Processor Components

Where we are now

Assembly Language

Processors

Arithmetic Logic Units

Finite State Machines

Devices

Flip-flops

Circuits

Gates

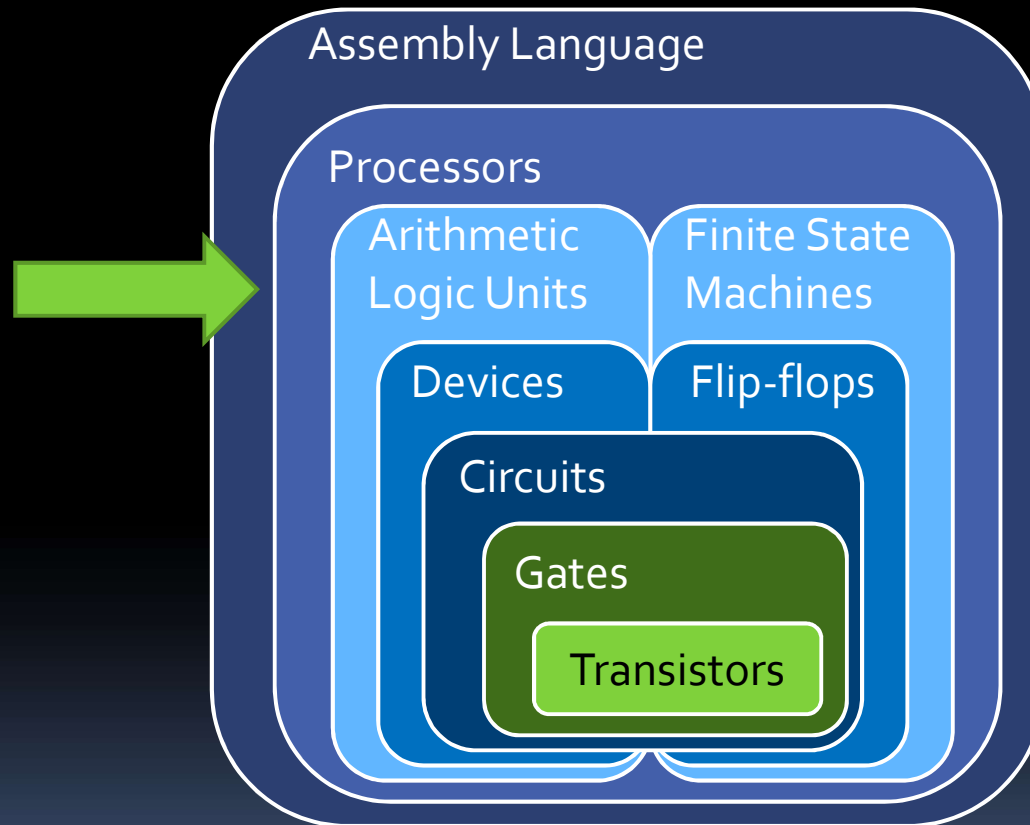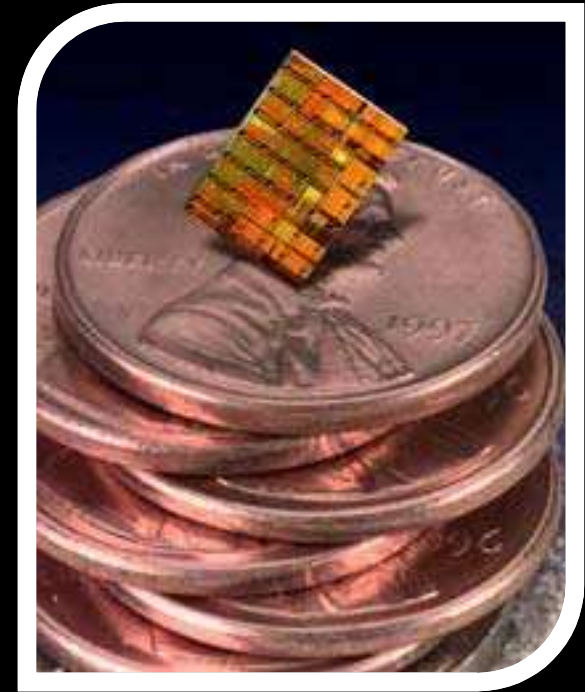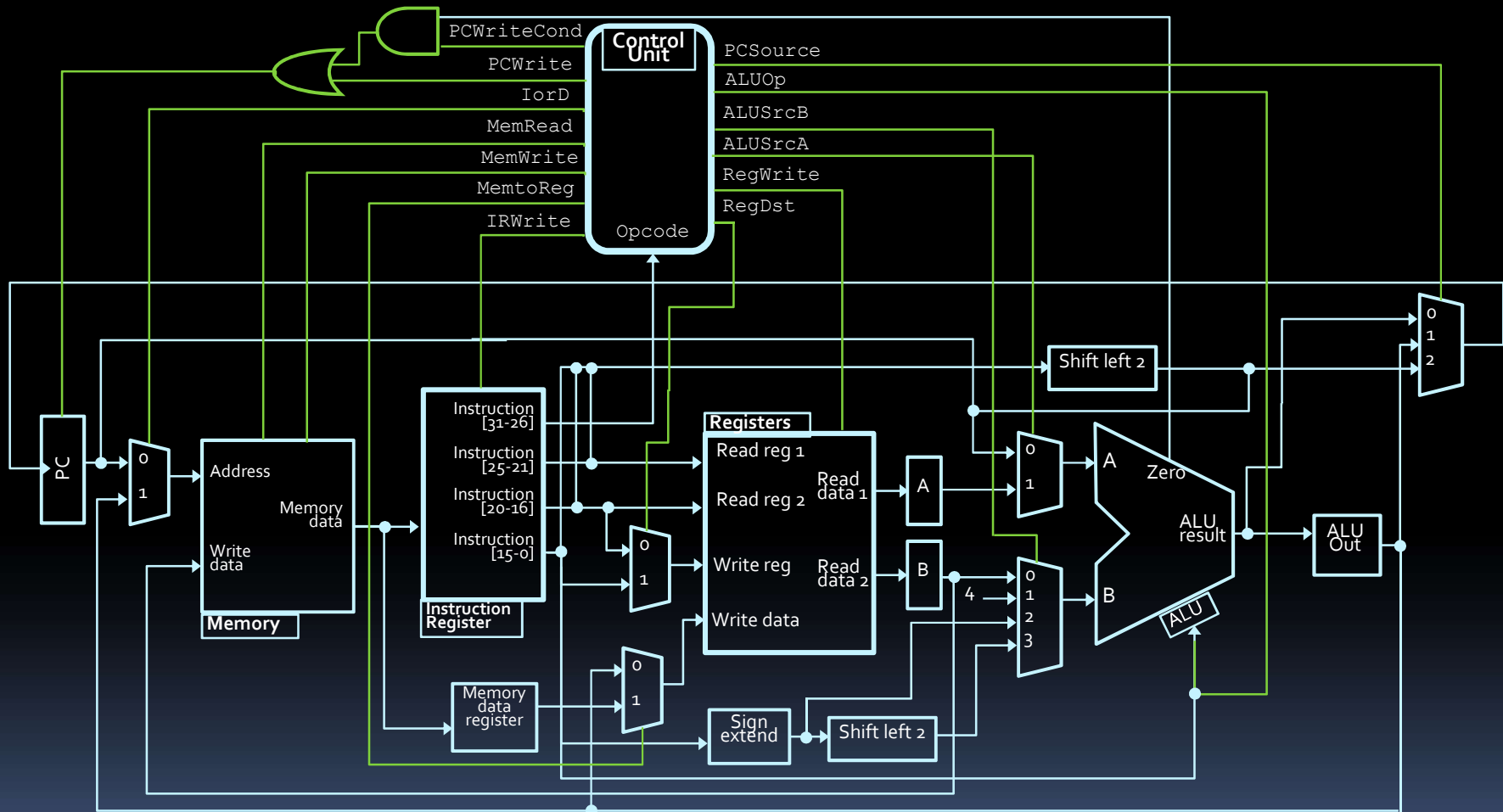Transistors

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.
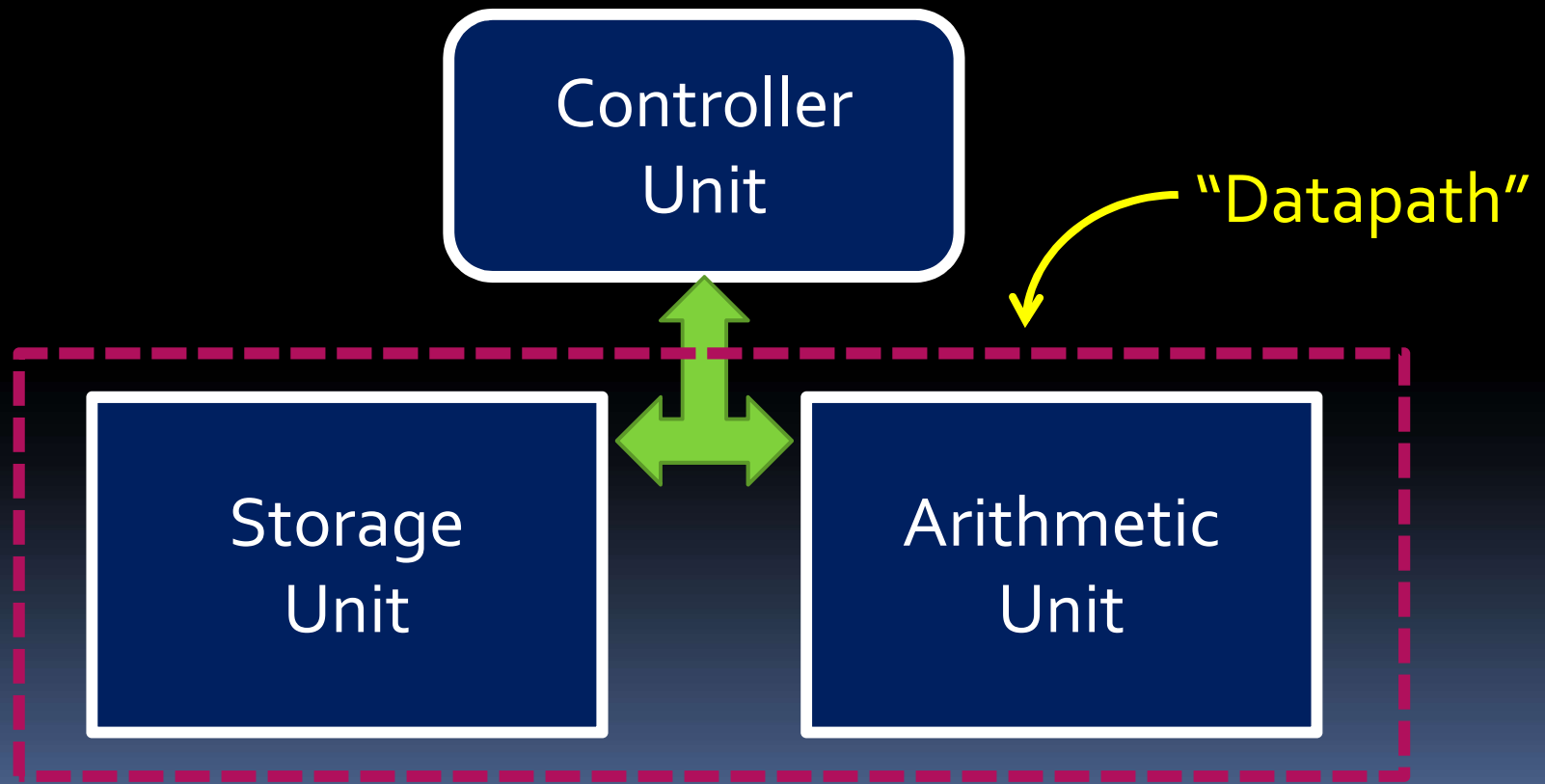
The Final Destination

Deconstructing processors

- Simpler at a high level:

Controller Unit

"Datapath"

