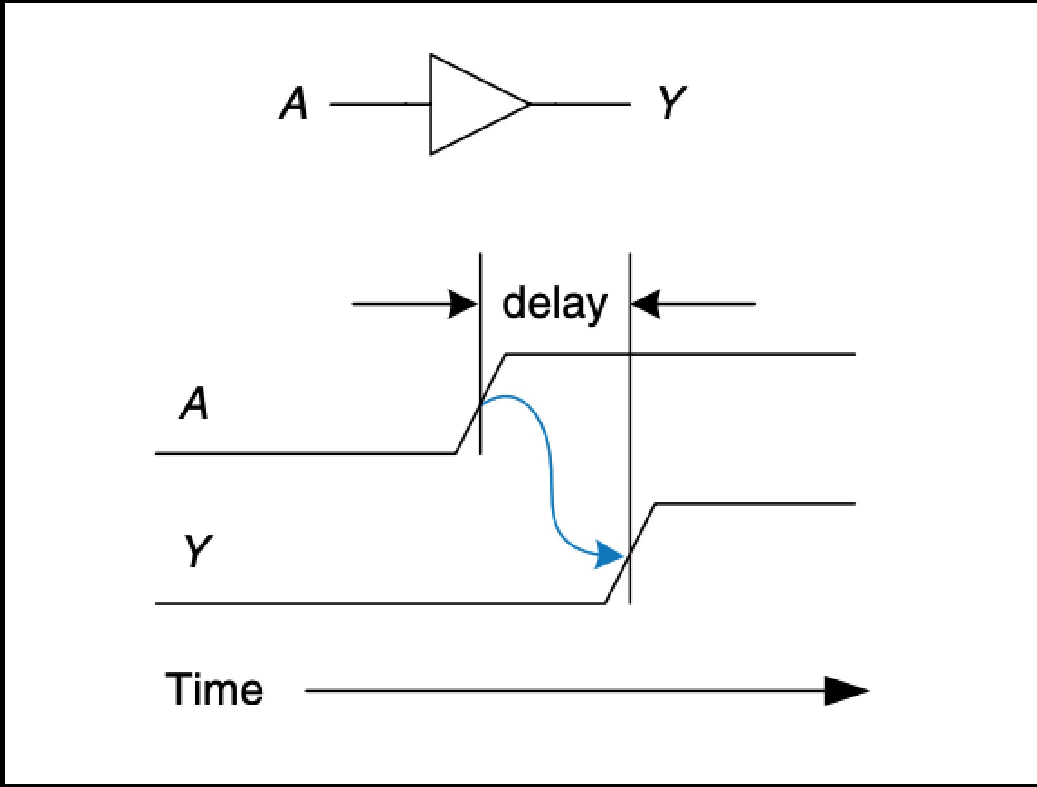
CSCB58 Week 7

Circuit Timing

Timing

- So far we have been worrying whether a circuit is correct.
- Now let's think about how to make a circuit fast.
- Key concept: latency
  - propagation delay
  - contamination delay

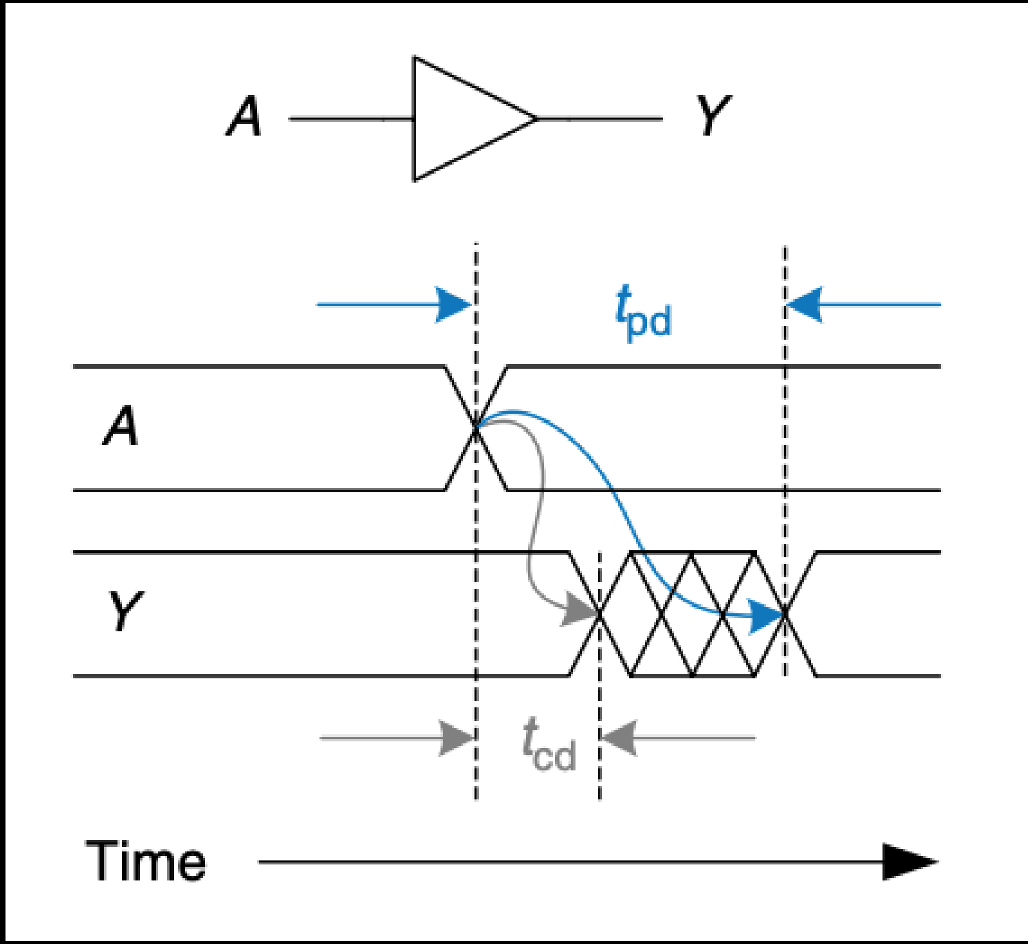
Delay Example



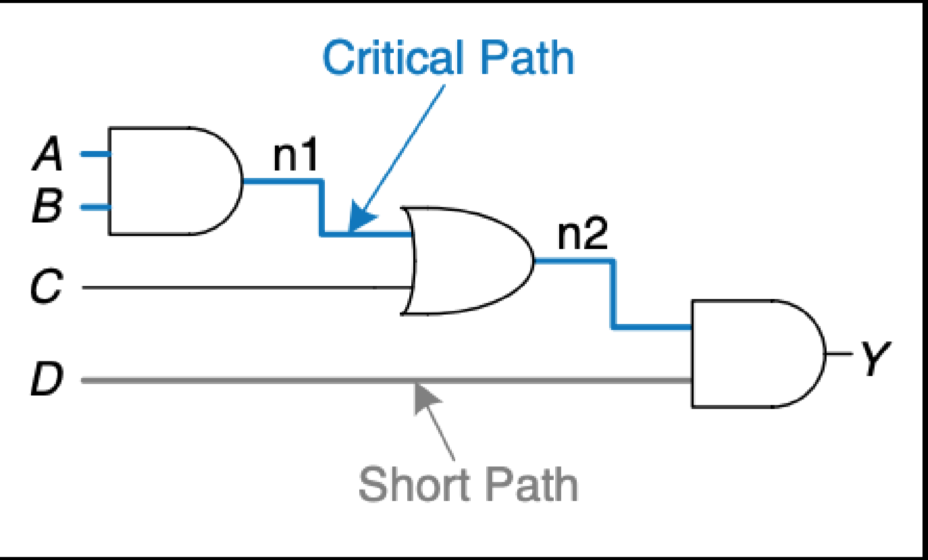
- We measure the interval between the two "50% points" of the changing signals

Propagation & Contamination Delay

- **Propagation delay**: the maximum time from when an input changes until the output or outputs reach their final value.
- **Contamination delay**: the minimum time from when an input changes until any output starts to change its value.

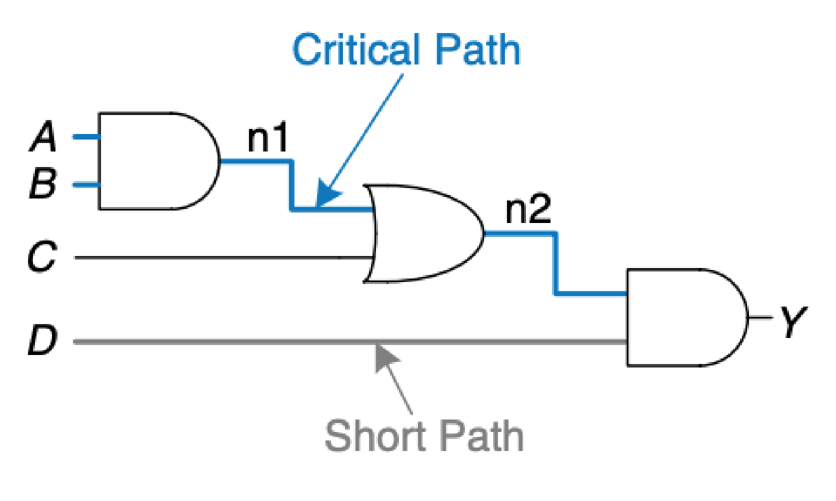


Given a circuit diagram, calculate its propagation delay and contamination delay



Need to know

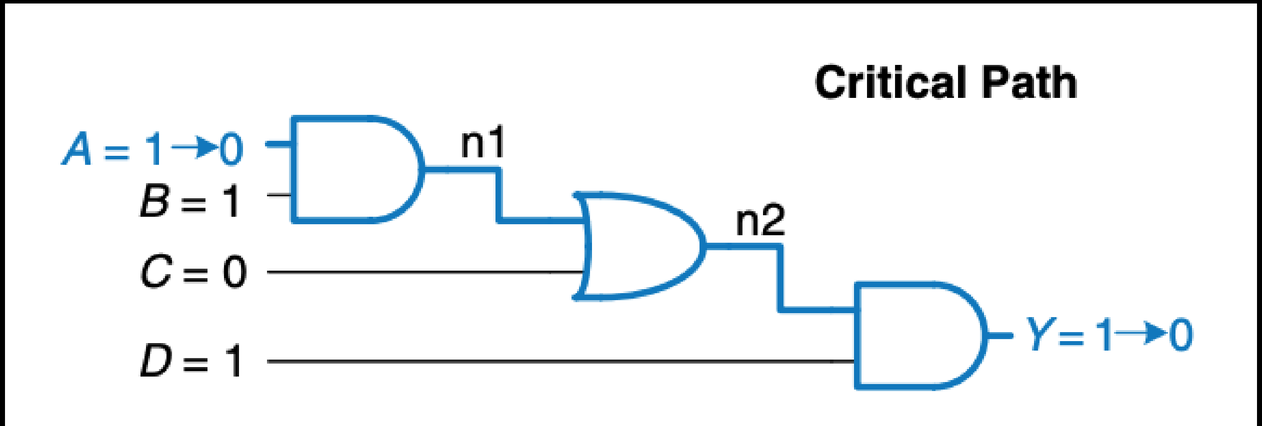
- The propagation and contamination delay of each logic gate used



Gate	t_pd (propagation)	t_cd (contamination)
2-input AND	100 picoseconds	60 picoseconds
2-input OR	120 picoseconds	40 picoseconds

Calculate Propagation Delay

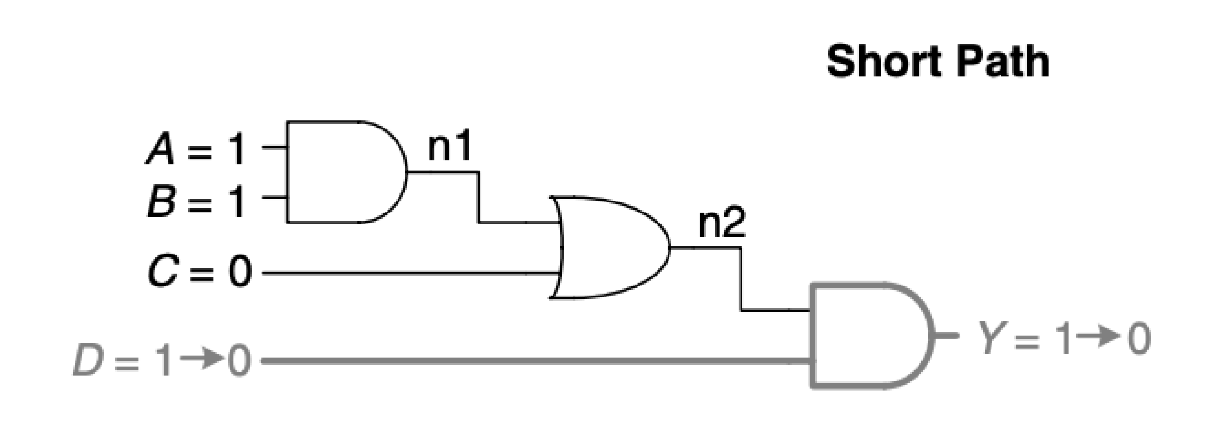
- Find the **critical path** (path with the largest number of gates)
- then sum up the **propagation** delay of all the gates on the critical path
- 100 + 120 + 100 = 320 picoseconds



Gate	t_pd	t_cd
2-input AND	100 ps	60 ps
2-input OR	120 ps	40 ps

Calculate Contamination Delay

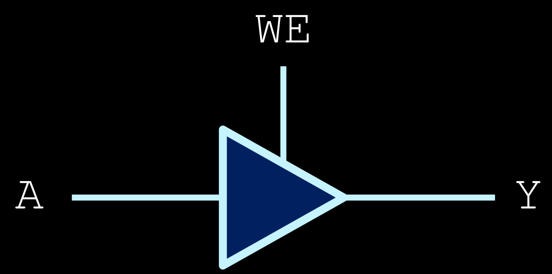
- Find the **short path** (path with the smallest number of gates)
- then sum up the **contamination** delay of all the gates on the short path
- 60 picoseconds



Gate	t_pd	t_cd
2-input AND	100 ps	60 ps
2-input OR	120 ps	40 ps

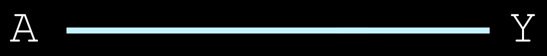
Knowing how to calculate delays allows us to design circuits that are fast.