Storage Unit

Arithmetic Unit

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
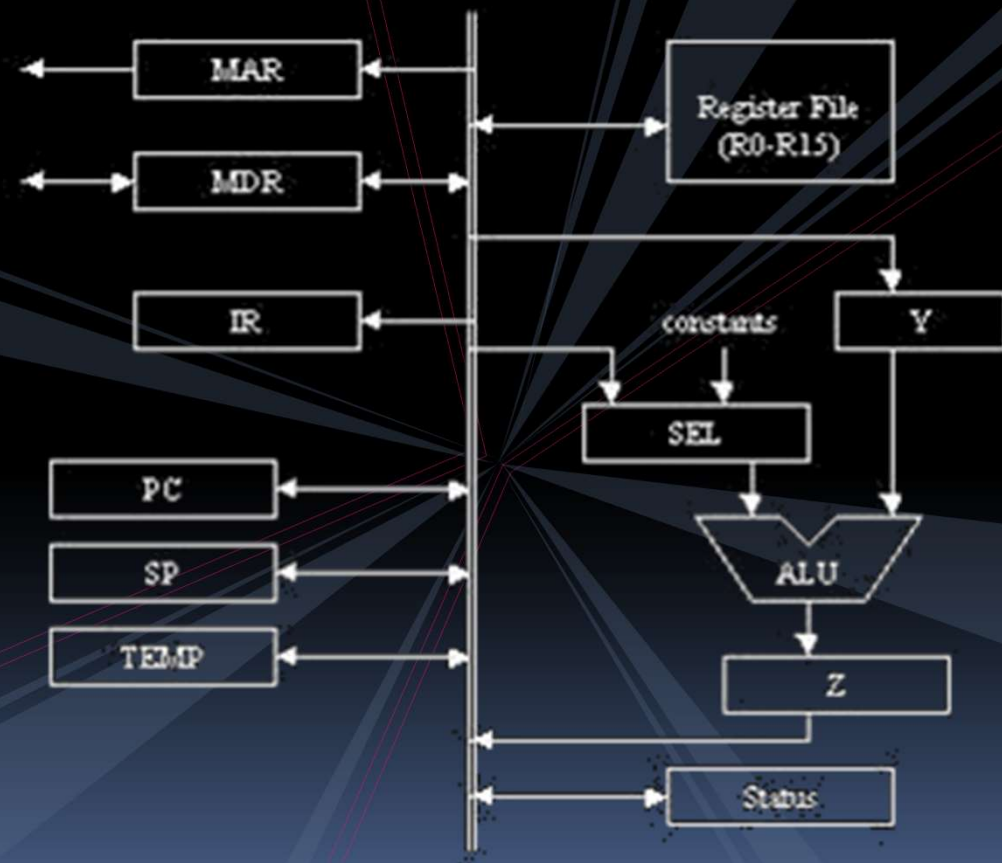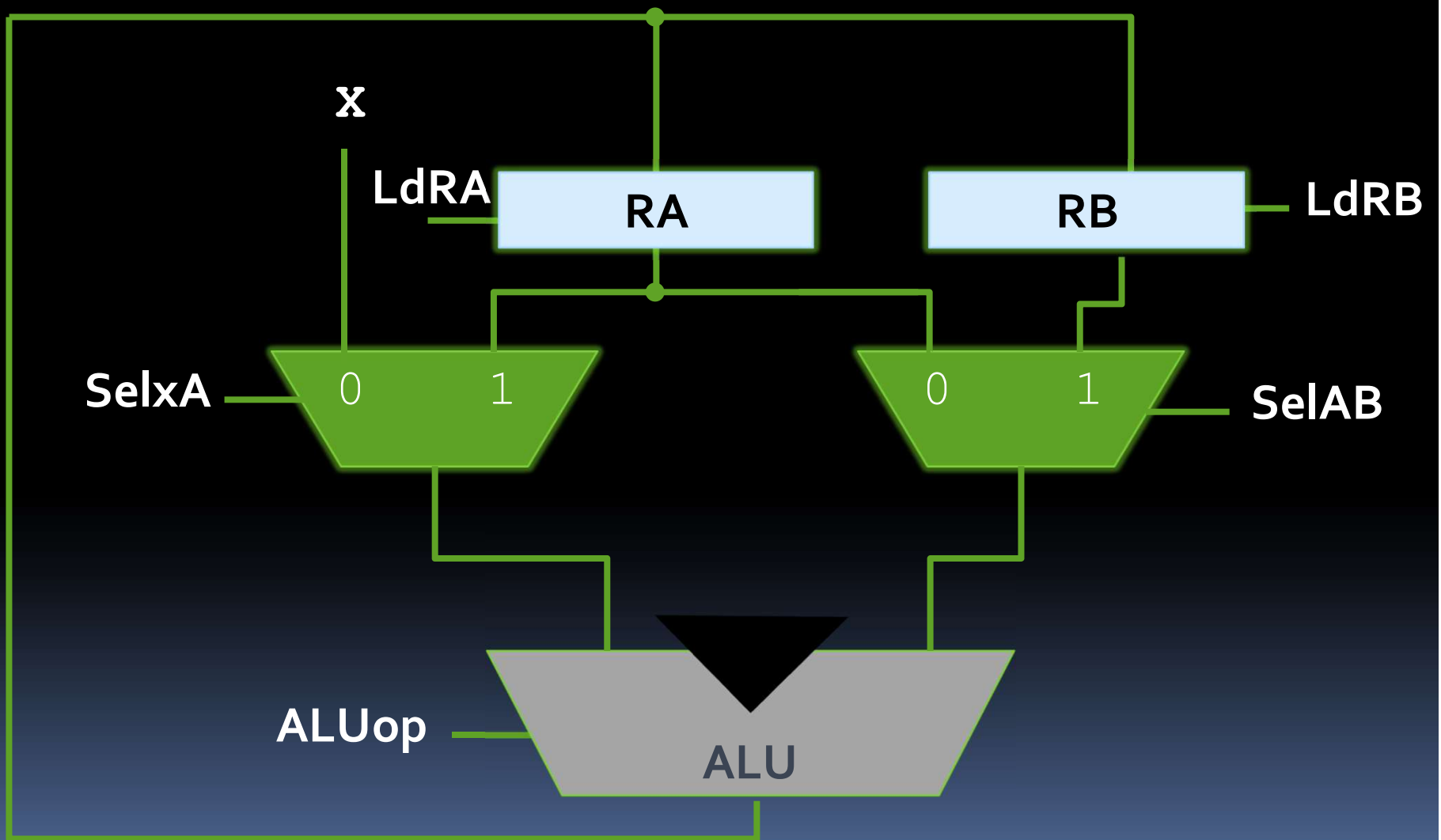
Example: Calculate $x^2 + 2x$

- 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?
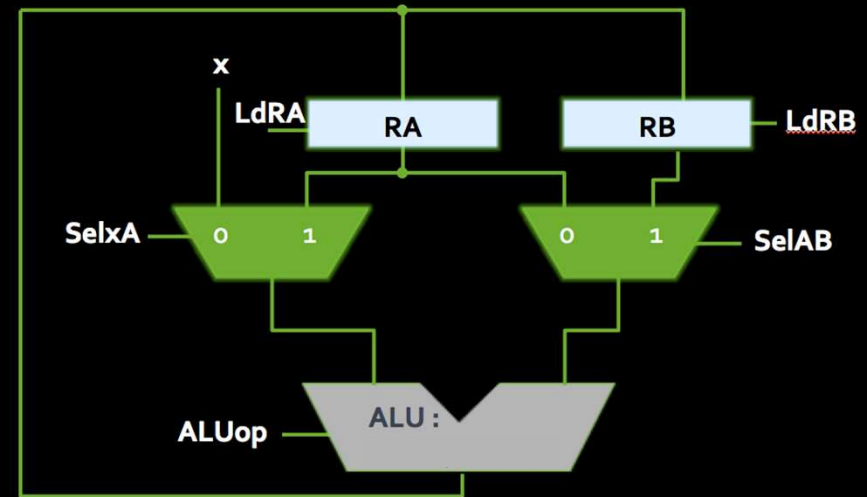
Making the calculation

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 $x^2 + 2x$.

- **Who sends these signals?**

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

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
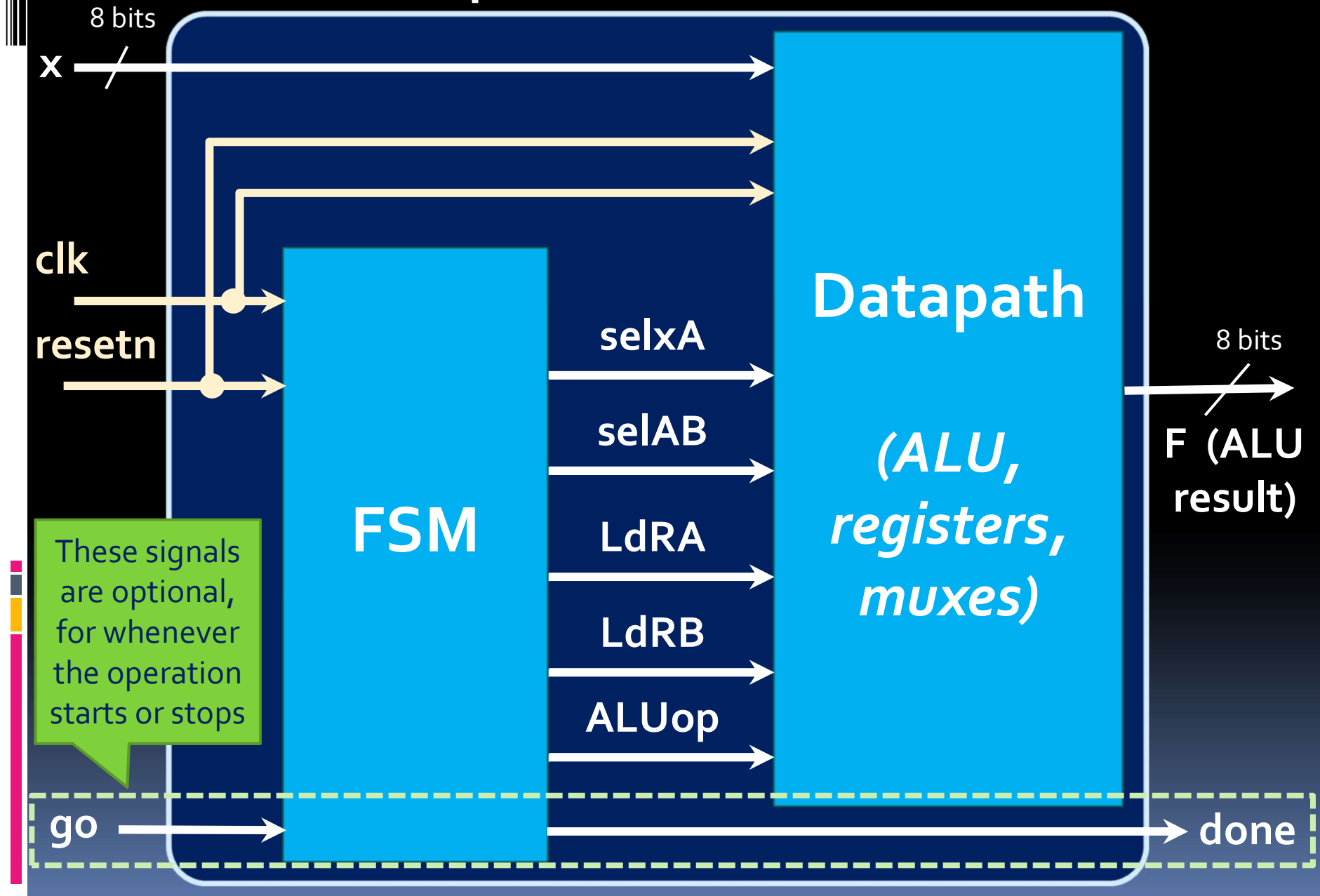
Microprocessors

- 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.
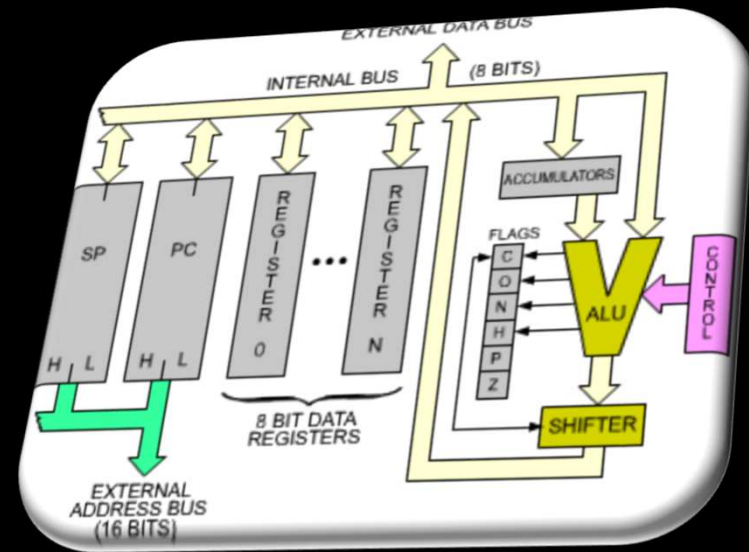
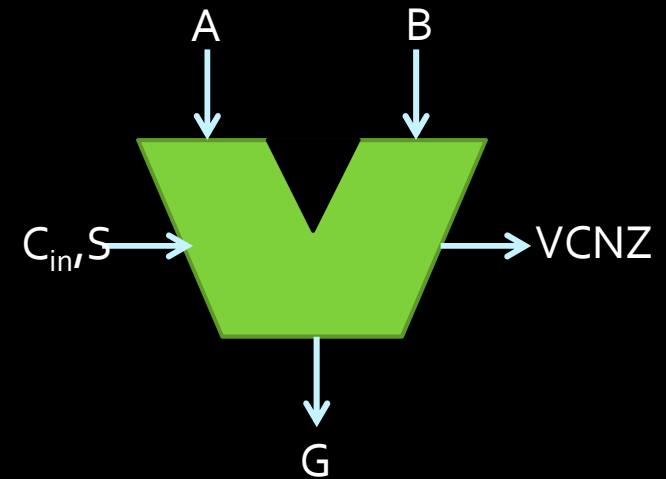# The Arithmetic Unit

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

A   B

C_{in}, S → → VCNZ

G

- 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 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

A   B

C_{in}, S → [ALU] → VCNZ

G
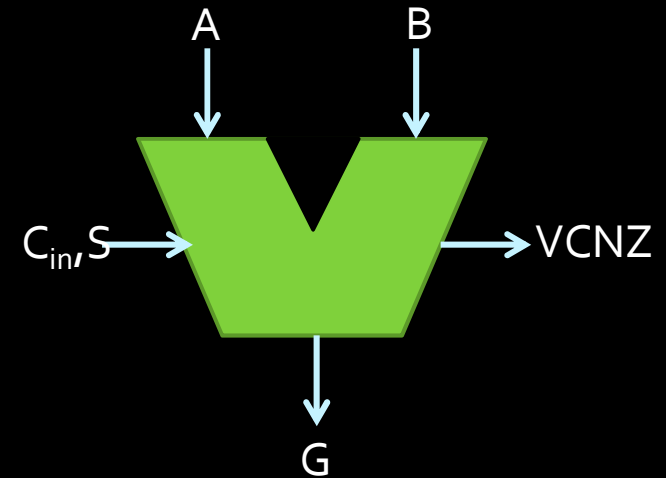
ALU block diagram

- 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`).
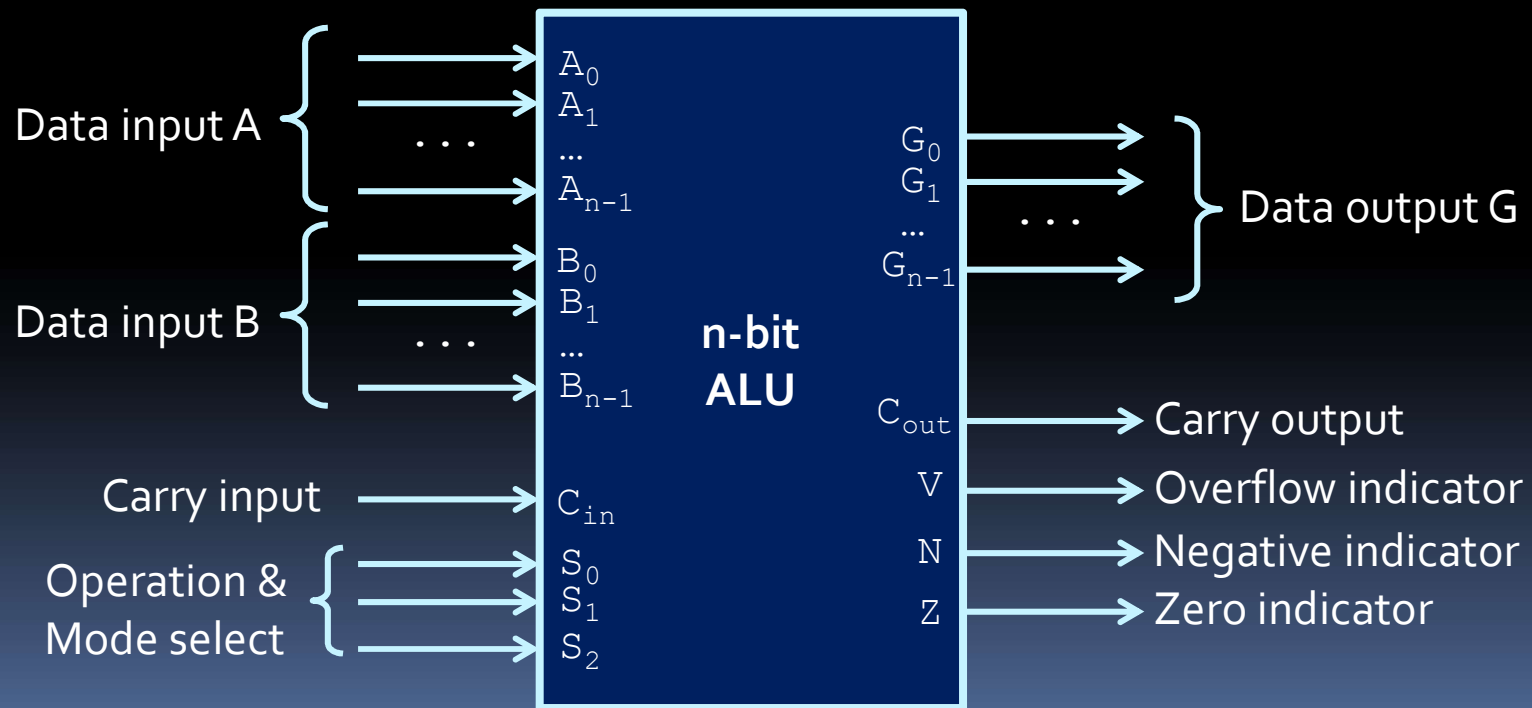
ALU Disclaimer

- 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 one implementation (that you need to learn), but just keep in mind that others are possible as well.

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:
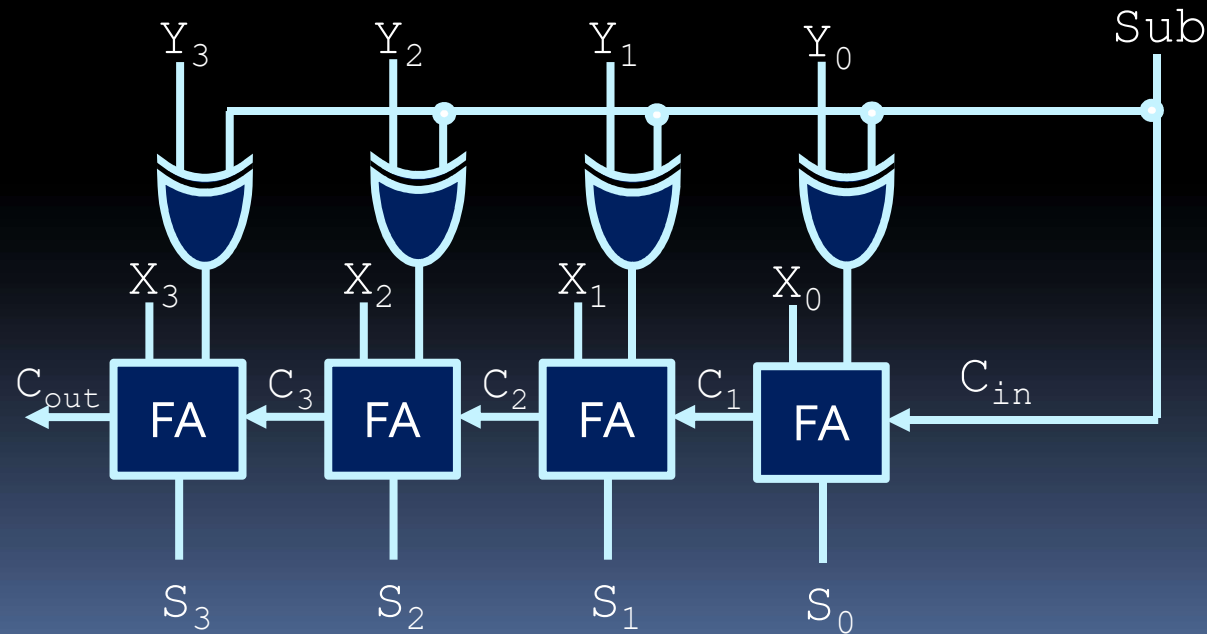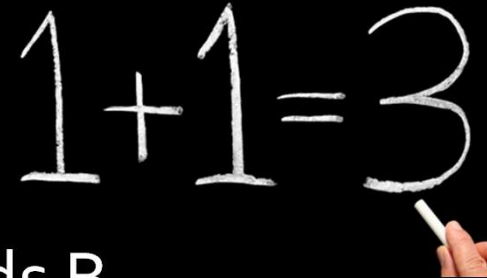
Arithmetic components

$$G = X + Y + C_{in}$$
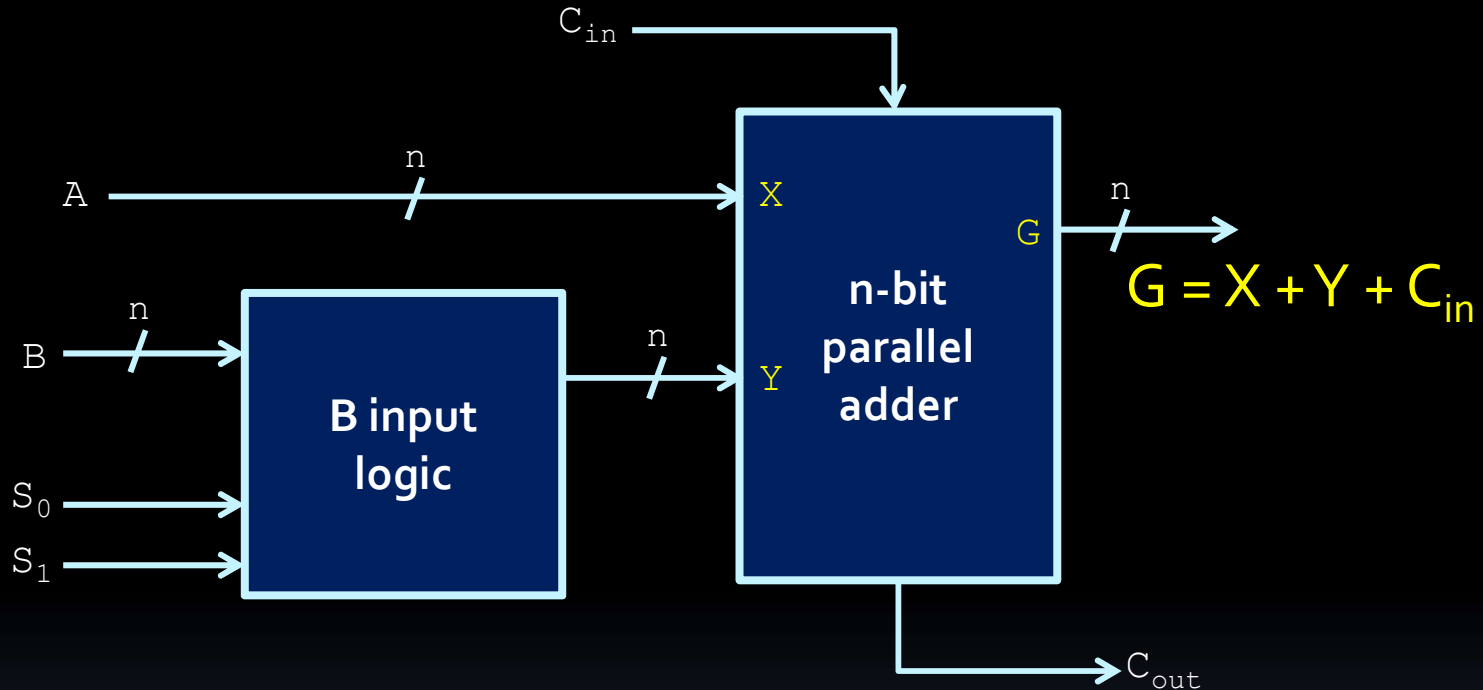
- In addition to addition and subtraction, many more operations can be performed by manipulating what is added to input $\mathbb{A}$, as shown in the diagram above.