Quick intro:
Tri-state buffer

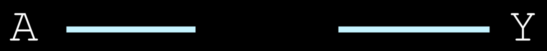


WE	A	Y
0	x	z
1	0	0
1	1	1

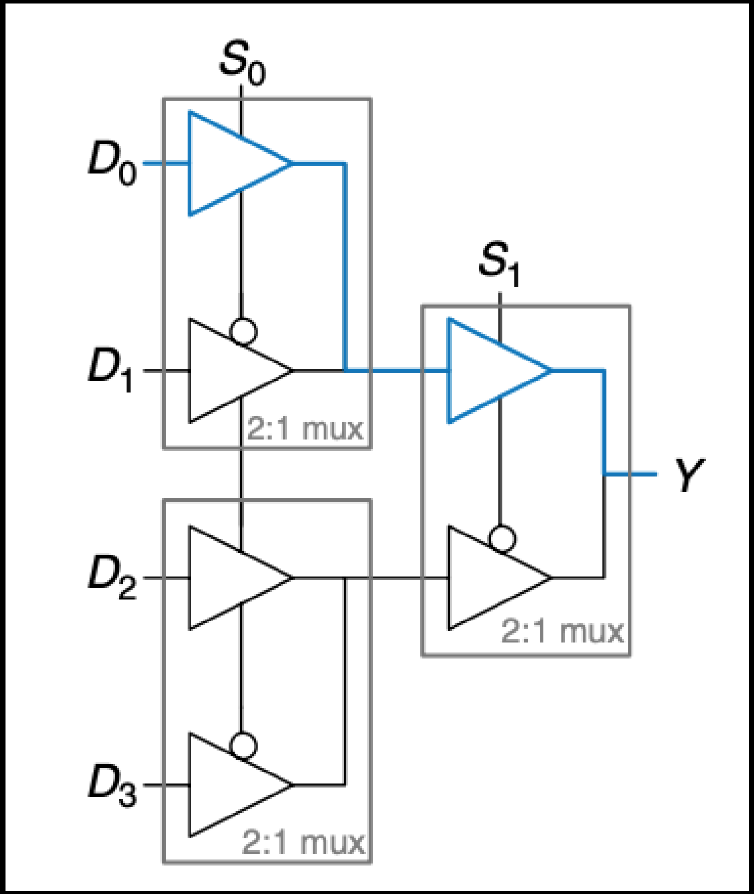
WE = 1

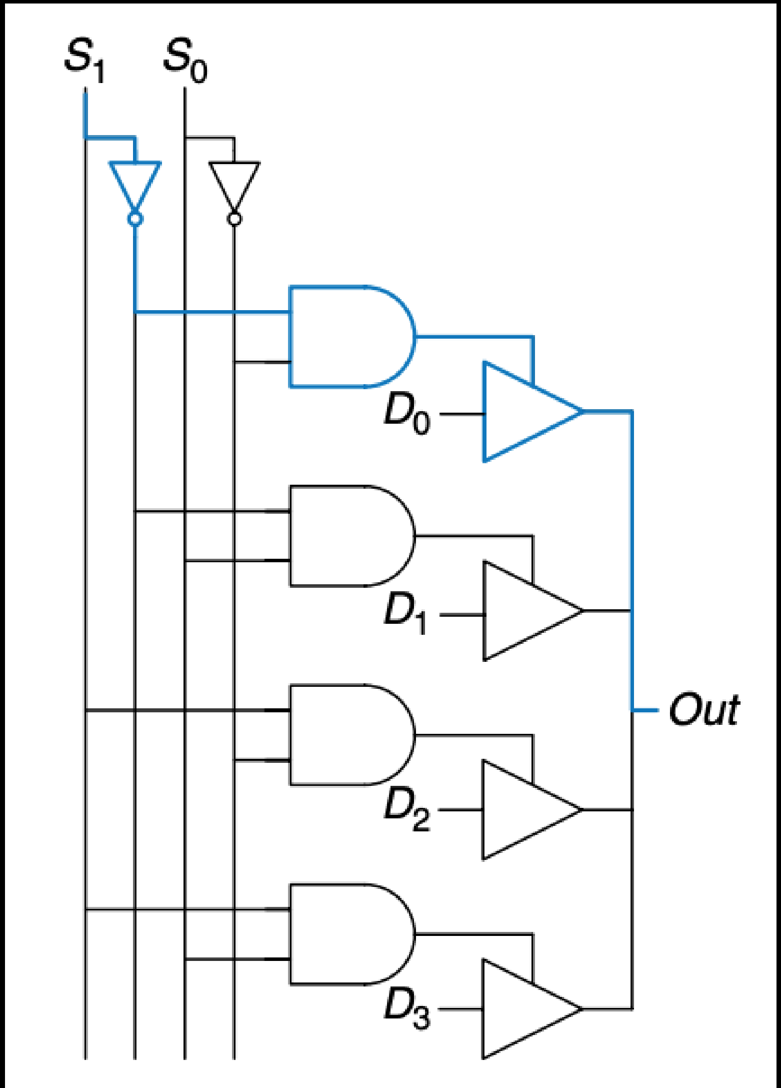


WE = 0



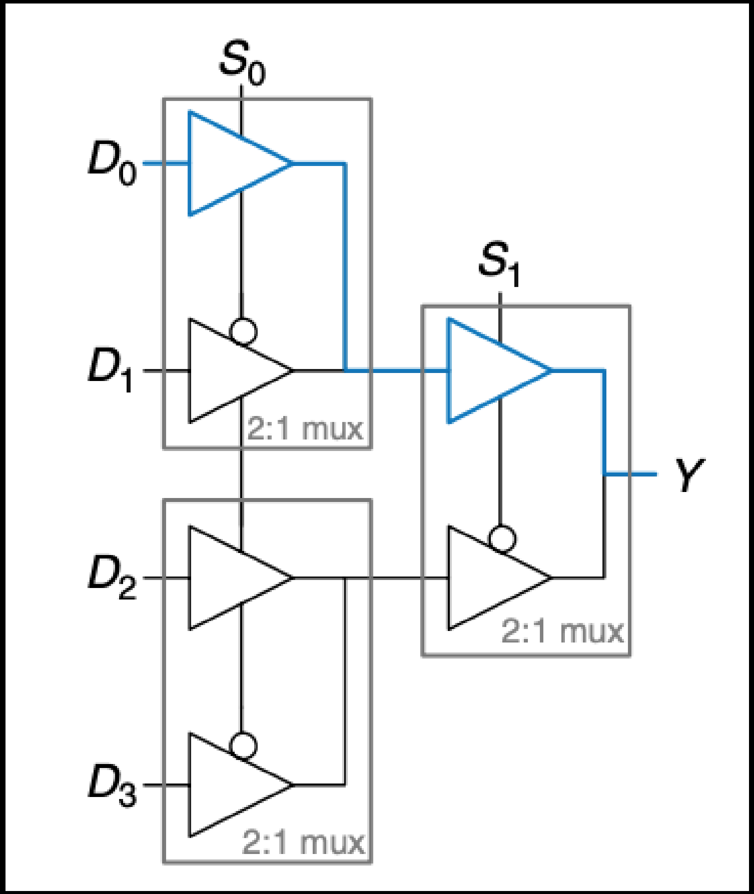
Example: design fast circuit

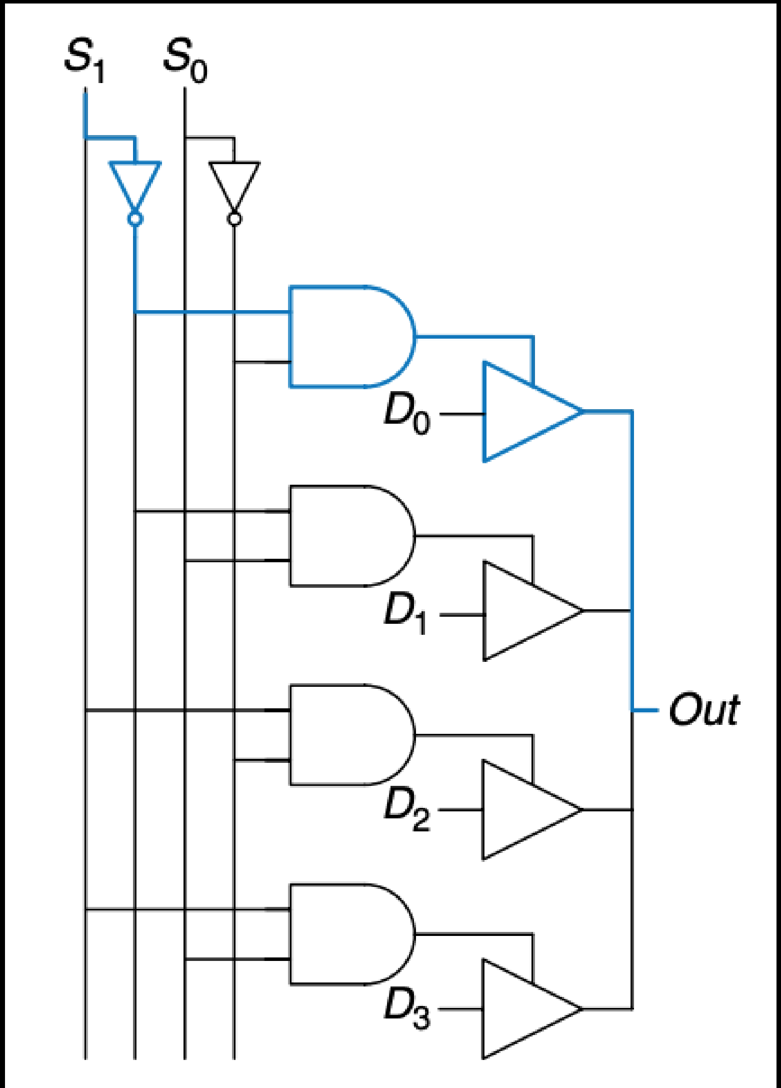




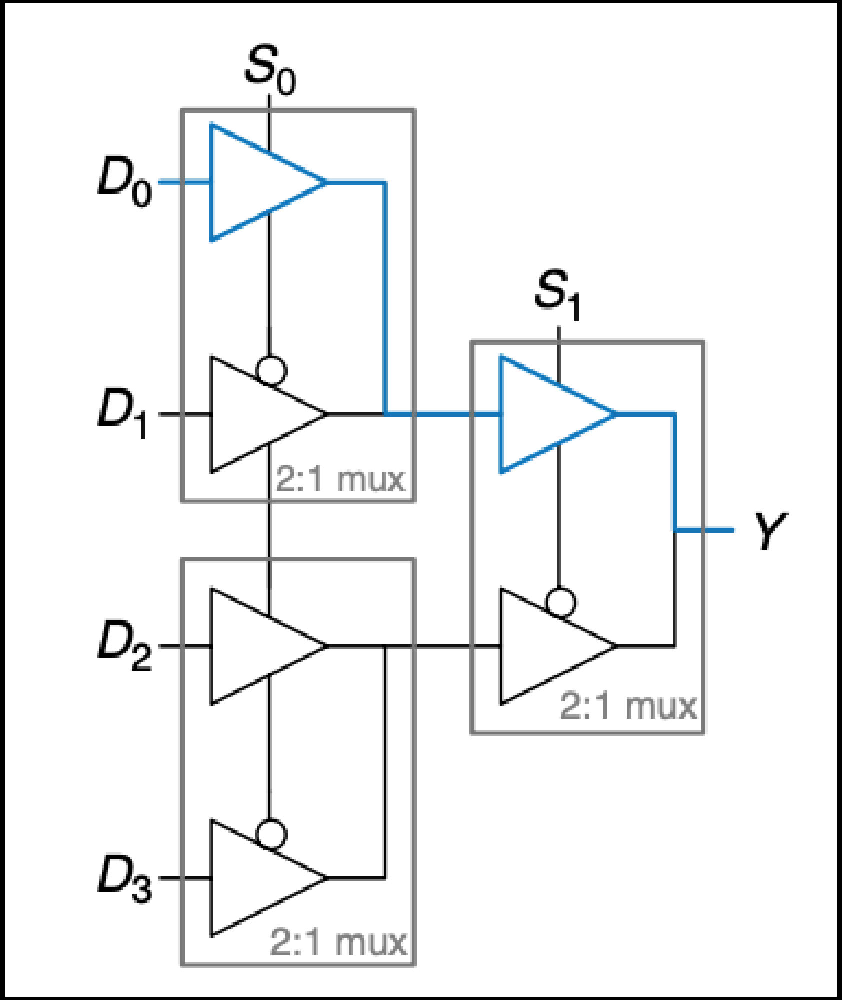
two different
4-1 muxes

Example: design fast circuit





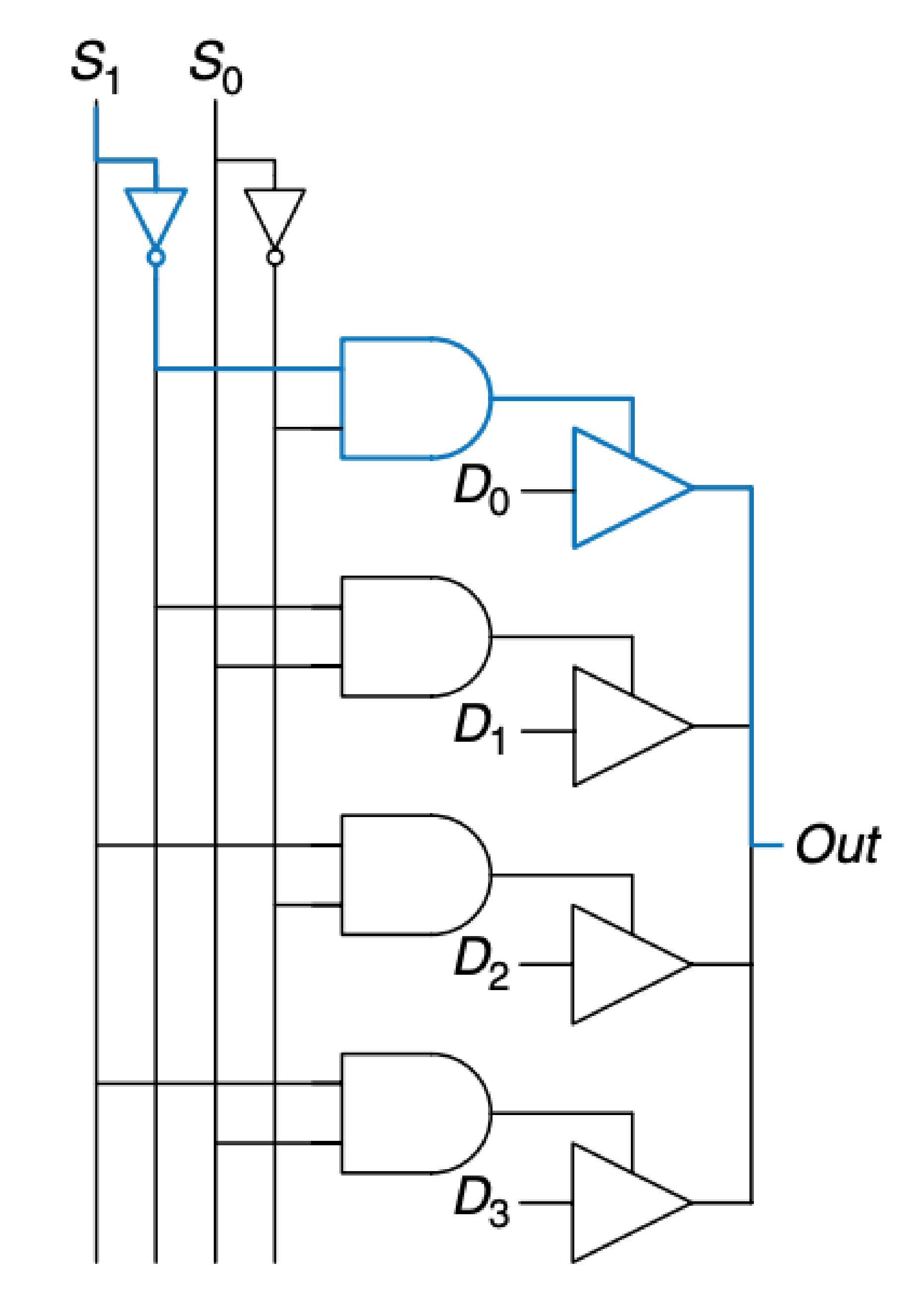
- We care about the **propagation** delays of the two circuits.
  - it tells us "how soon I can get the answer"
- More specifically, we care about the D-to-Y delay and S-to-Y delay because D and S may arrive at different time.



Gate	t_{pd} (ps)
NOT	30
2-input AND	60
3-input AND	80
4-input OR	90
tristate (A to Y)	50
tristate (enable to Y)	35

- D-to-Y propagation delay:
- 2 x TRISTATE_AY = 100

- S-to-Y propagation delay
- TRISTATE_ENY + TRISTATE_AY
- = 35 + 50 = 85



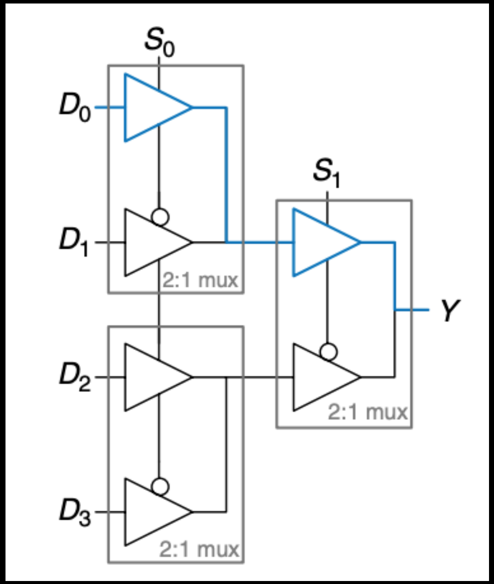
Gate	t_{pd} (ps)
NOT	30
2-input AND	60
3-input AND	80
4-input OR	90