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.
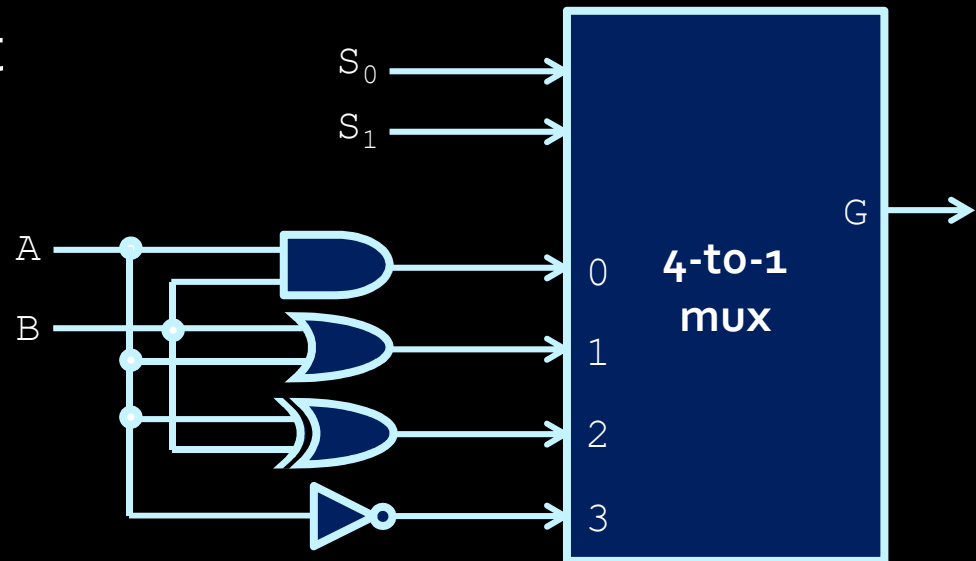
ALU: Arithmetic + Logic
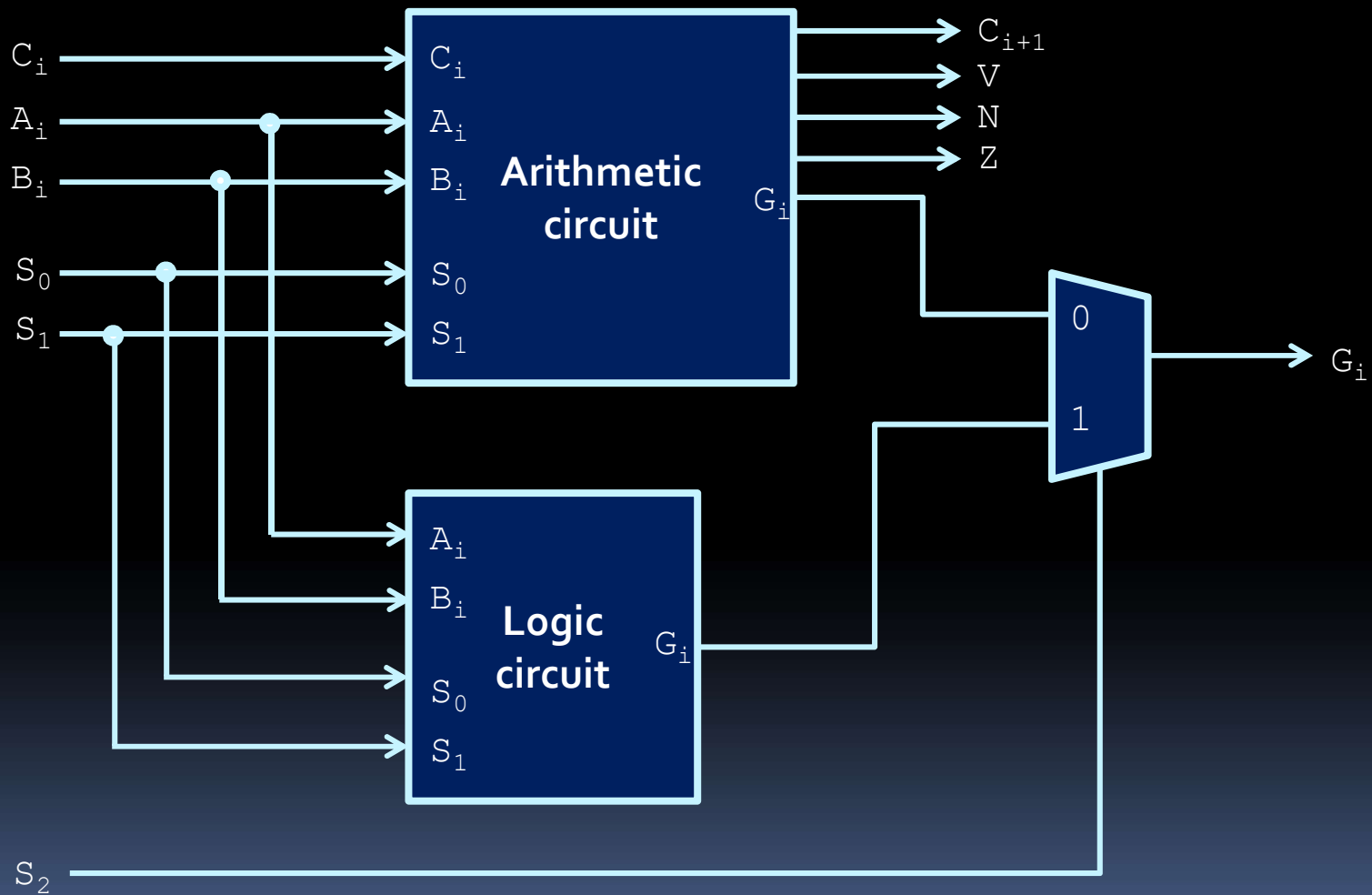
What about multiplication?

- 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…

5*6 (unsigned)

```
   101
x  110
```

```
  10 1
x  110
  ───
  110
```

```
  1 0 1
x   110
  ─────
   110
   000
```

```
  1 01
x  110
  ────
   110
   000
   110
```

```
    101
x   110
  ─────
    110
   000
   110
  ──────
  11110
```