tristate (*A* to *Y*)	50
tristate (enable to *Y*)	35

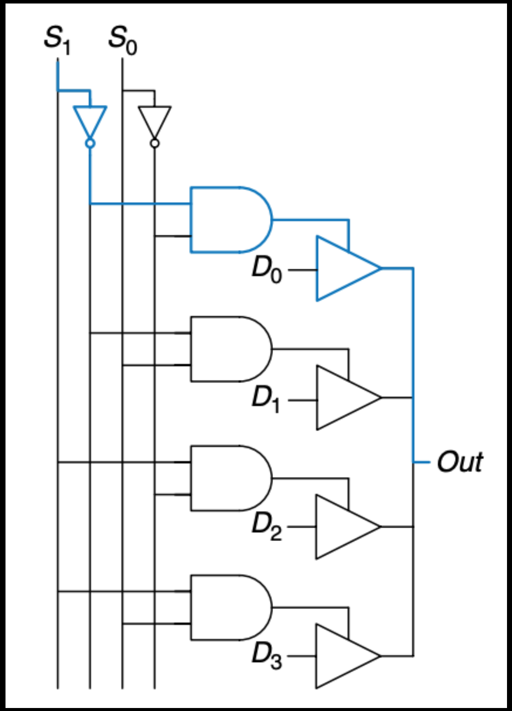
- D-to-Y propagation delay:
- TRISTATE_AY = 50

- S-to-Y propagation delay
- NOT + AND2 + TRISTATE_ENY
- = 30 + 60 + 35 = 125

Analysis result

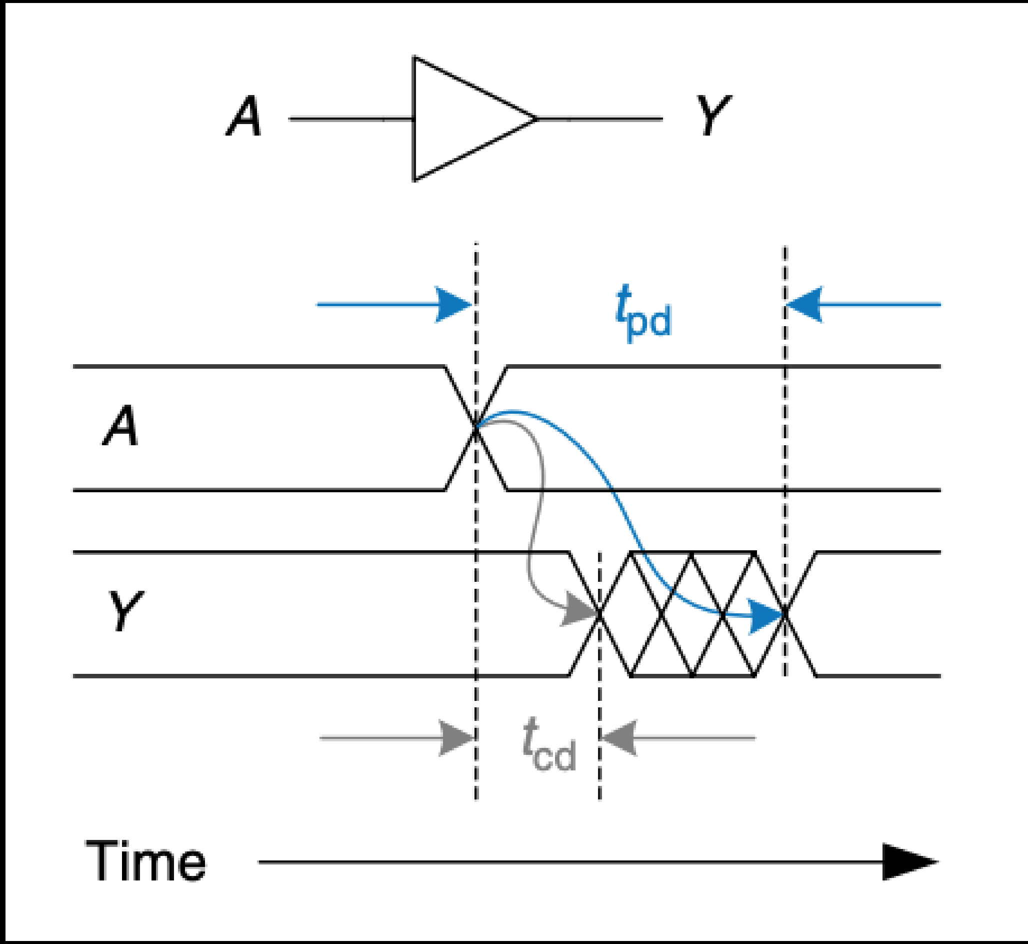
- Circuit 1 propagation:
  - D-to-Y: 100 ps
  - S-to-Y: 85 ps
- Circuit 2 propagation
  - D-to-Y: 50 ps
  - S-to-Y: 125 ps
- Which circuit is faster?
  - What if D and S arrive at the same time?
  - What if D arrives earlier than S?
  - What if S arrives earlier than D?

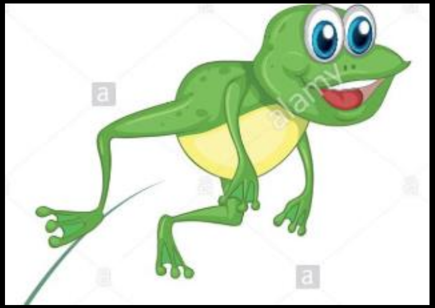




Delays: the lower/higher, the better?

- Propagation delay, typically, should be upper-bounded.
  - shorter propagation means getting answer faster
  - How to make it lower?
  - shorten the critical path
- Contamination delay, typically, should be lower-bounded
  - want to reliably sample the value before change.
  - How to make it longer?
  - add **buffers** to the short path



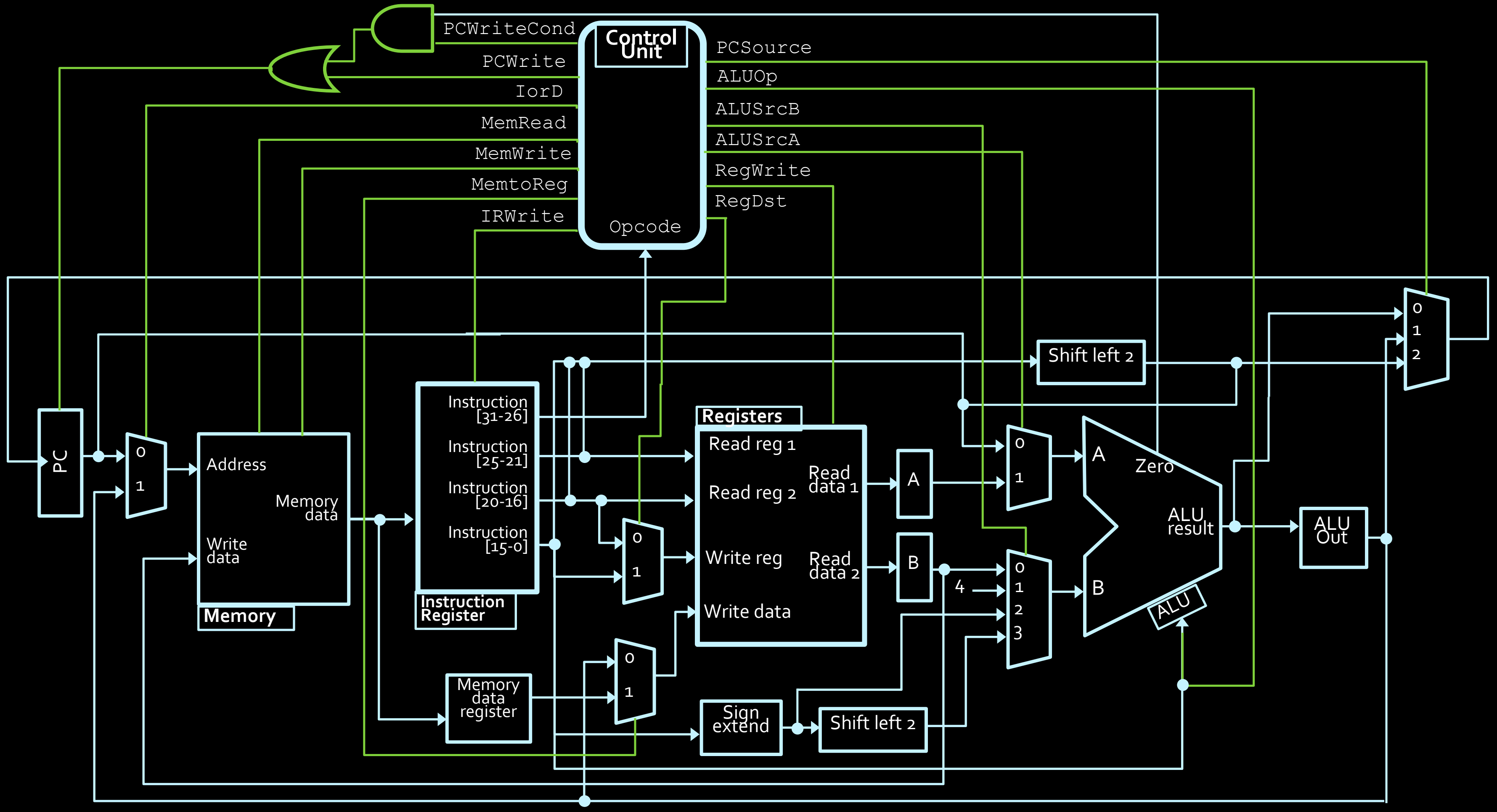


New Topic:

Processor Components

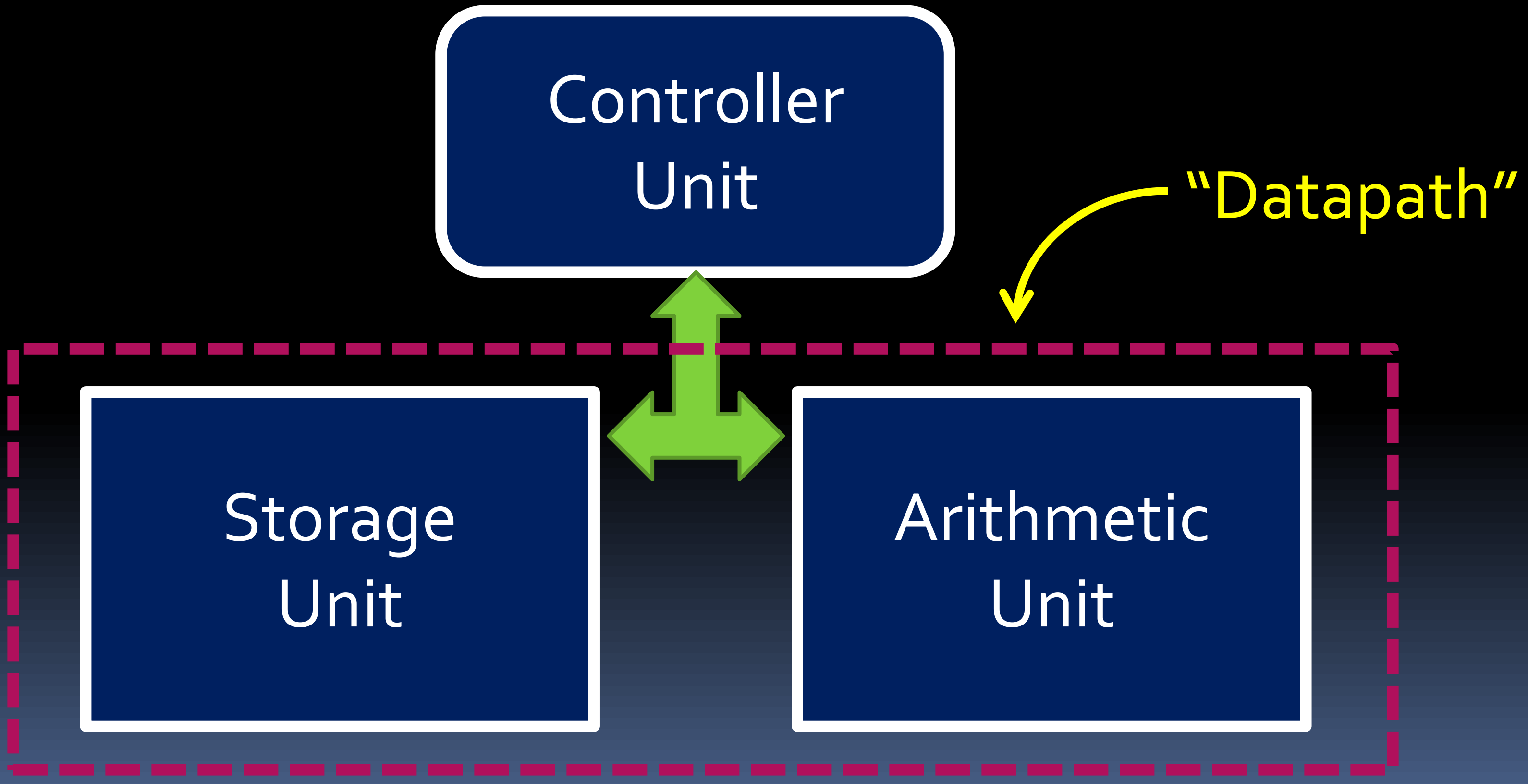
Using what we have learned so far (combinational logic, devices, sequential circuits, FSMs), how do we build a processor?

The Final Destination



Deconstructing processors

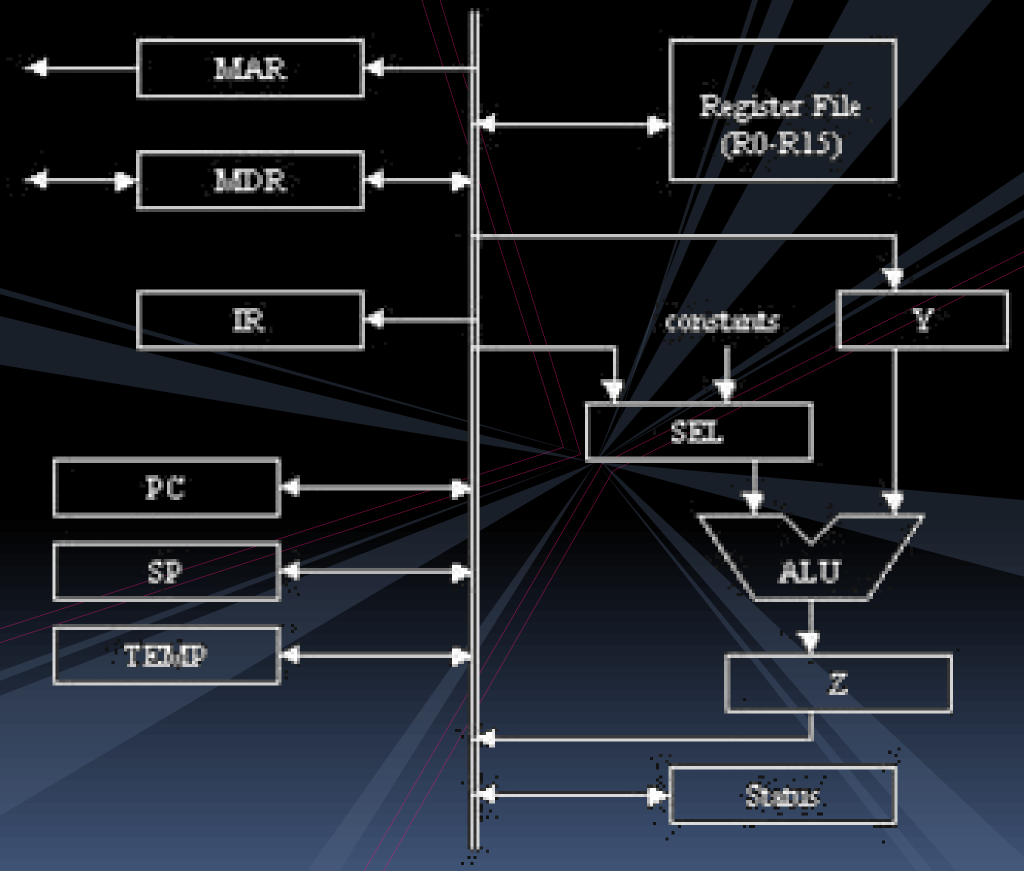
- Simpler at a high level:



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.

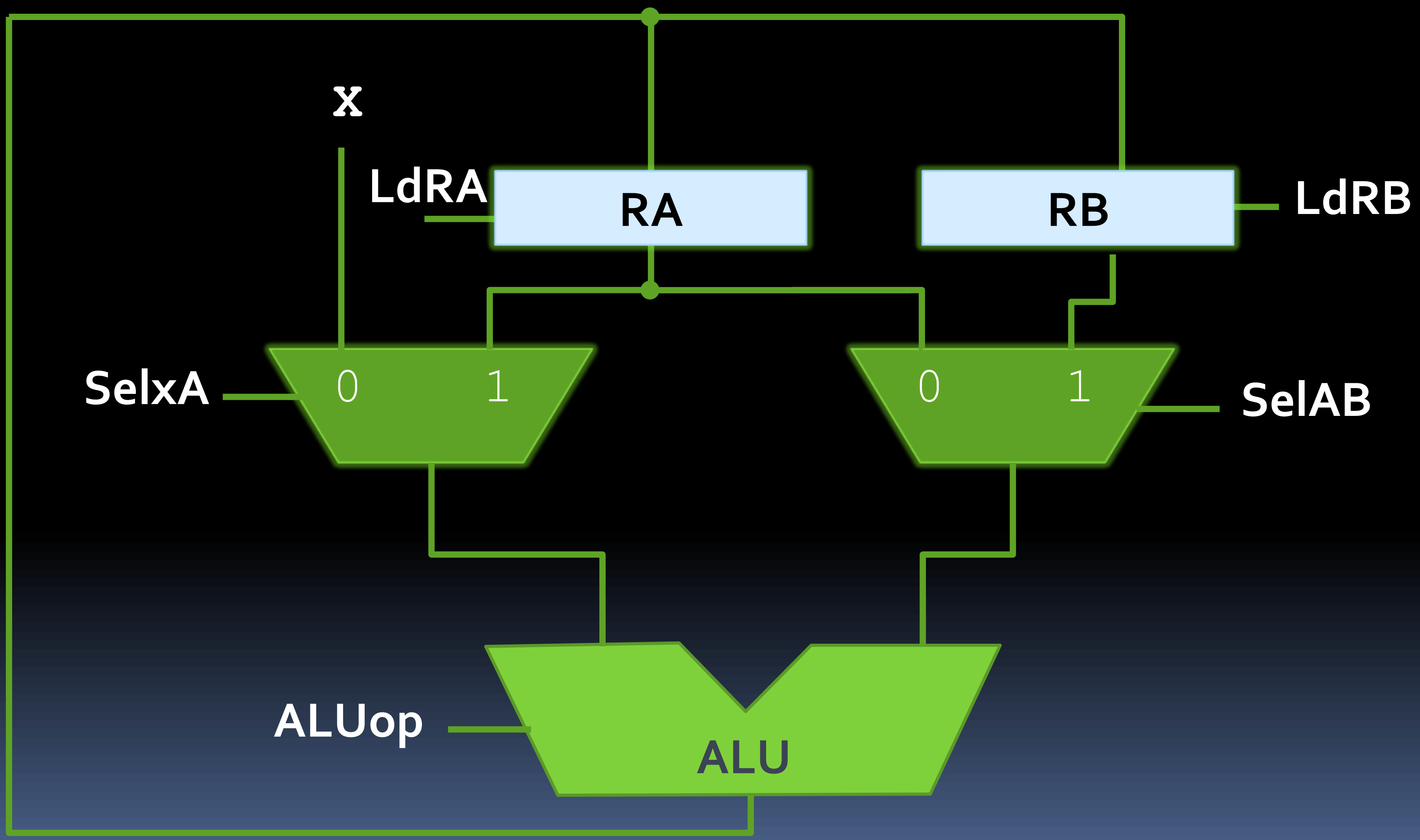
Datapath example



Example: Calculate $x^2 + 2x$

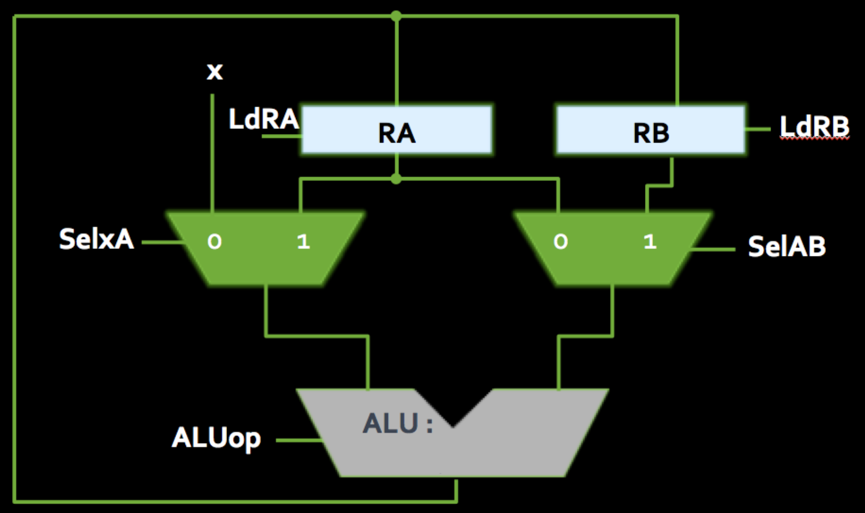
- Assume that you have access to a value from an external source. How would you calculate $x^2 + 2x$ with components you've seen so far?
- Components needed:
  - ALU (to add, subtract and multiply values)
  - Multiplexers (to determine what the inputs should be to the ALU)
  - Registers (to hold values used in the calculation)

Example schematic



Making the calculation

- Steps for $x^2 + 2x$:
  - 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 $x^2 + 2x$.



- How do we make this 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

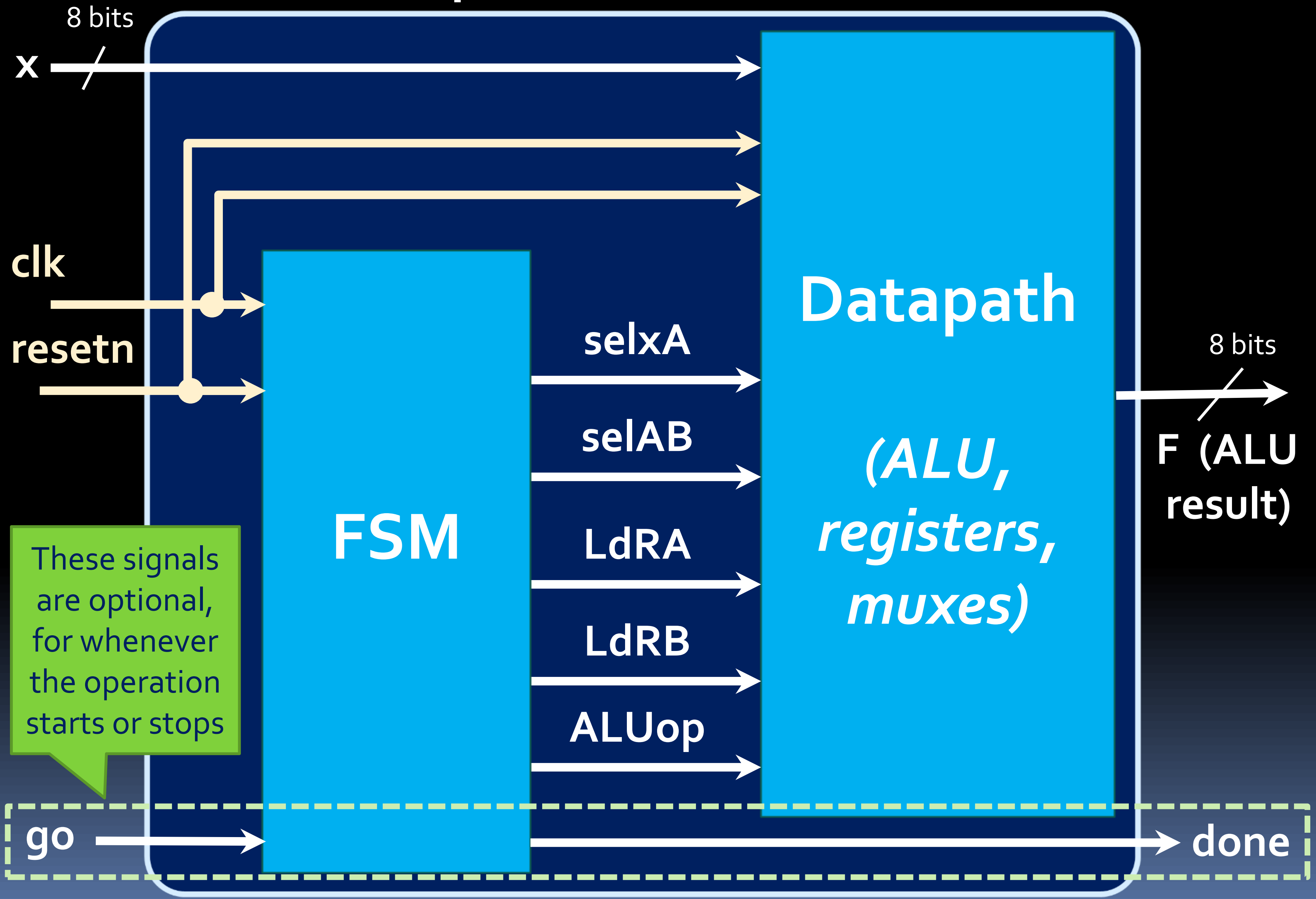
- SelxA = 0, ALUop = A, LdRA = 1, LdRB = 1
- SelxA = 1, SelAB = 1, ALUop = Multiply, LdRA = 1
- SelxA = 0, SelAB = 0, ALUop = Add, LdRA = 1
- SelxA = 0, SelAB = 0, ALUop = Add

- **Who sends these signals?**

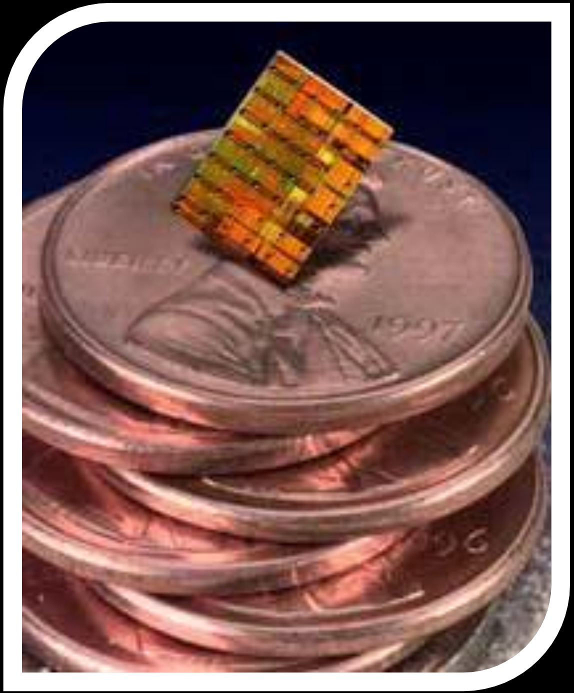
Control Unit

- Basically, 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

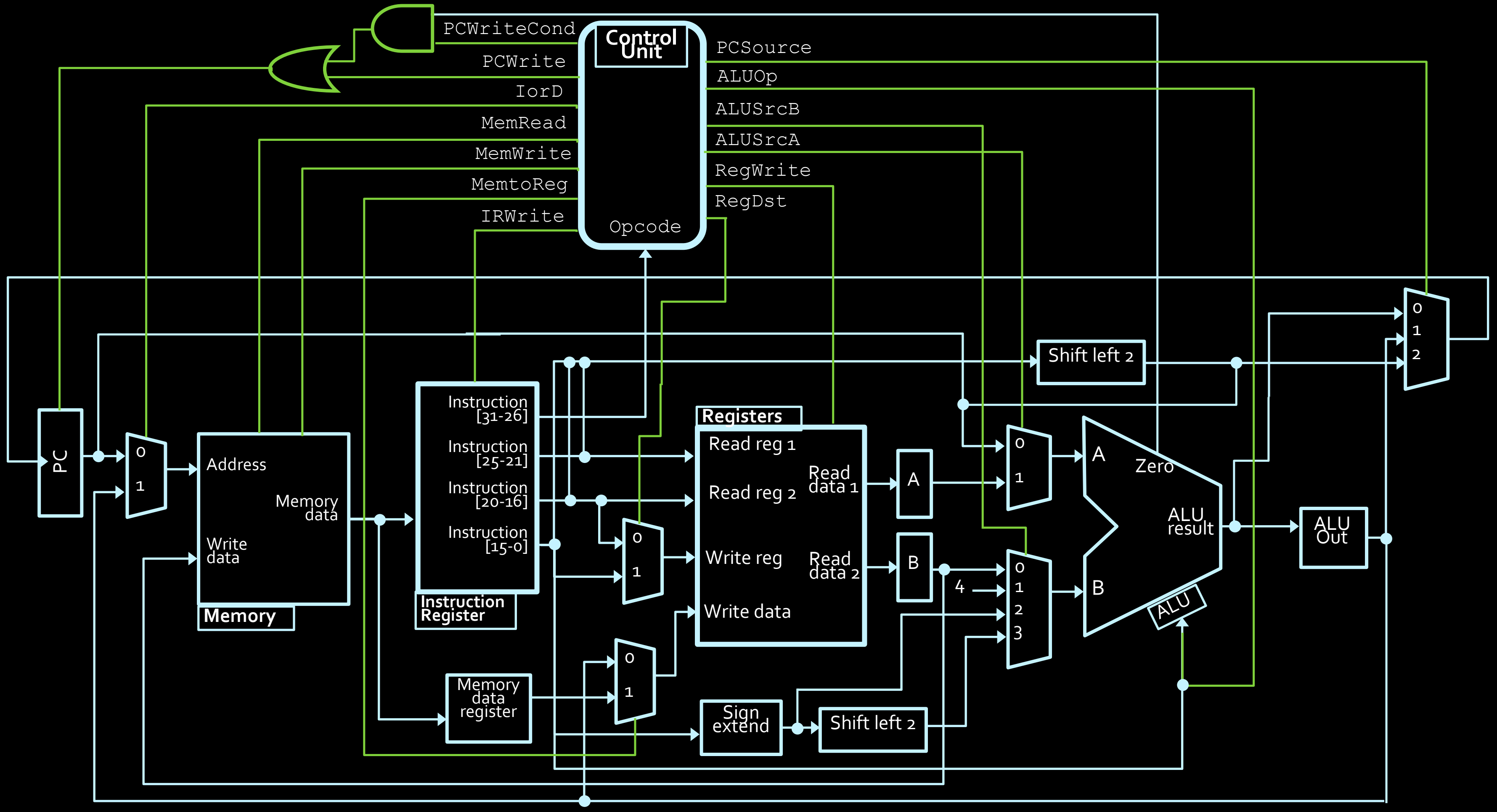


Microprocessors



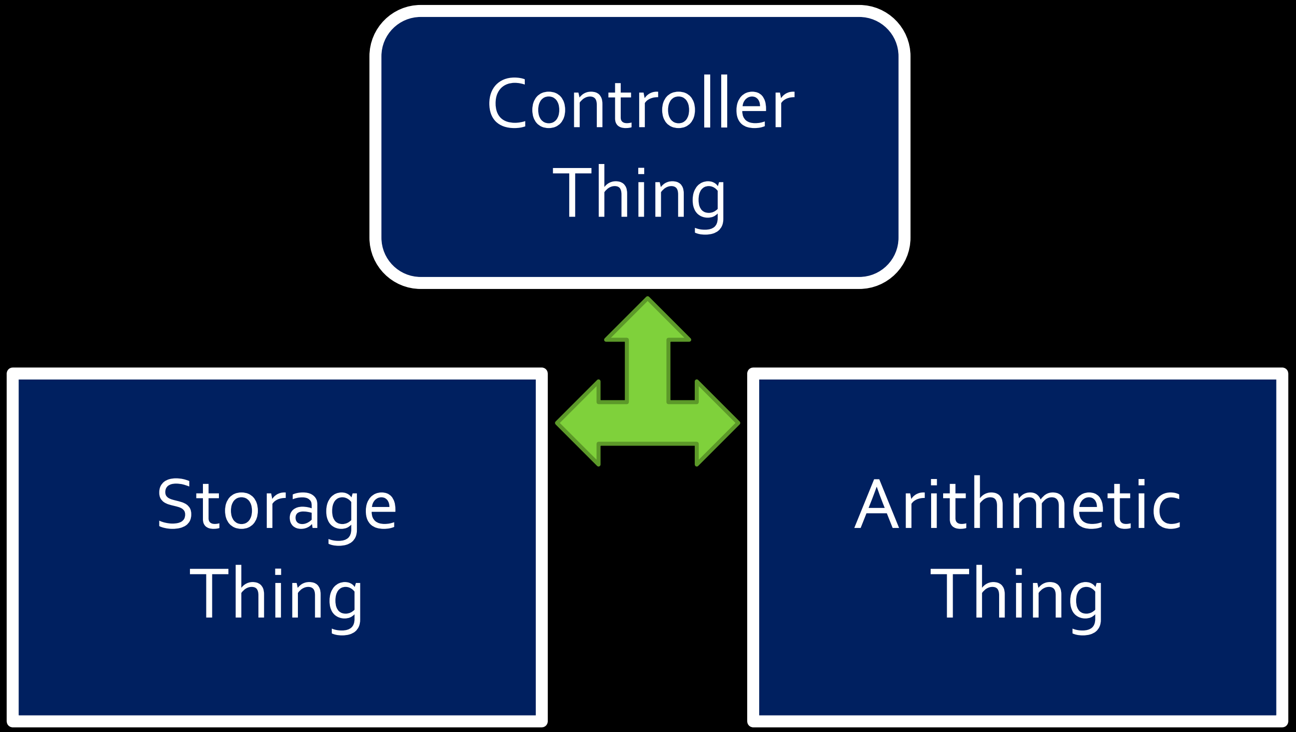
- So far, we've been talking about making devices, such as adders, counters and registers.
- The ultimate goal is to make a microprocessor, which is a digital device that processes input, can store values and produces output, according to a set of on-board instructions.

The Final Destination

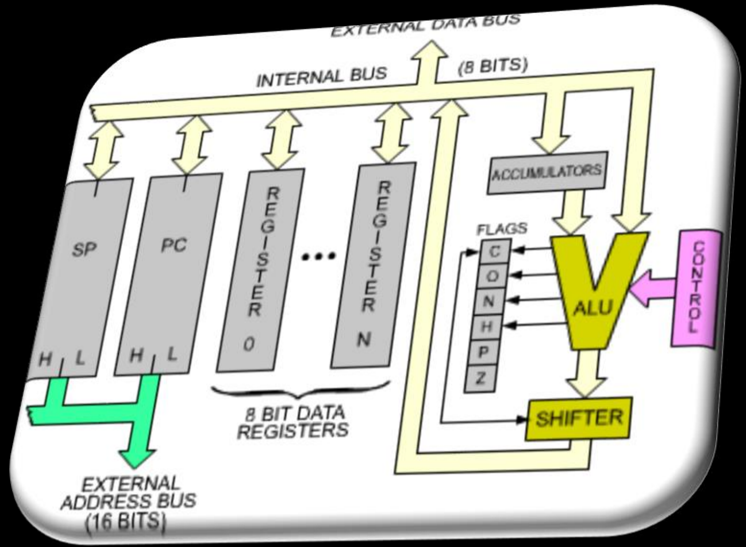


Deconstructing processors

- Processors aren't so bad when you consider them piece by piece:



Microprocessors

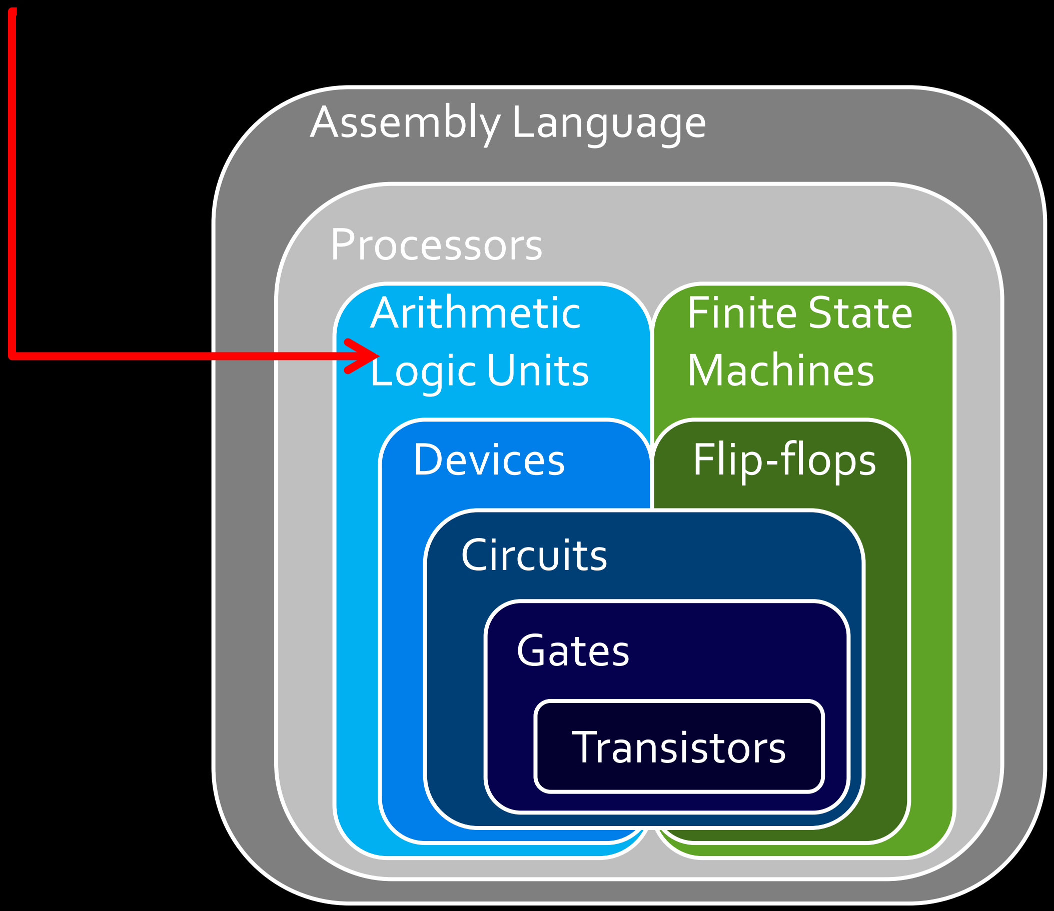


- These devices are a combination of the units that we've discussed so far:
  - Registers to store values.
  - Adders and shifters to process data.
  - Finite state machines to control the process.
- Microprocessors are the basis of all computing since the 1970's, and can be found in nearly every sort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