Result: 30
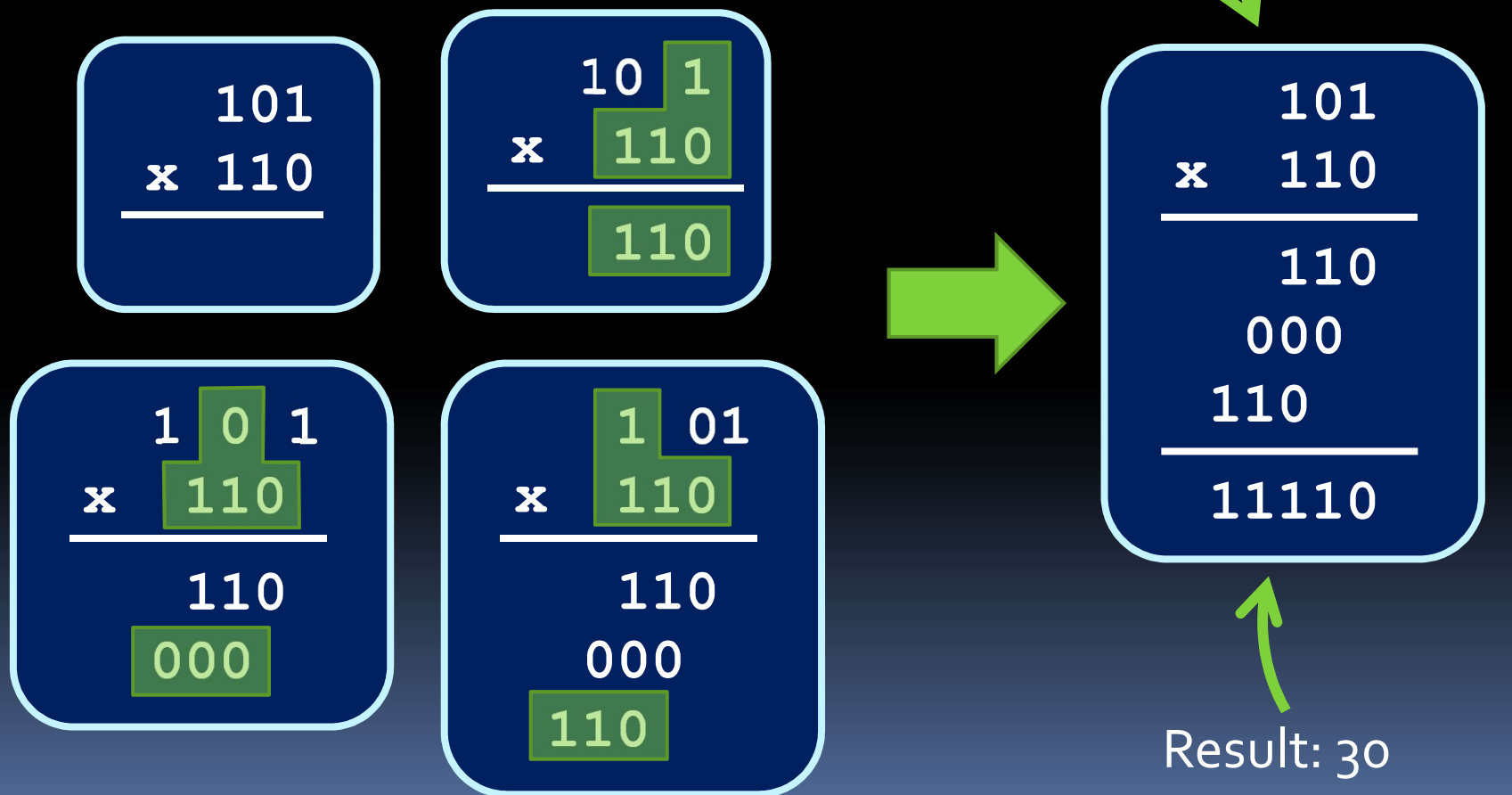
Binary Multiplication

- 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
```
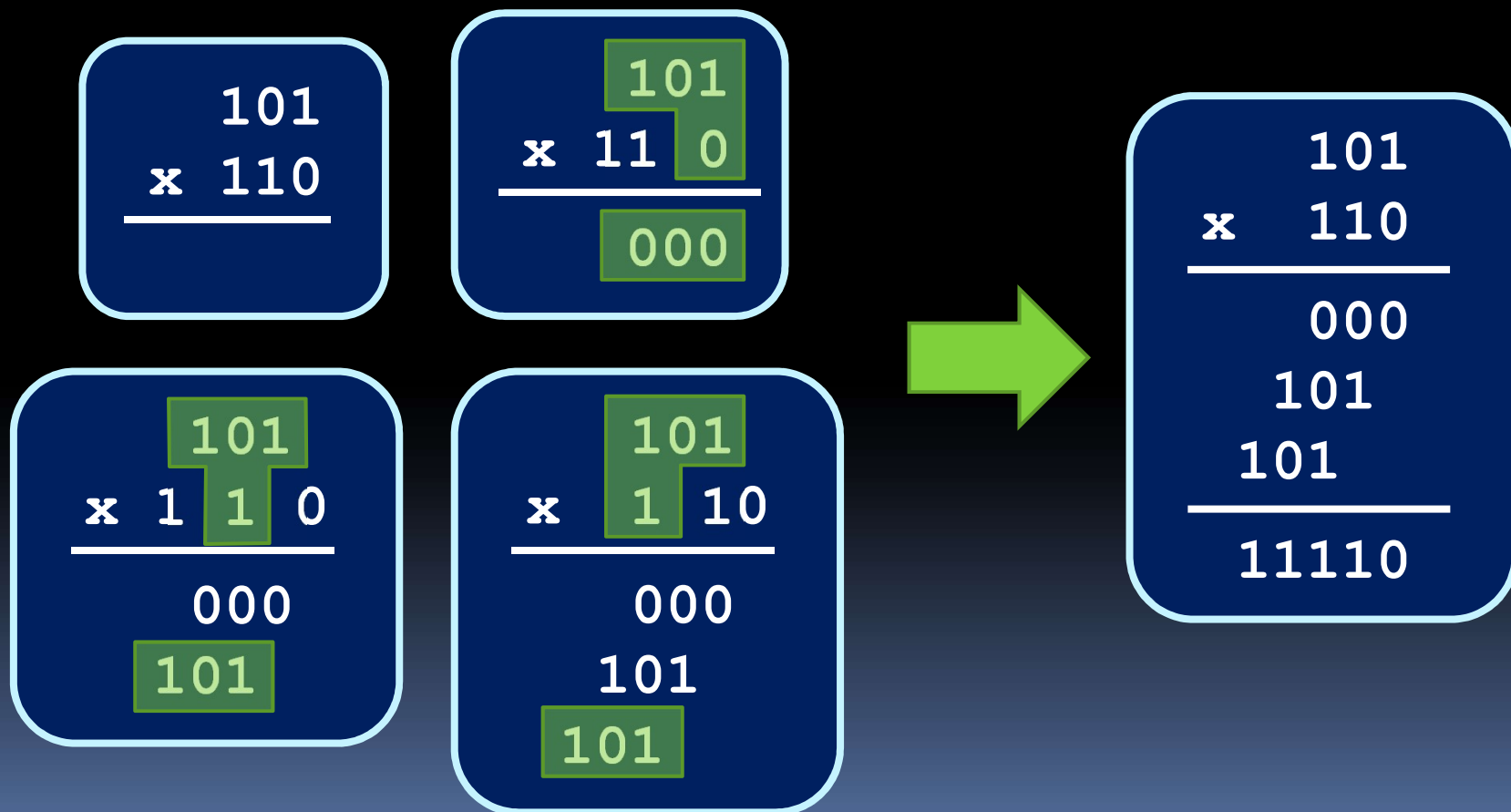
Binary Multiplication

$$\begin{array}{ccccc}
 & a_3 & a_2 & a_1 & a_0 \\
\times & b_3 & b_2 & b_1 & b_0 \\
\hline
\end{array}$$

$$\begin{array}{cccccccc}
 & & & a_3 b_0 & a_2 b_0 & a_1 b_0 & a_0 b_0 \\
 & & a_3 b_1 & a_2 b_1 & a_1 b_1 & a_0 b_1 & \\
 & a_3 b_2 & a_2 b_2 & a_1 b_2 & a_0 b_2 & & \\
a_3 b_3 & a_2 b_3 & a_1 b_3 & a_0 b_3 & & & \\
\hline
p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0
\end{array}$$

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.
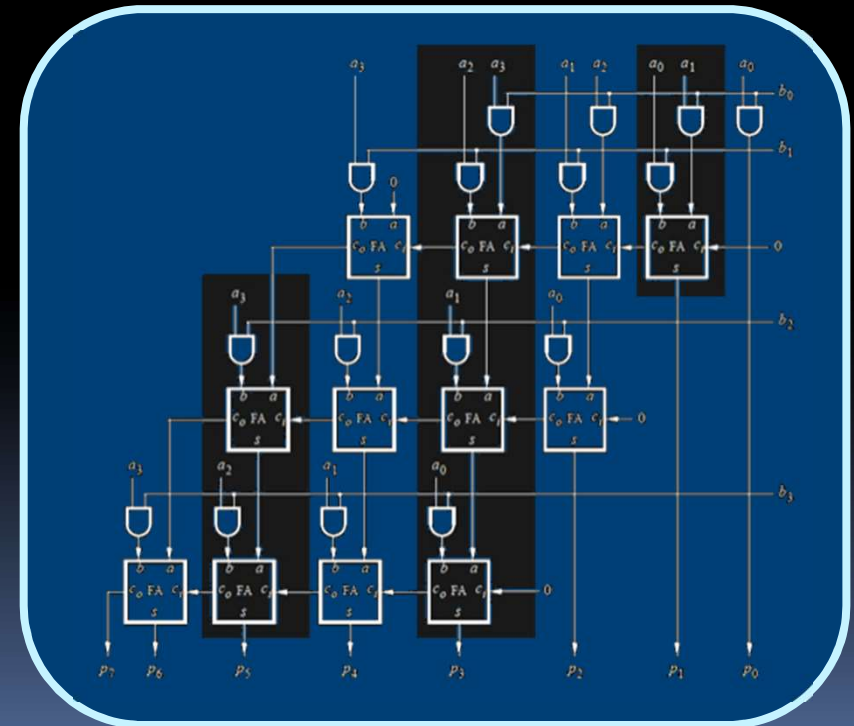
Multiplication

- 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?
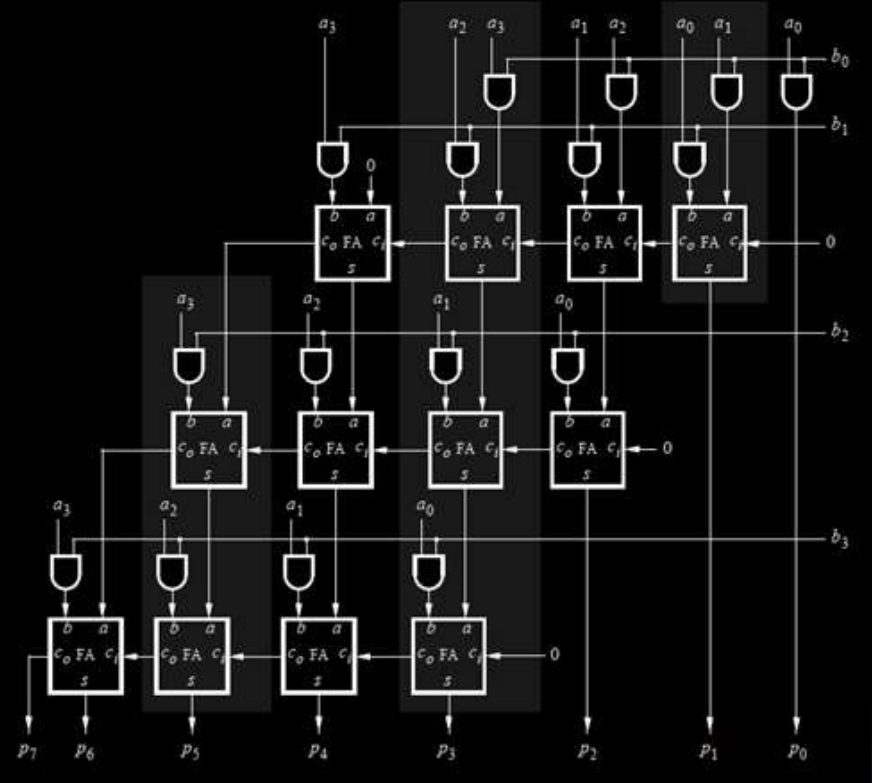
Accumulator circuits

- 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?
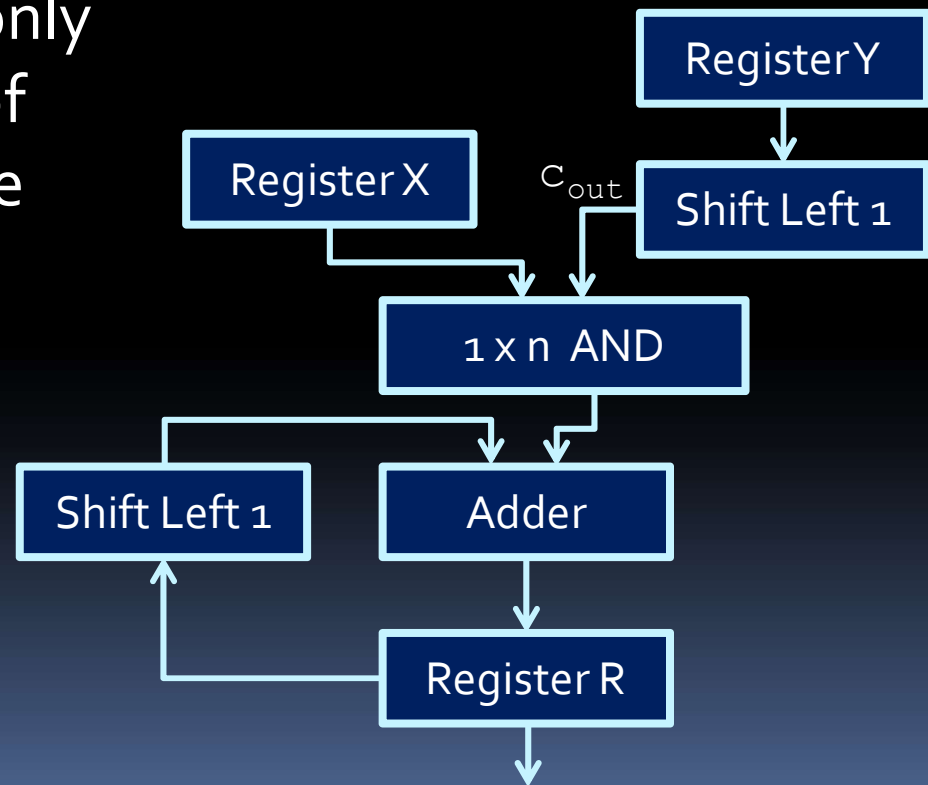
Booth's Algorithm

- 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:
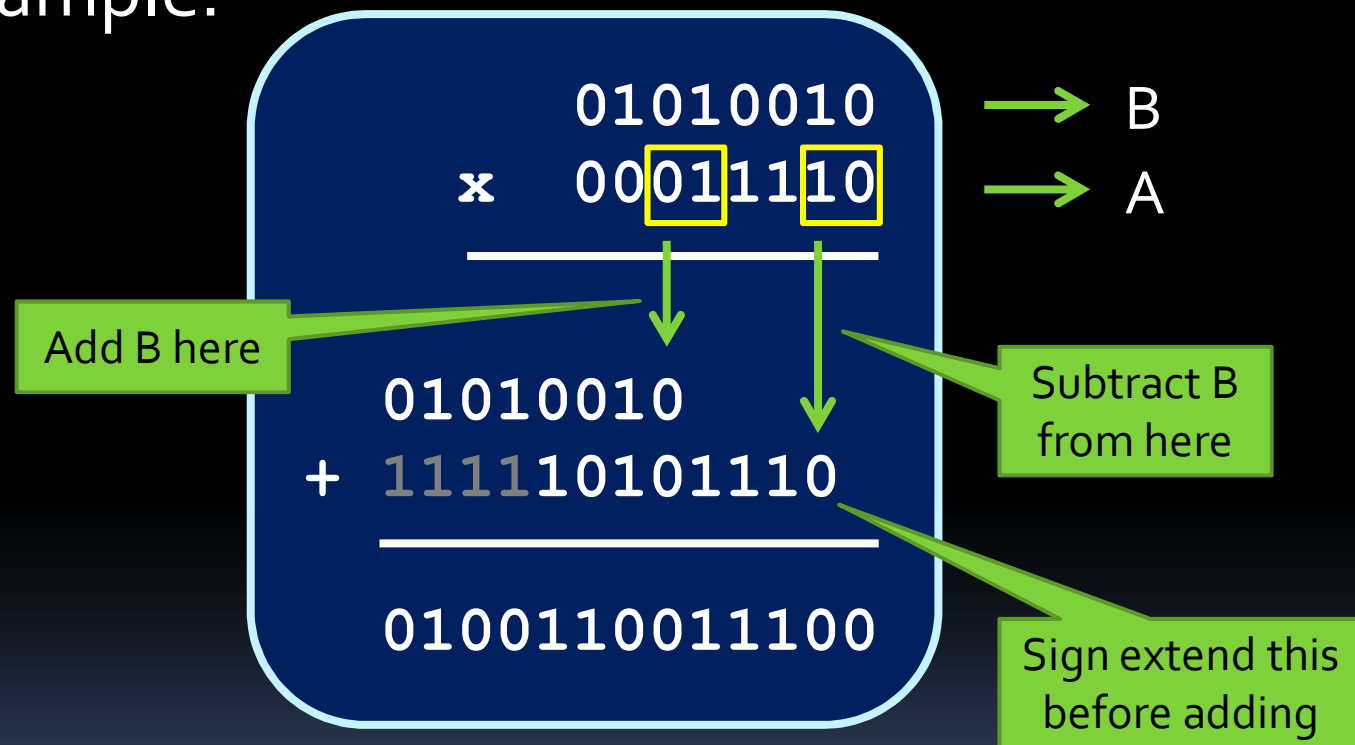
Booth's Algorithm

- 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...

  - <u>Option #3:</u> 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.

    - <u>Solution!</u> 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:

    $$1101 \quad \boxed{10}$$

  - 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 **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.

A Barrel Shifter unit