The “Arithmetic Thing”

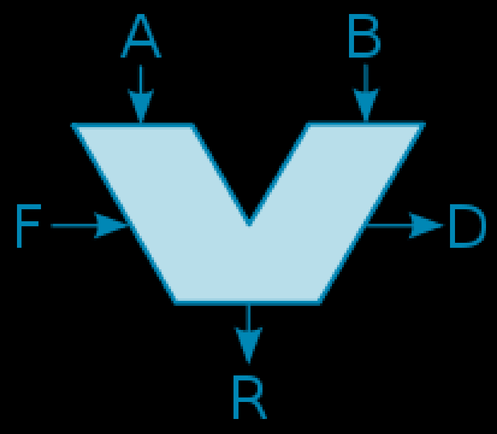
aka: the Arithmetic Logic Unit (ALU)

We are here


Assembly Language
Processors
Arithmetic Logic Units
Finite State Machines
Devices
Flip-flops
Circuits
Gates
Transistors

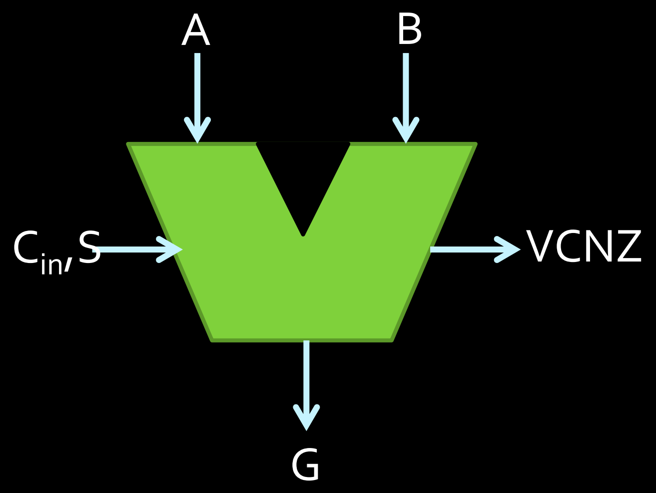
Arithmetic Logic Unit

- 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).
- This larger structure is responsible for the processing of all data values in a basic CPU.



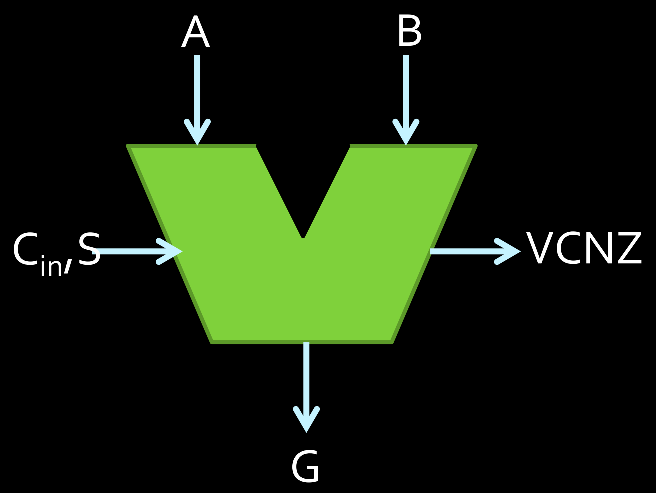
ALU inputs

- The ALU performs all of the arithmetic operations covered in this course so far, and logical operations as well (AND, OR, NOT, etc.)
  - A and B are the operands
  - The select bits (S) indicate which operation is being performed (S2 is a mode select bit, indicating whether the ALU is in arithmetic or logic mode).
  - The carry bit C_{in} is used in operations such as incrementing an input value or the overall result.



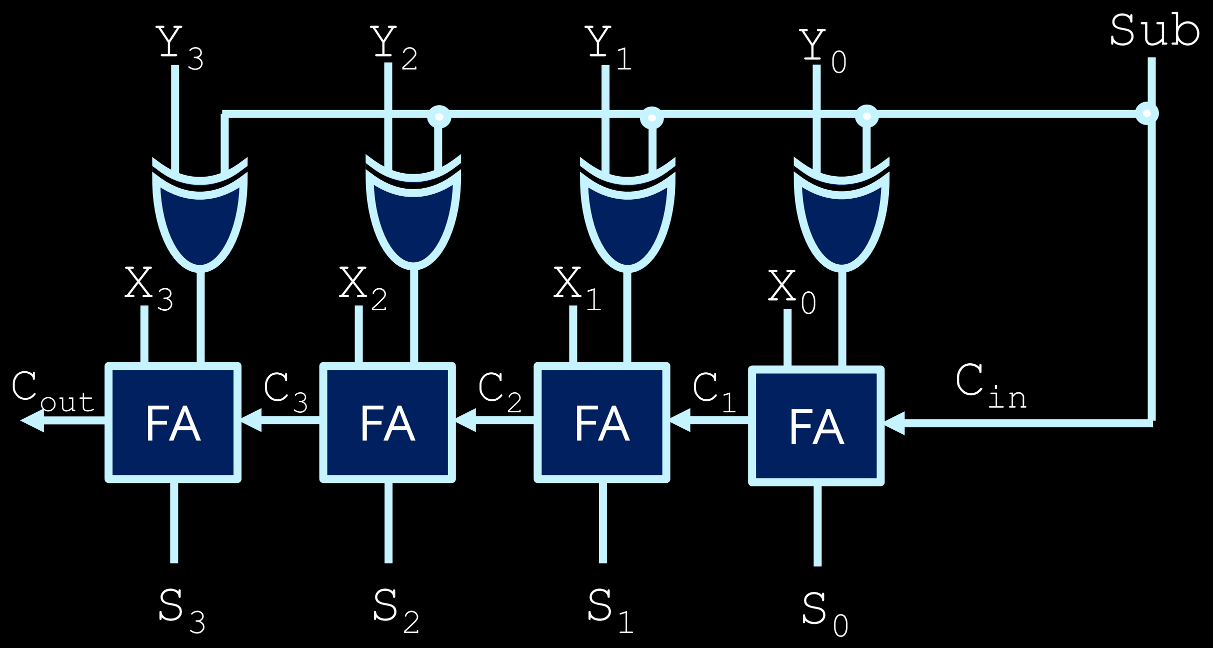
ALU outputs

- 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
  - N: Negative indicator
  - Z: Zero-condition indicator



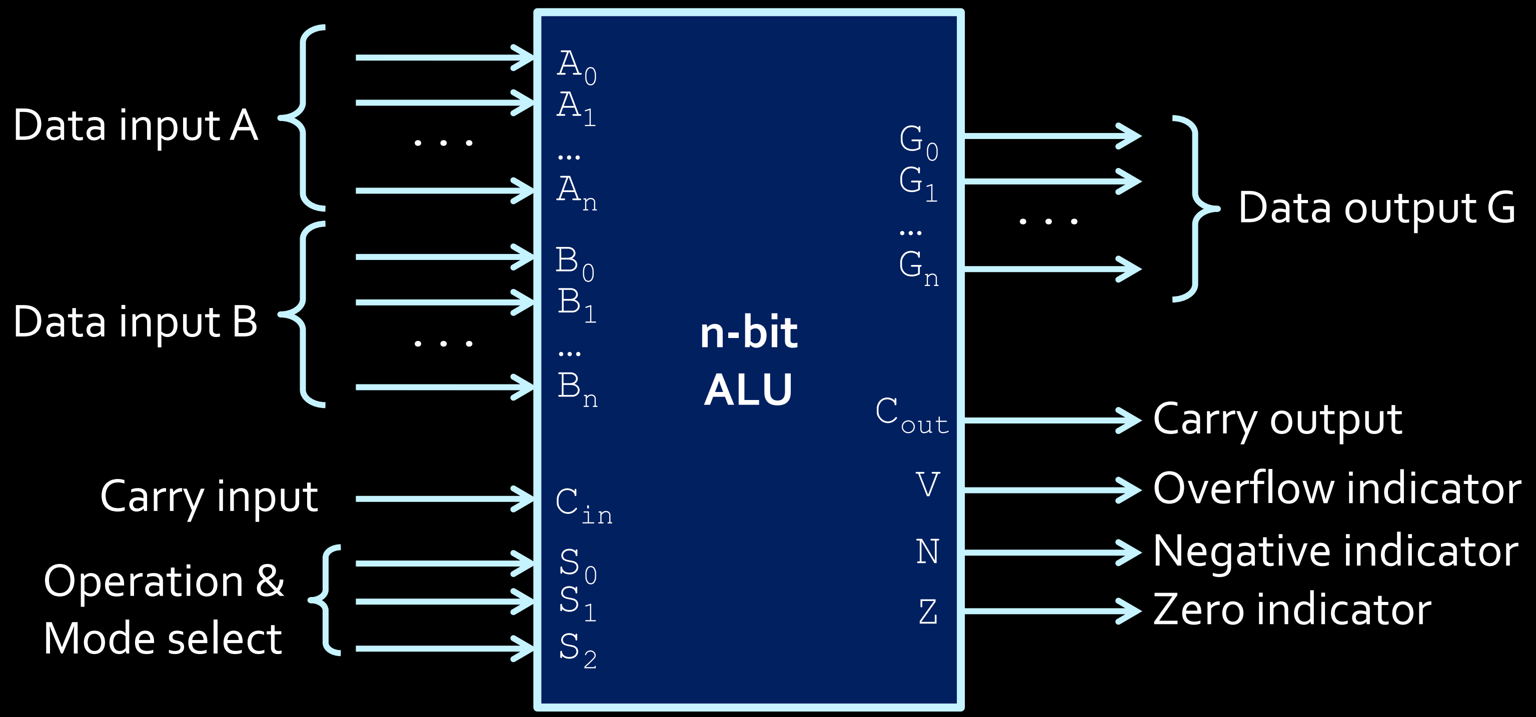
The “A” of ALU

- 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:

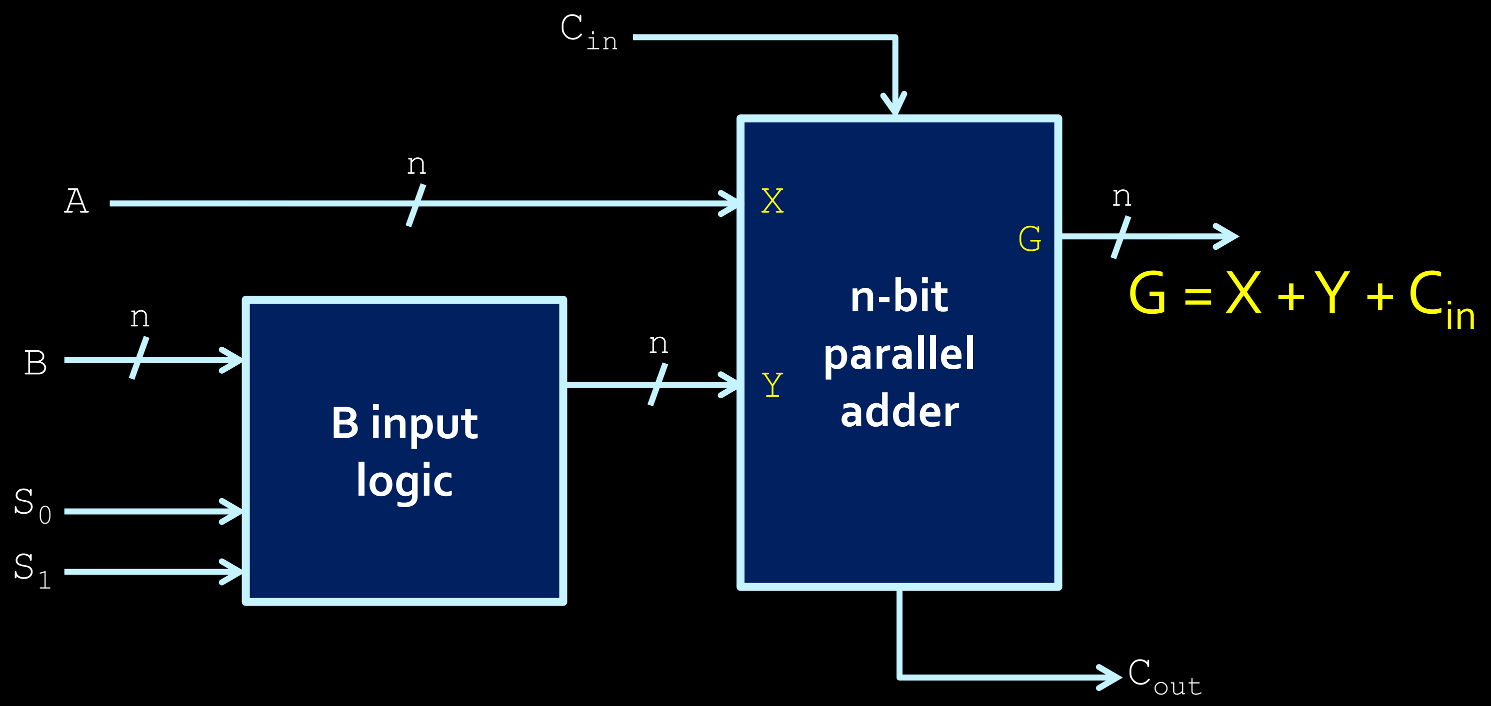


ALU block diagram

- 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`).

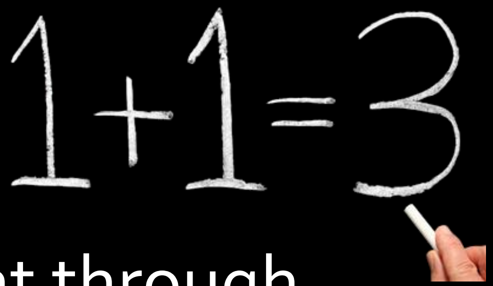


Arithmetic components



- In addition to addition and subtraction, many more operations can be performed by manipulating what is added to input B, 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

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…
- What about the carry bit?

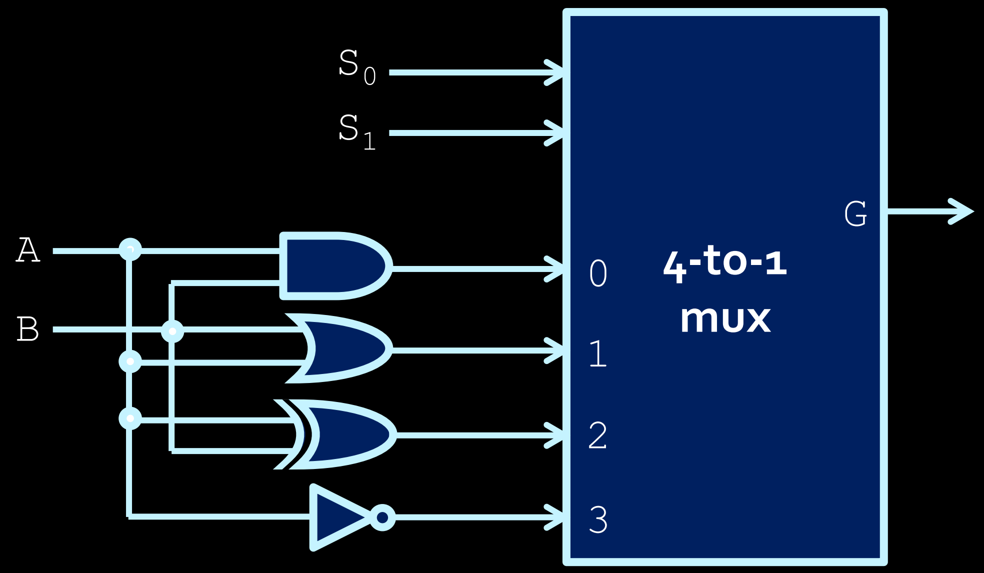
Full operation selection

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)

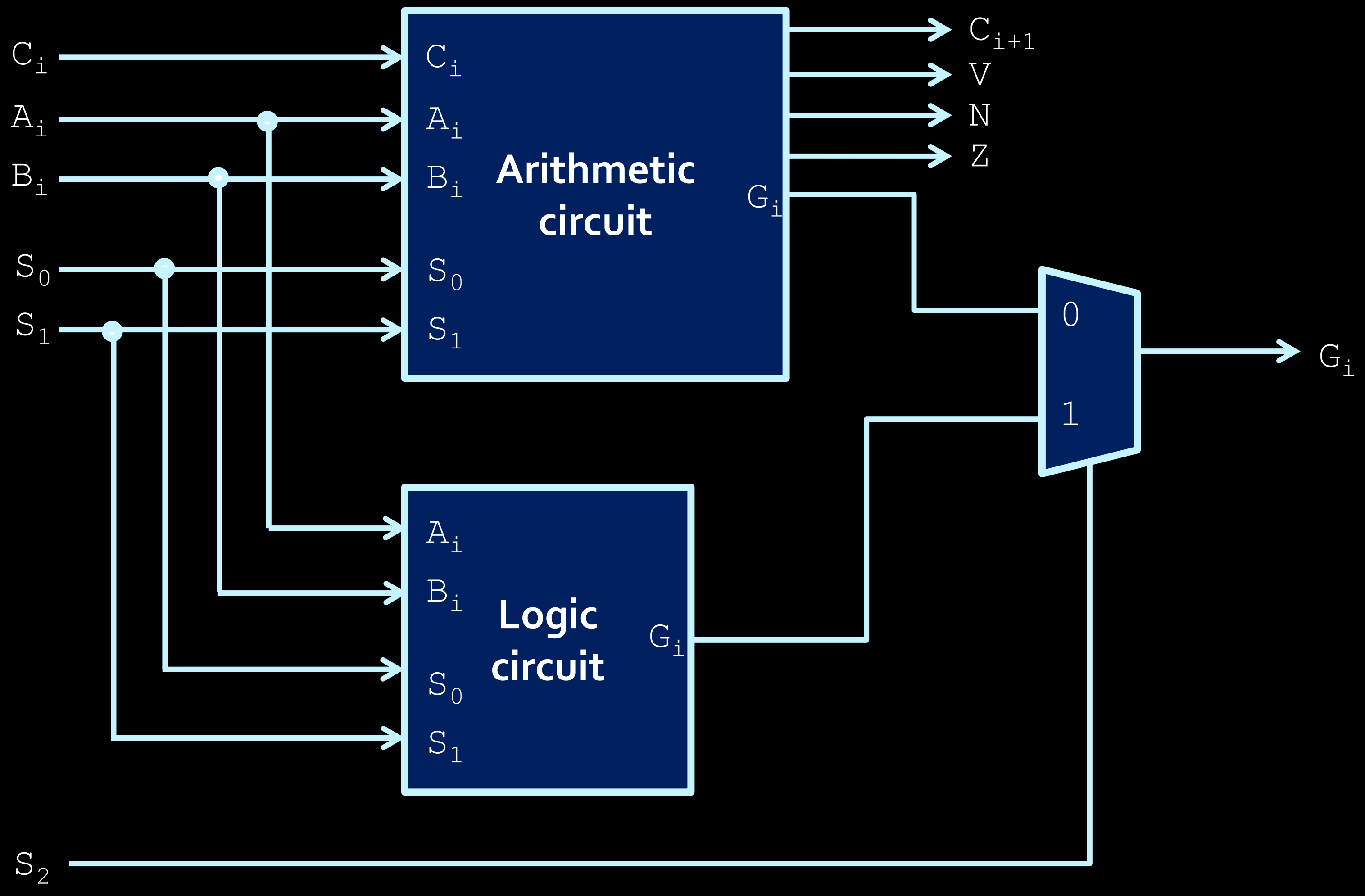
- Based on the values on the select bits and the carry bit, we can perform any number of basic arithmetic operations by manipulating what value is added to A.

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 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

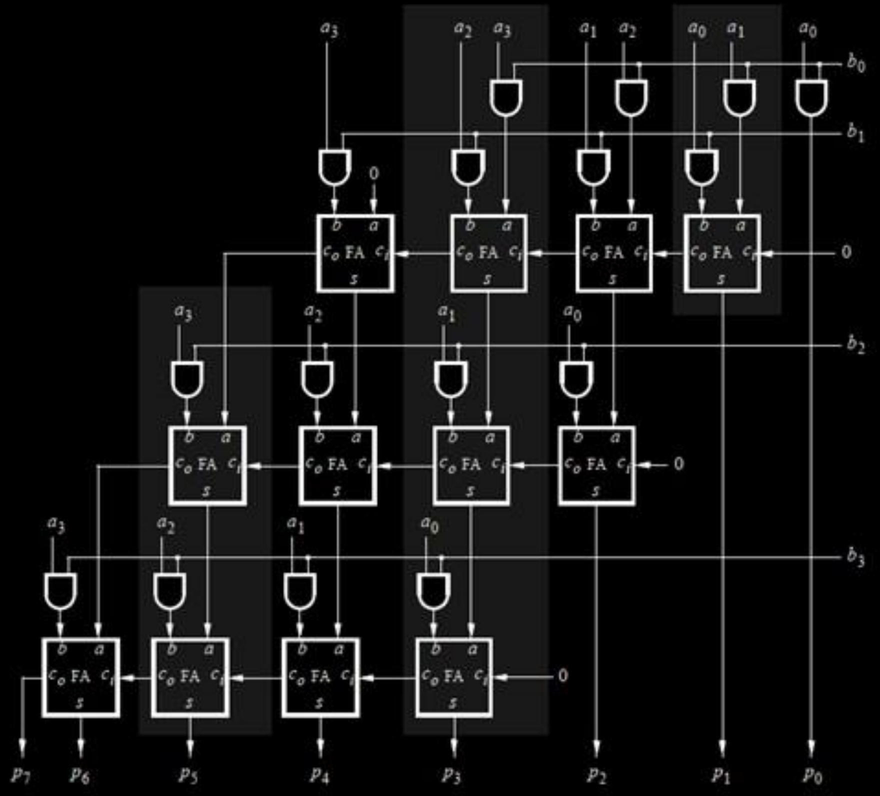


Multiplication

What about multiplication?

- Multiplication (and division) operations are always more complicated than other arithmetic (addition, subtraction) 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

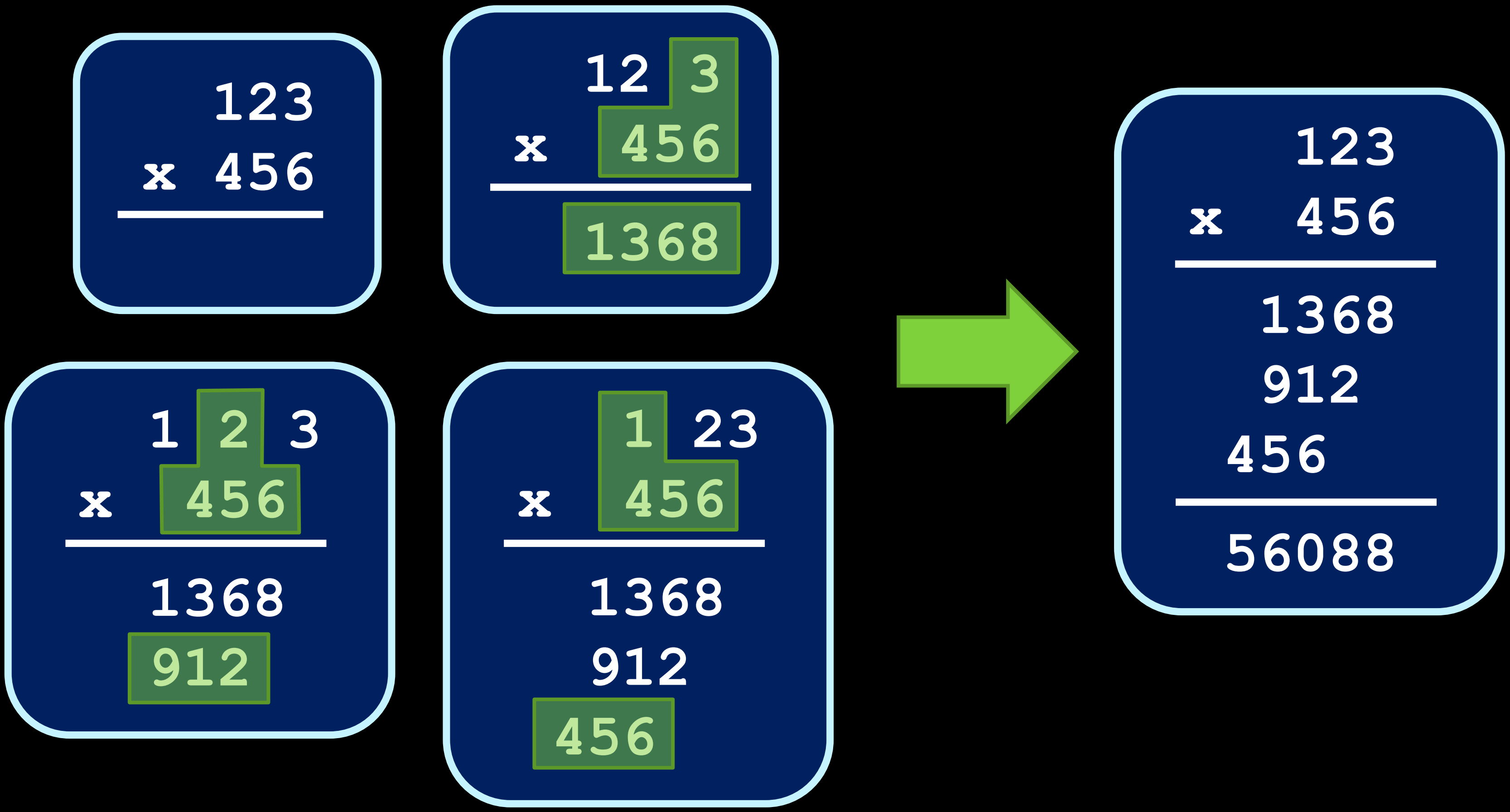
Multiplication



- Multiplier circuits can be constructed as an array of adder circuits.
- This can get a little expensive as the size of the operands grows.
- Is there an alternative to this circuit?

Multiplication

- Revisiting grade 3 math…



Multiplication

- And now, in binary…

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

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

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

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

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

Observations

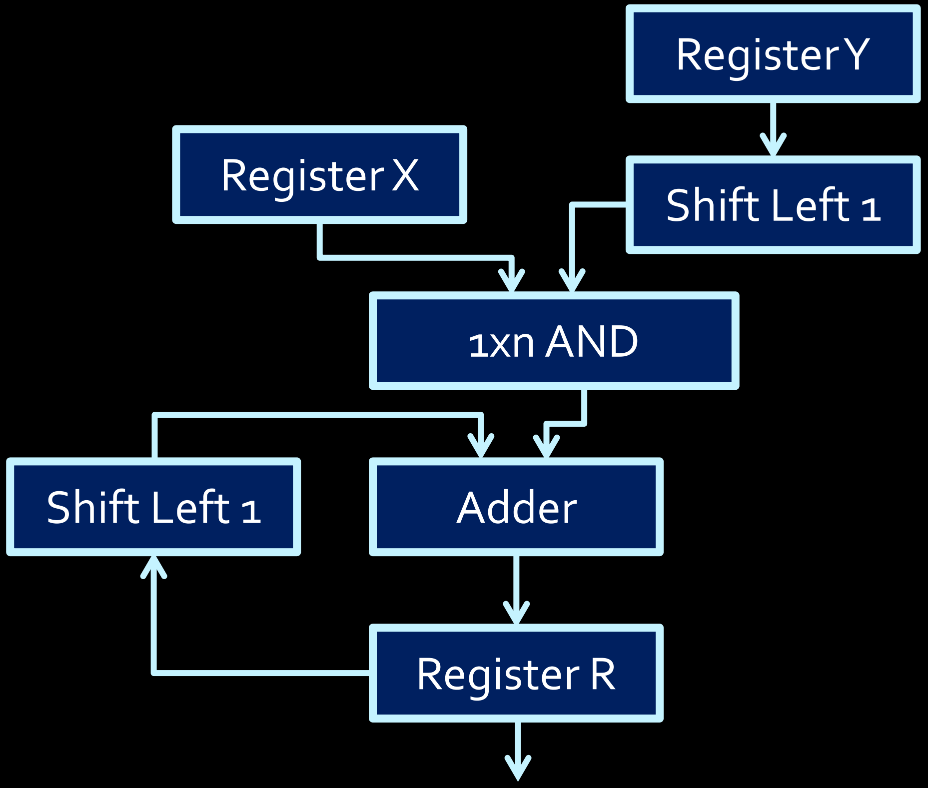
- Calculation flow
  - Multiply by 1 bit of multiplier
  - Add to sum and shift sum
  - Shift multiplier by 1 bit
  - Repeat the above
- What is "multiply by 1 bit of binary"?
  - `10101 x 1 ?`
  - `10101 x 0 ?`
  - `It's an` **AND**`!`

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

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.