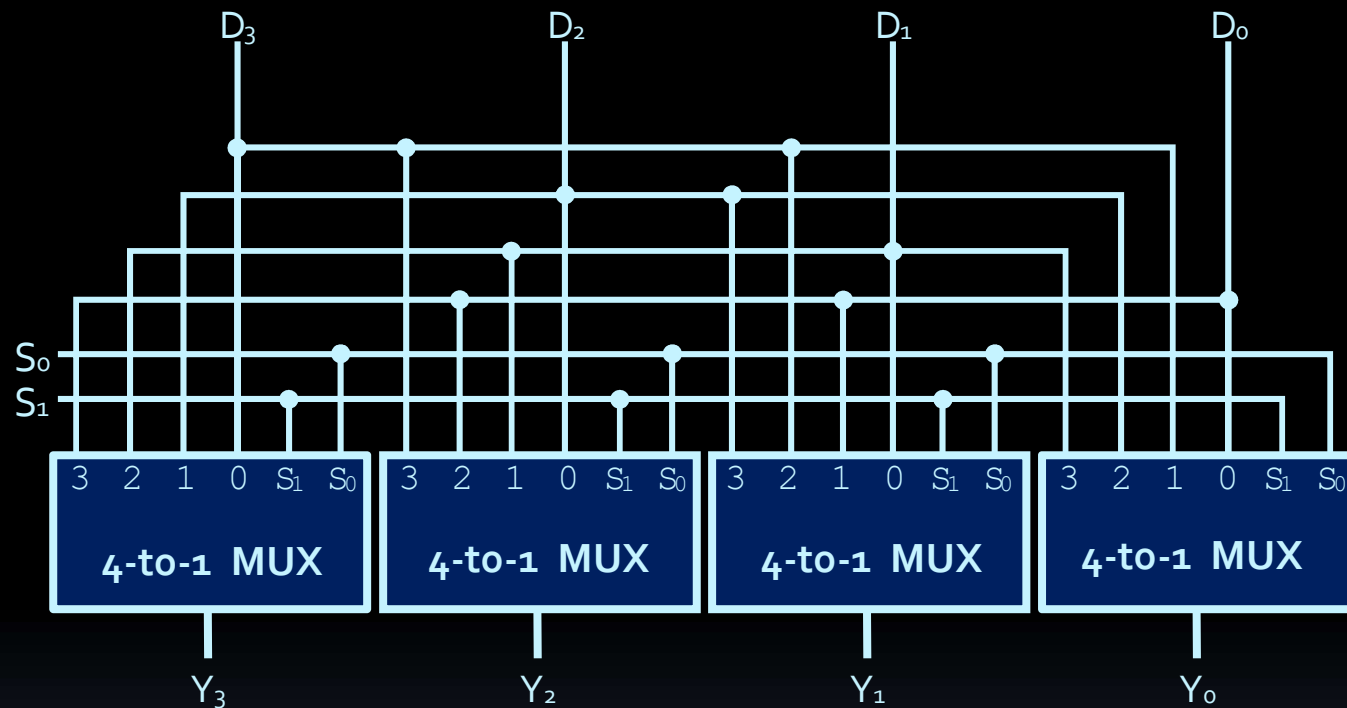
- This barrel shifter **shifts and rotates** D to the left by S bits.
  - If $S_1 S_0$ is 01 => $Y = D_2 D_1 D_0 D_3$
  - If $S_1 S_0$ is 11 => $Y = D_0 D_3 D_2 D_1$
- This is a purely combinational circuit, unlike the shift registers in the lab.

Expanding our view
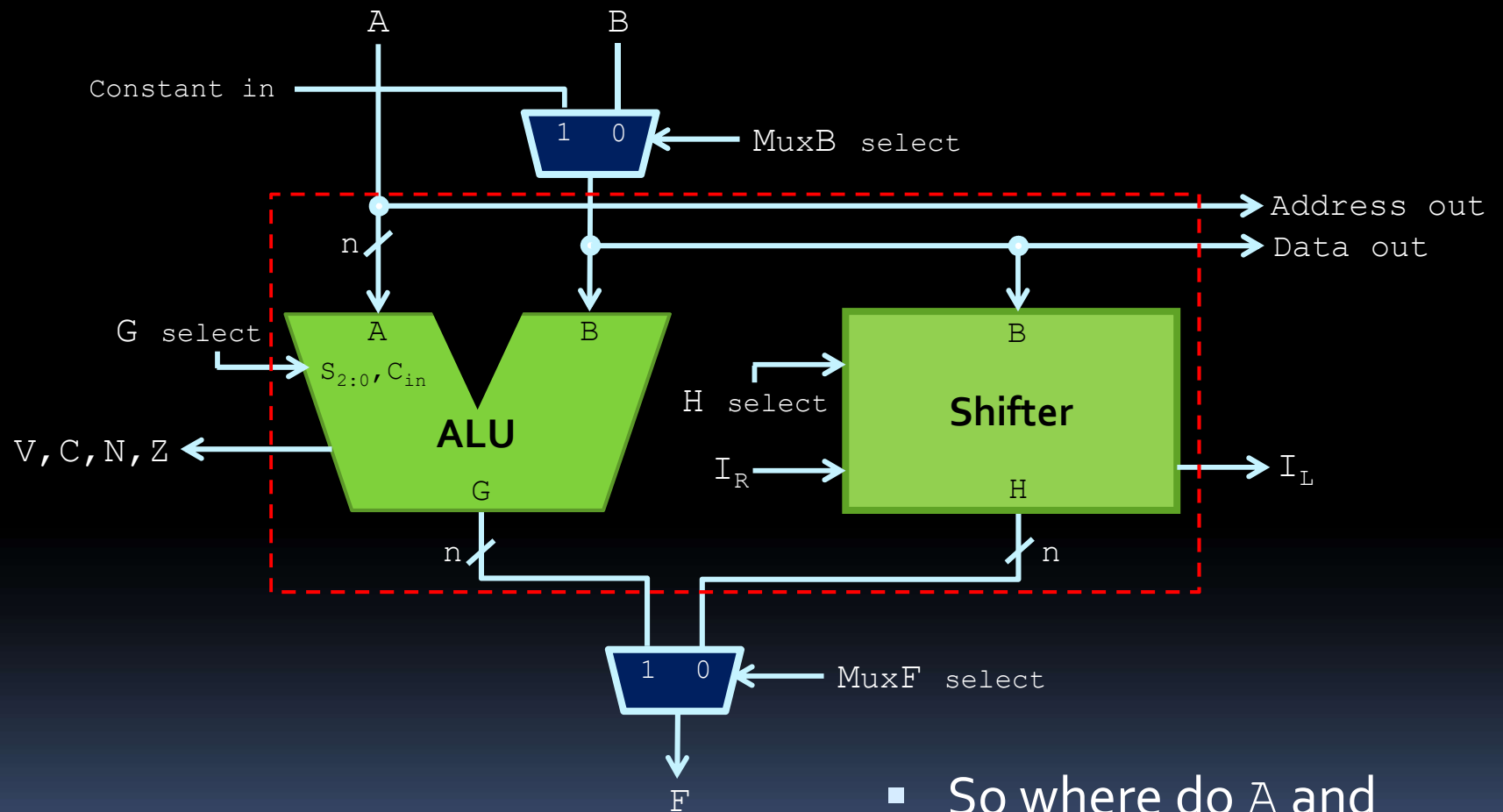
- So where do A and B come from?

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

- Registers as 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)

Register File Functionality

Read/Write

Destination Reg.
(n-bit address)

Data to write

Register A
(n-bit address)

Register B
(n-bit address)

| Register 0 |
| Register 1 |
| Register 2 |
| ... |
| ... |
| Register 2^n |

Register File

Value from
Reg. A

Value from
Reg. B
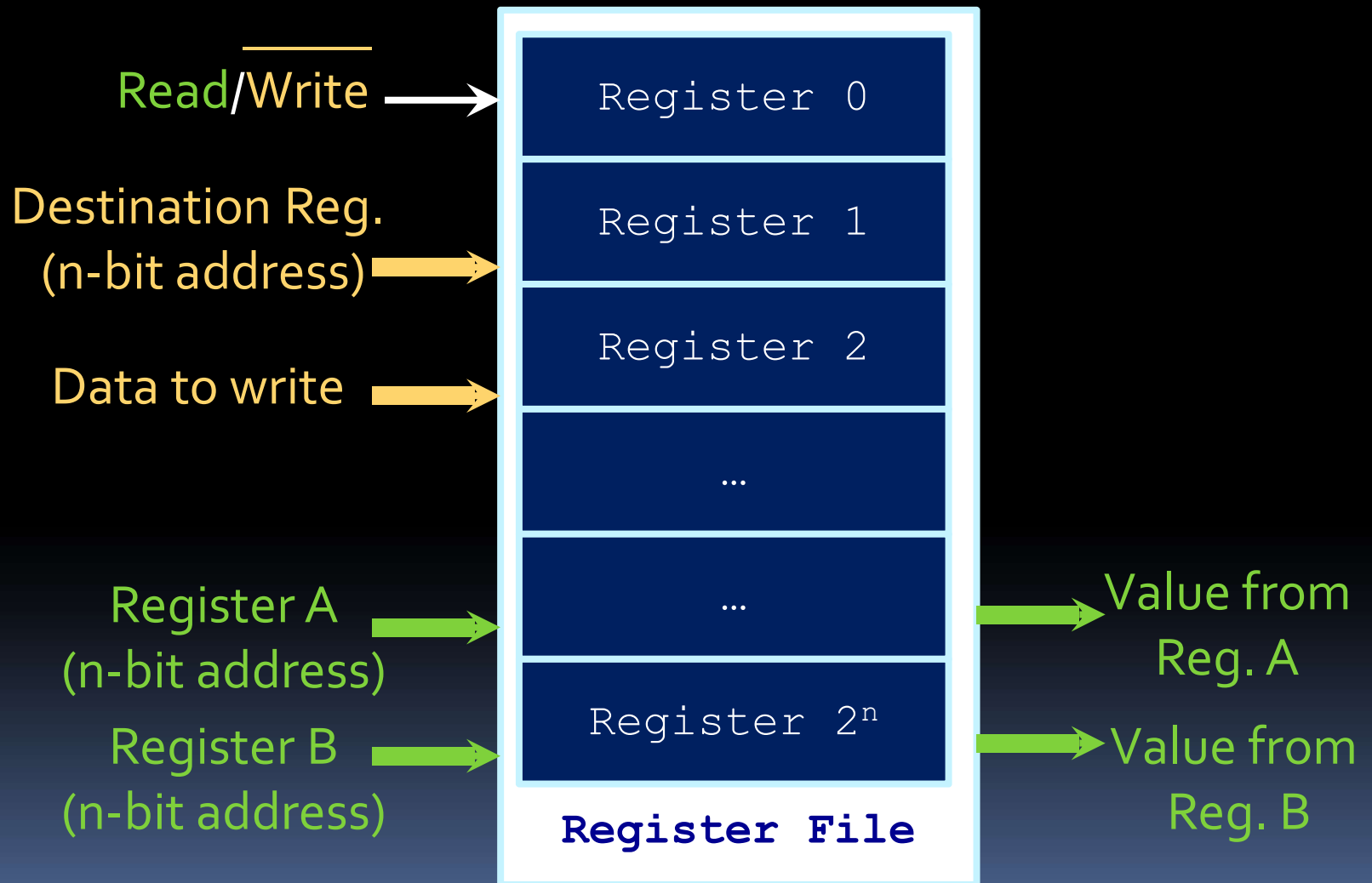
Register File – Write Operation
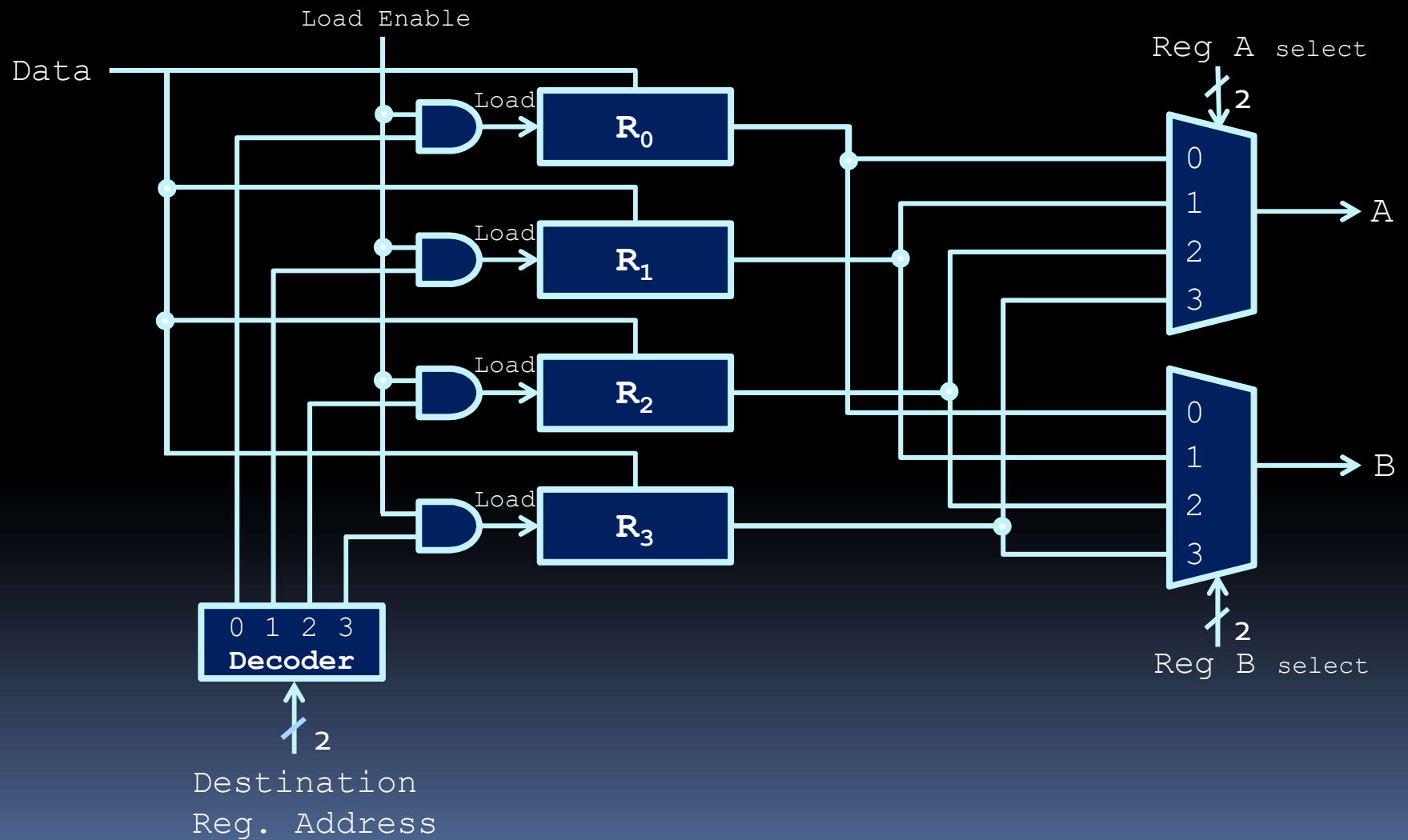
Register File – Write Operation
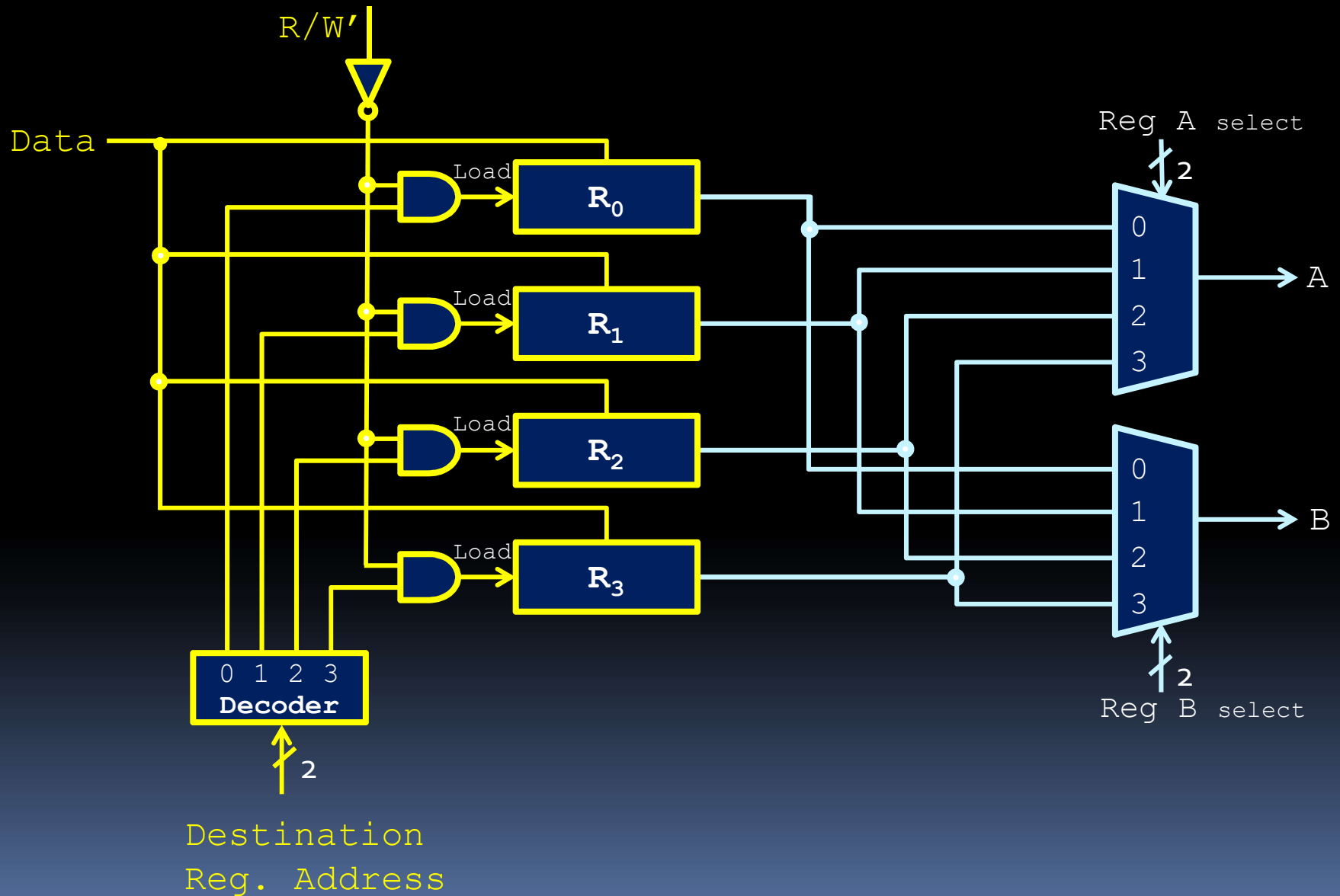
Register File – Read Operation
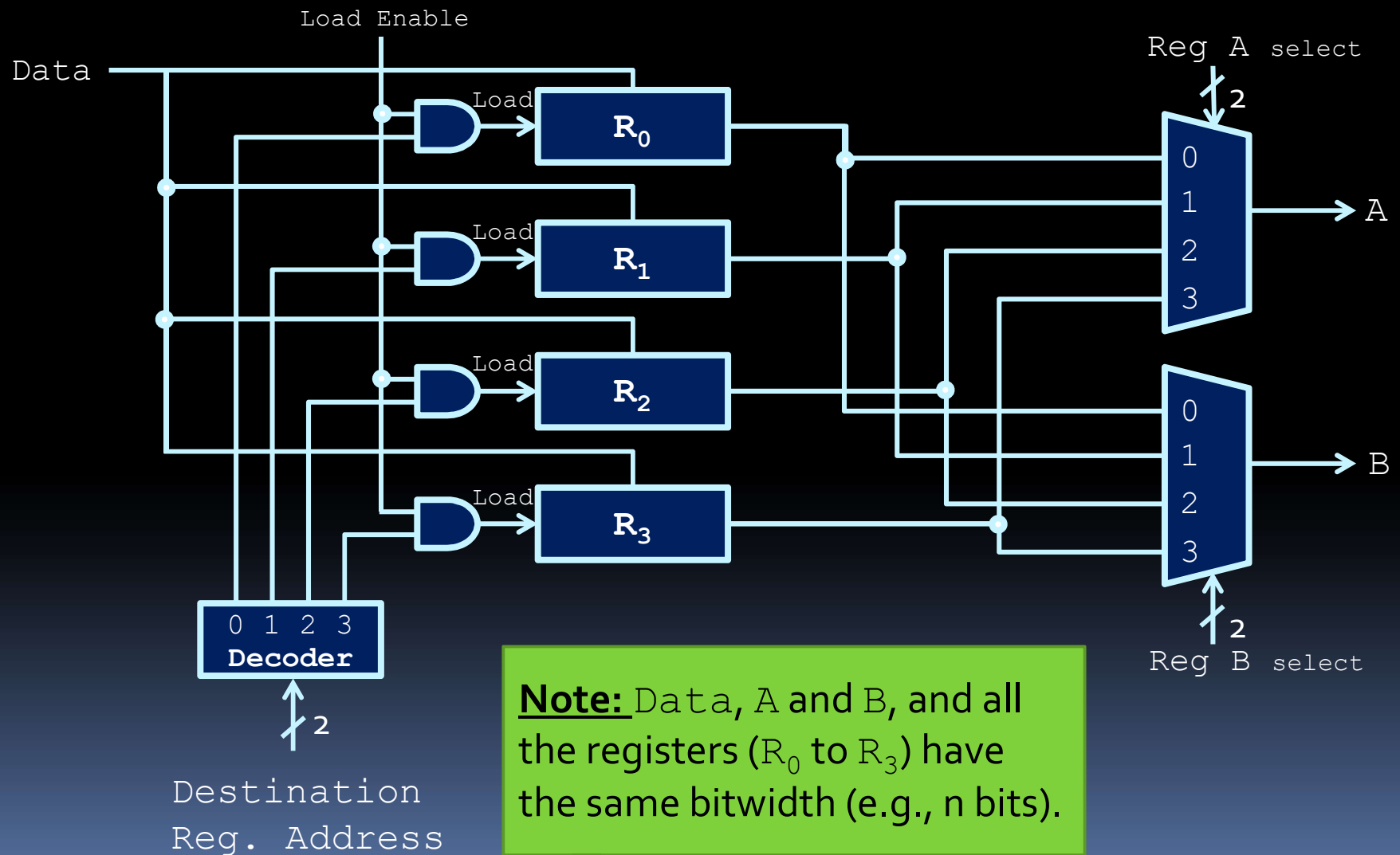
Register File – Read Operation
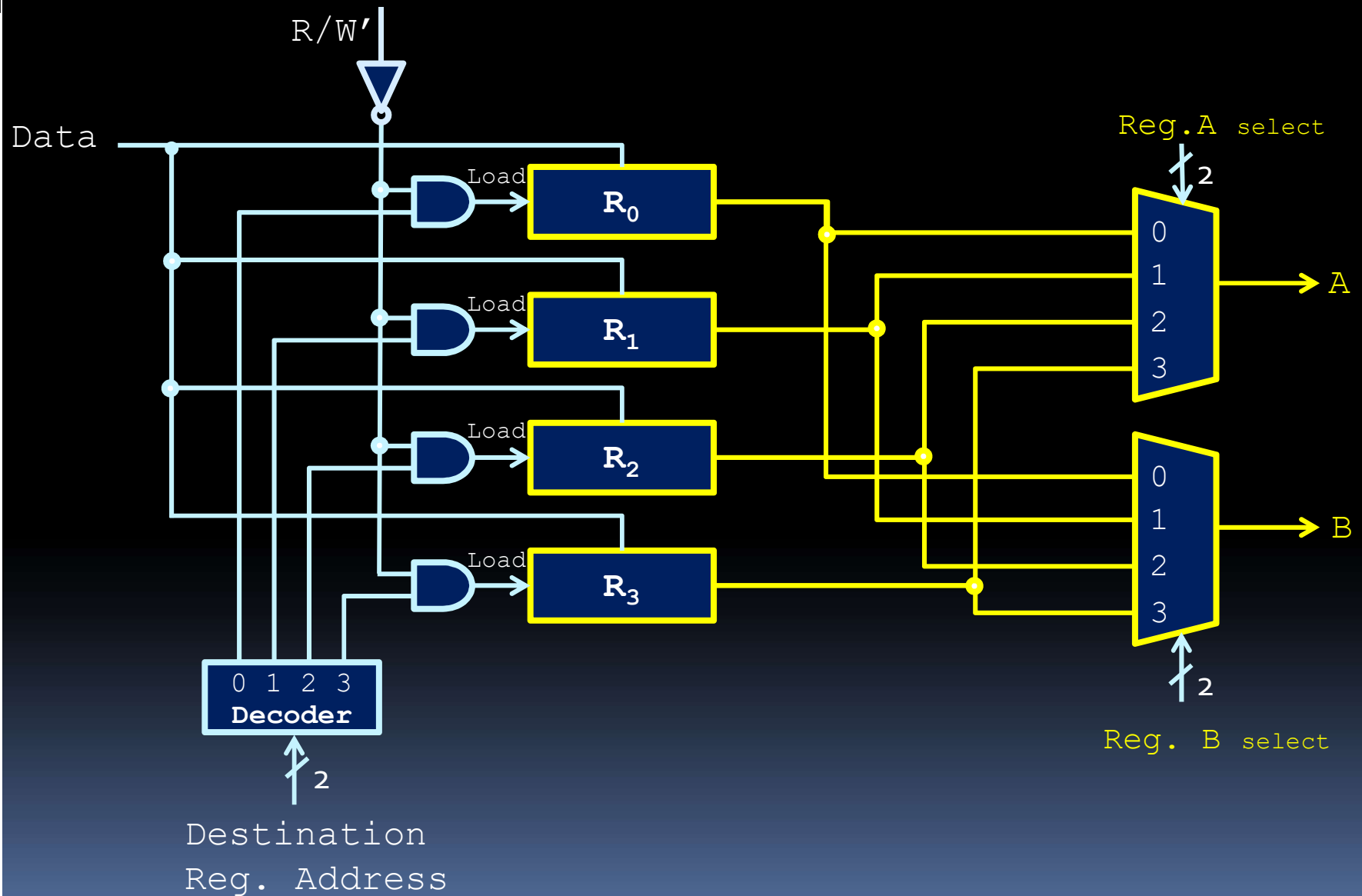
Electronic Memory

- Like register files, main memory is made up of a decoder and rows of memory units.

	Row 0
Address Lines → m → Decoder	**Row 1**
	Row 2
	Row 3
	. . .
	Row 2^m-1

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

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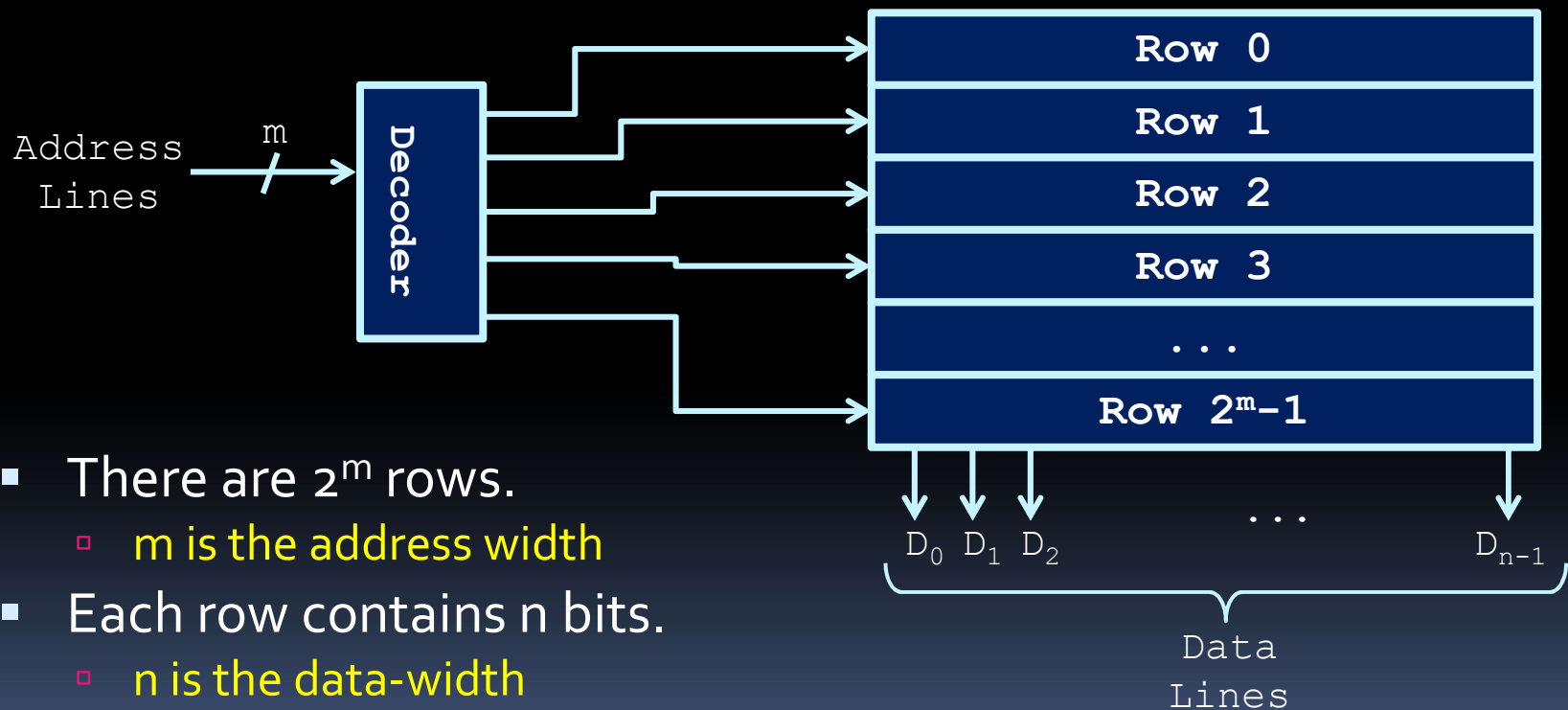
- There are 2^m rows.