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



Make it more efficient

Think about 258 x 9999

- Multiply by 9, add to sum, shift, multiply by 9, add to sum, shift, multiple by 9, add to sum, shift, multiply by 9, add to sum.

- 258 x 9999 = 258 x (10000 - 1) = 258 x 10000 – 258
- Just shift 258, becomes 2580000, then do 2580000 – 258
- More efficient!

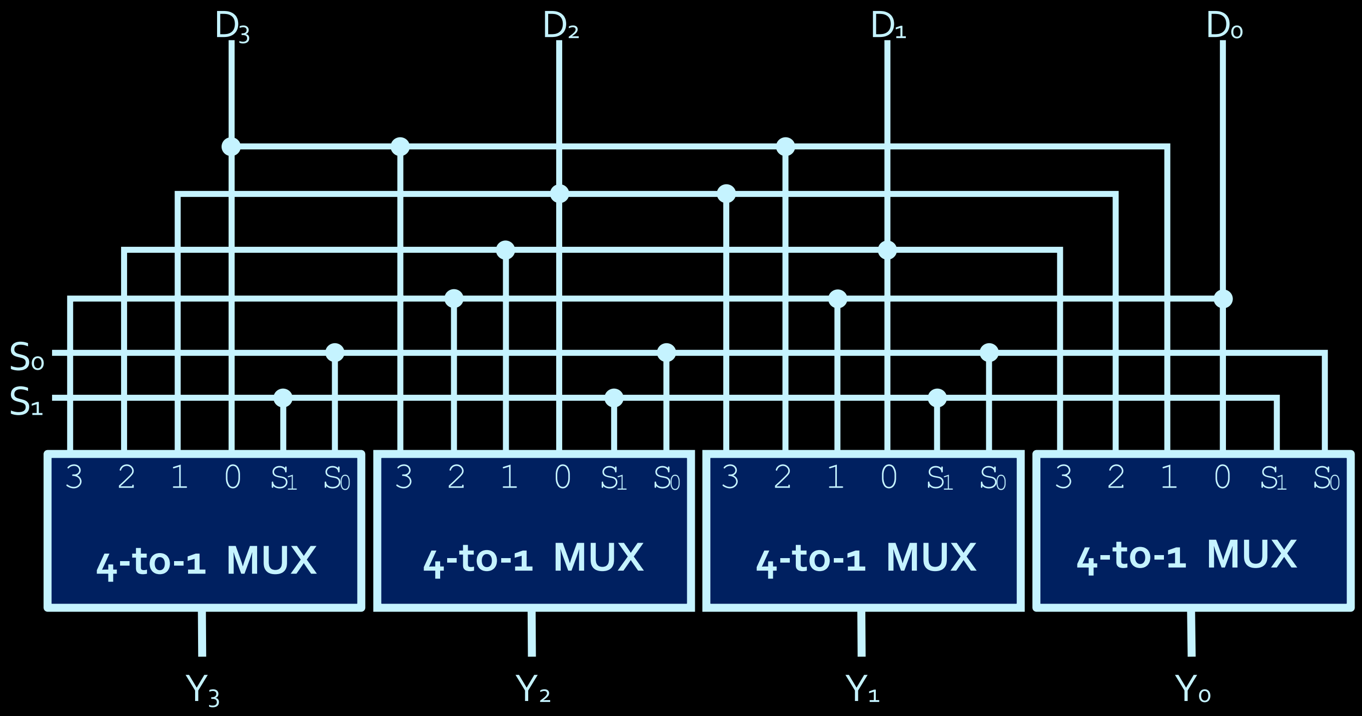
Efficient Multiplication: Booth's Algorithm

- Take advantage of circuits where shifting is cheaper than adding, or where space is at a premium.
  - 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`
- More details: https://en.wikipedia.org/wiki/Booth%27s_multiplication_algorithm

Reflections on multiplication

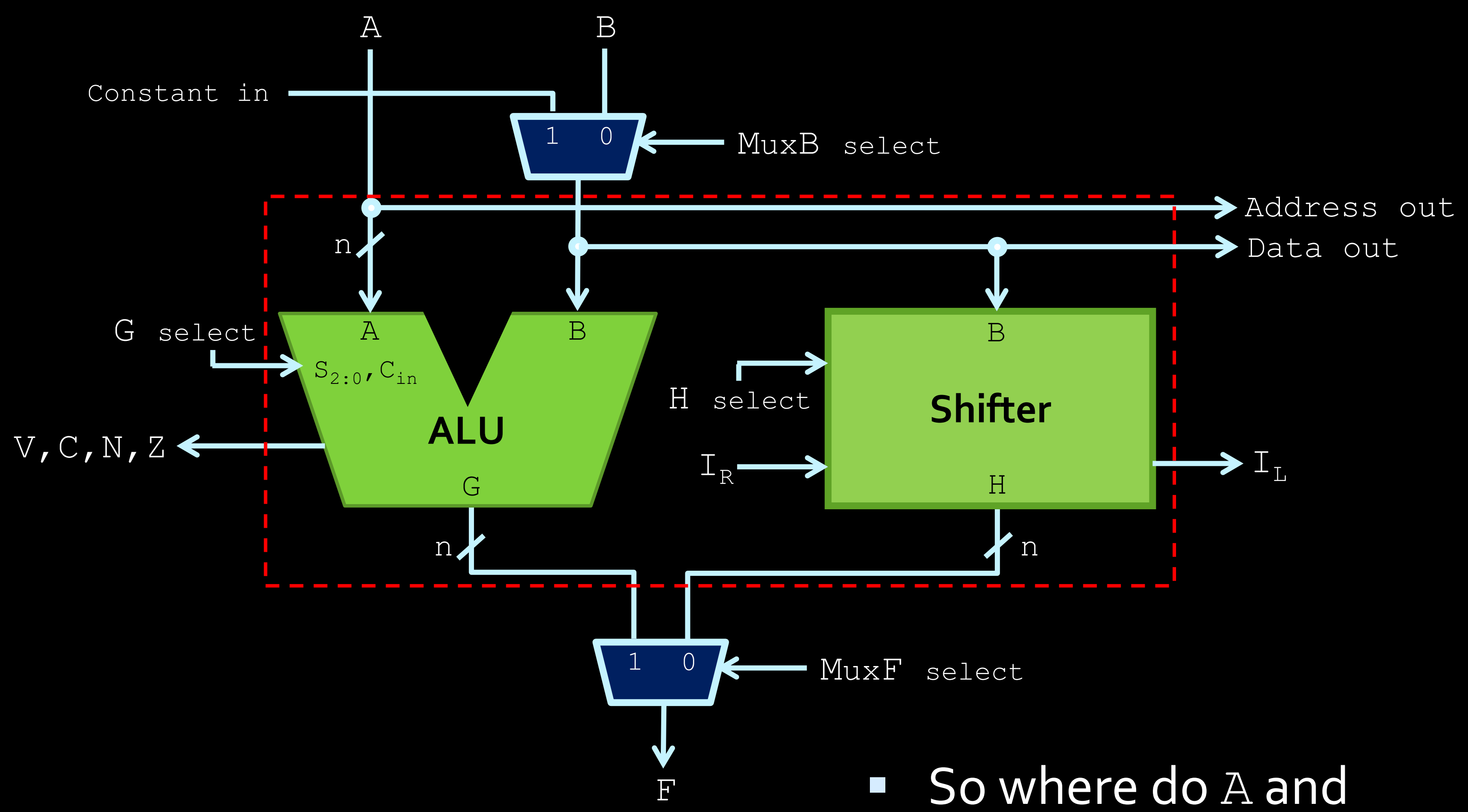
- Multiplication isn't as common an operation as addition or subtraction, but occurs enough that its implementation is handled in the hardware.
- Most common multiplication and division operations are powers of 2. For this, we do shifting instead of using the multiplier circuit.
  - e.g., in your code, do `x << 3`, instead of `x * 8`

A Barrel Shifter unit

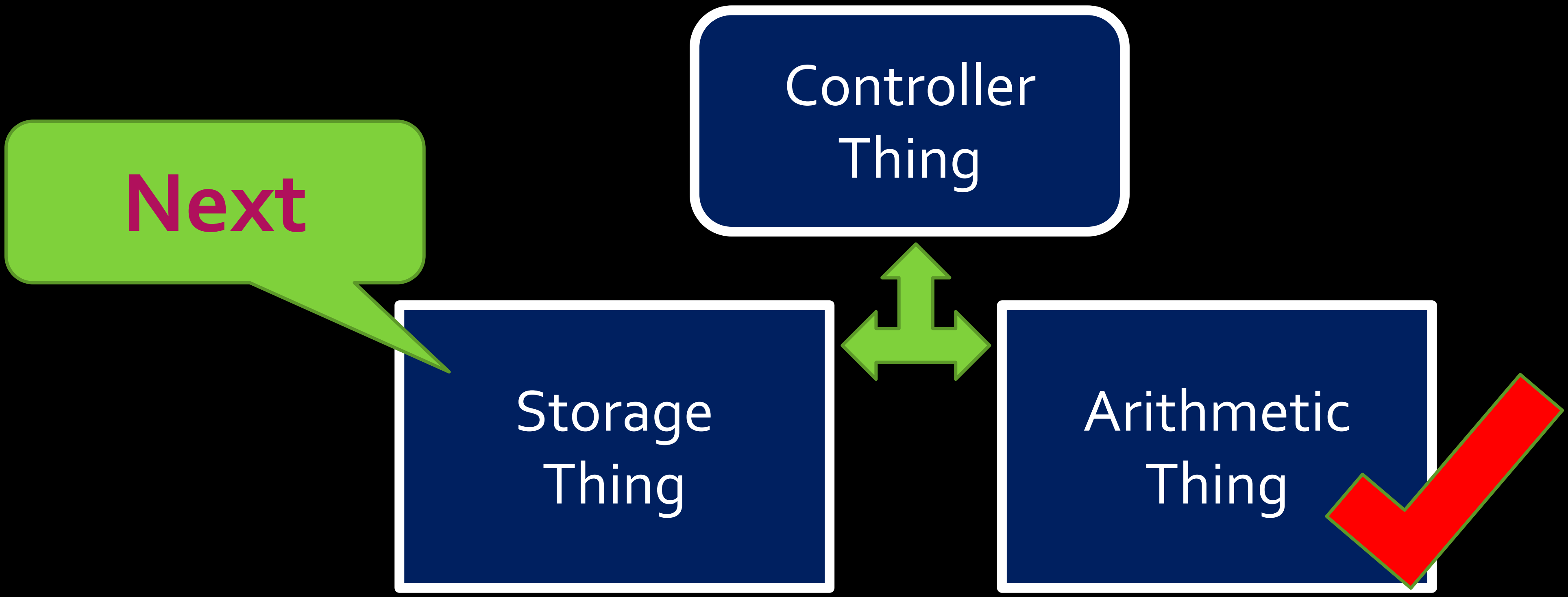


- This barrel shifter **shifts and rotates** D to the left by `S` bits.
  - If S_1S_0 is `01` => $Y = D_2D_1D_0D_3$
  - If S_1S_0 is `11` => $Y = D_0D_3D_2D_1$
- This is a purely combinational circuit, unlike the shift registers in the lab.

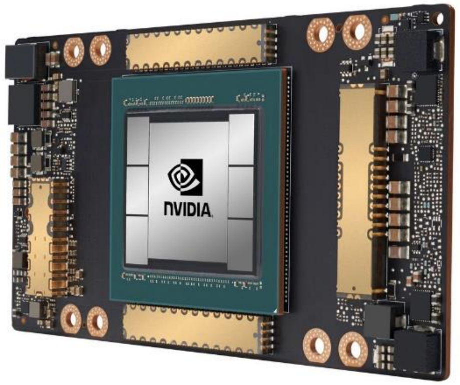
Expanding our view



- So where do A and B come from?


Controller Thing
Next
Storage Thing
Arithmetic Thing

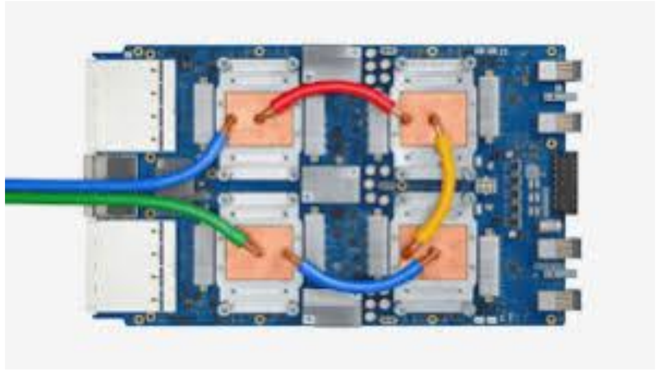
CSCB58:
Computer Organization



Prof. Gennady Pekhimenko

University of Toronto

Fall 2020



The content of this lecture is adapted from the lectures of Larry Zheng and Steve Engels

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     → B
      x     00011110     → A
      ------------
    01010010
  + 111110101110
  -------------
    0100110011100
```

Add B here

Subtract B from here

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

- 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

Booth's Algorithm Example

- 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

- 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

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

    1111 $\boxed{11}$

  - 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