  - m is the address width
- Each row contains n bits.
  - n is the data-width
- What's the size of this memory?
  - $2^m * n$ bits => $2^m * n / 8$ Bytes

One-hot decoder

- The decoder takes in the m-bit binary address, and activates a single row in the memory array.

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

Storage cells

- Each row is made of n storage cells.
  - Each cell stores a single bit of information.
- Multiple ways of representing these cells.
  - e.g. RAM cell (know this):   DRAM IC cell (just FYI):
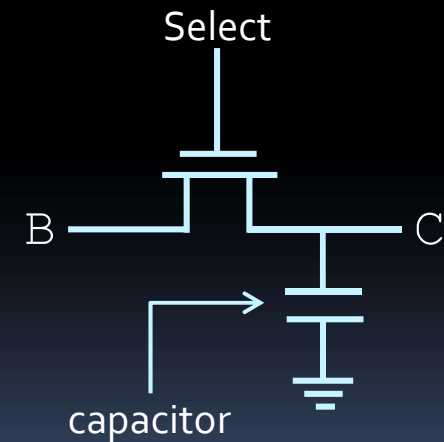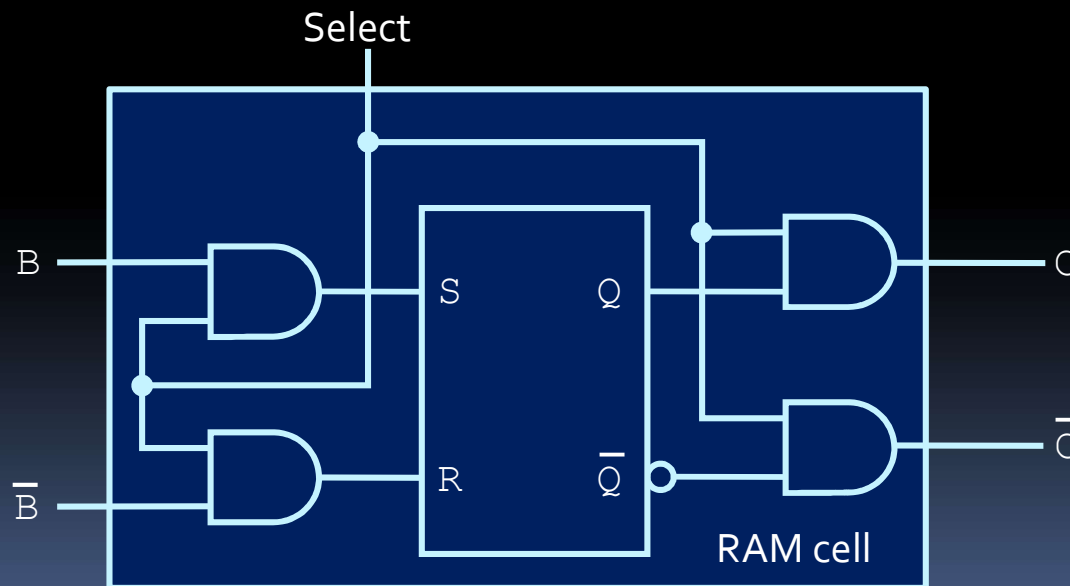
Data Bus

- Communication between components (especially memory) takes place through groups of wires called a bus (or data bus).
  - Multiple components can read from a bus, but only one can write to a bus at a time.
    - Also called a bus driver.
  - Each component has a tristate buffer that feeds into the bus. When not reading or writing, the tristate buffer drives high impedance onto the bus.

Controlling the flow

- Since some lines (buses) will now be used for both input and output, we introduce a (sort of) new gate called the tri-state buffer.

- When WE (write enable) signal is low, buffer output is a "high impedance" signal.
  - The output is neither connected to high voltage or to the ground.

WE

A ——————▷———————— Y
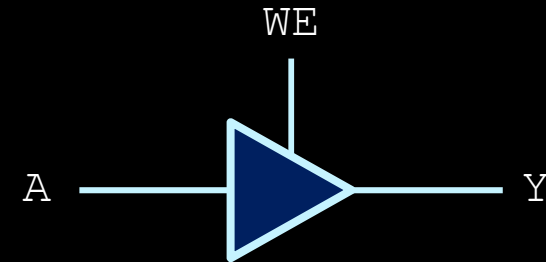
WE	A	Y
0	x	z
1	0	0
1	1	1

Tri-state Buffer Use

Assume `data_out` and `data_in` connect to a memory cell.

R/$\overline{W}$ Memory_Enable

Inside memory unit | Outside memory unit (to processor)

data_out

data

data_in
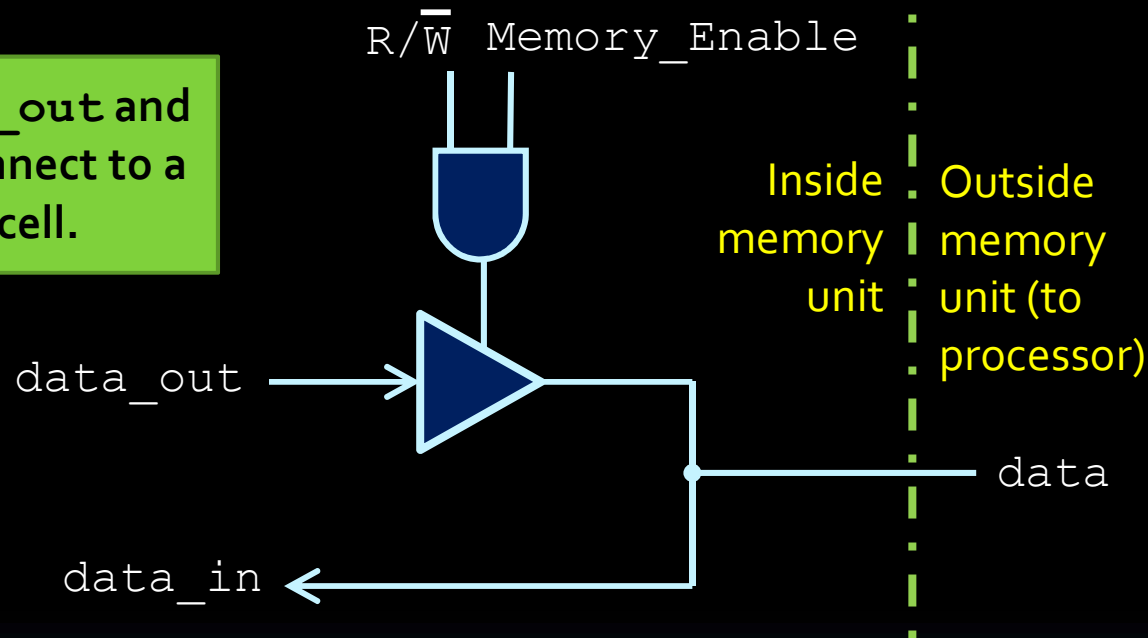
- If R/$\overline{W}$ is 1 **(read)** & `Memory_Enable` is 1
  - `data` behaves as output to processor
- Otherwise if R/$\overline{W}$ is 0
  - `data` behaves as input from processor to memory since the tri-state buffer is disabled.

RAM Memory Interface

- Write Enable (or R/$\overline{\text{W}}$, or some other variation) – Input
  - Memory write: Memory is modified if this signal is high.
  - Memory read: Memory is read if this signal is low.
- Data In – Input
  - The data to write (store in memory) if write-enable is set.
- Data Out – Output
  - The data read from memory if write-enable is 0.

- Additional signals needed for memory units:
  - Address Port - Input
    - Takes in m bits to address 2^m memory locations.
  - Chip Enable - Input
    - Activates the memory chip
  - Output Enable - Input
    - Accompanies data read
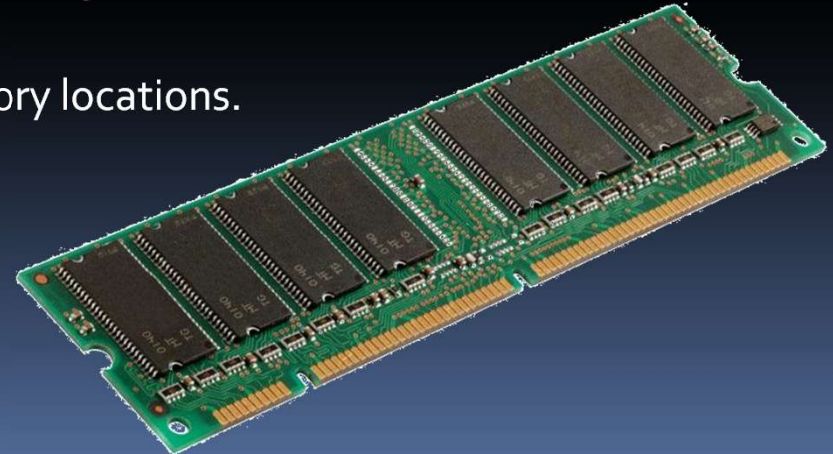
Example: Asynchronous SRAM Interface

Address
(n-bit) →

$\overline{CE}$ →

Read/$\overline{Write}$ →

$\overline{OE}$ →

SRAM

Data
(m-bit) ↔
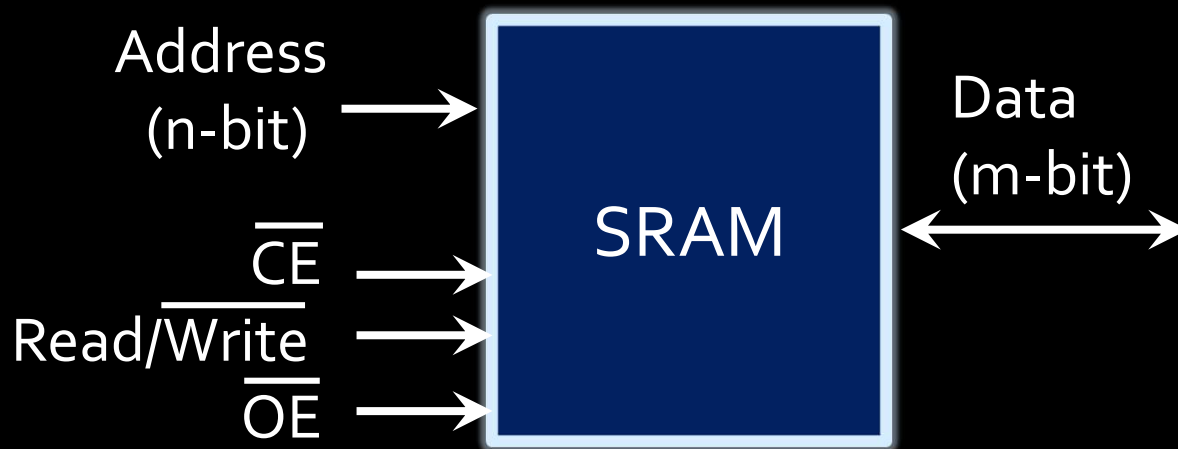
Chip Enable ($\overline{CE}$)	Read/$\overline{Write}$	Output Enable ($\overline{OE}$)	Access Type
0	0	1	SRAM Write
0	1	0	SRAM Read
1	X	X	SRAM not enabled

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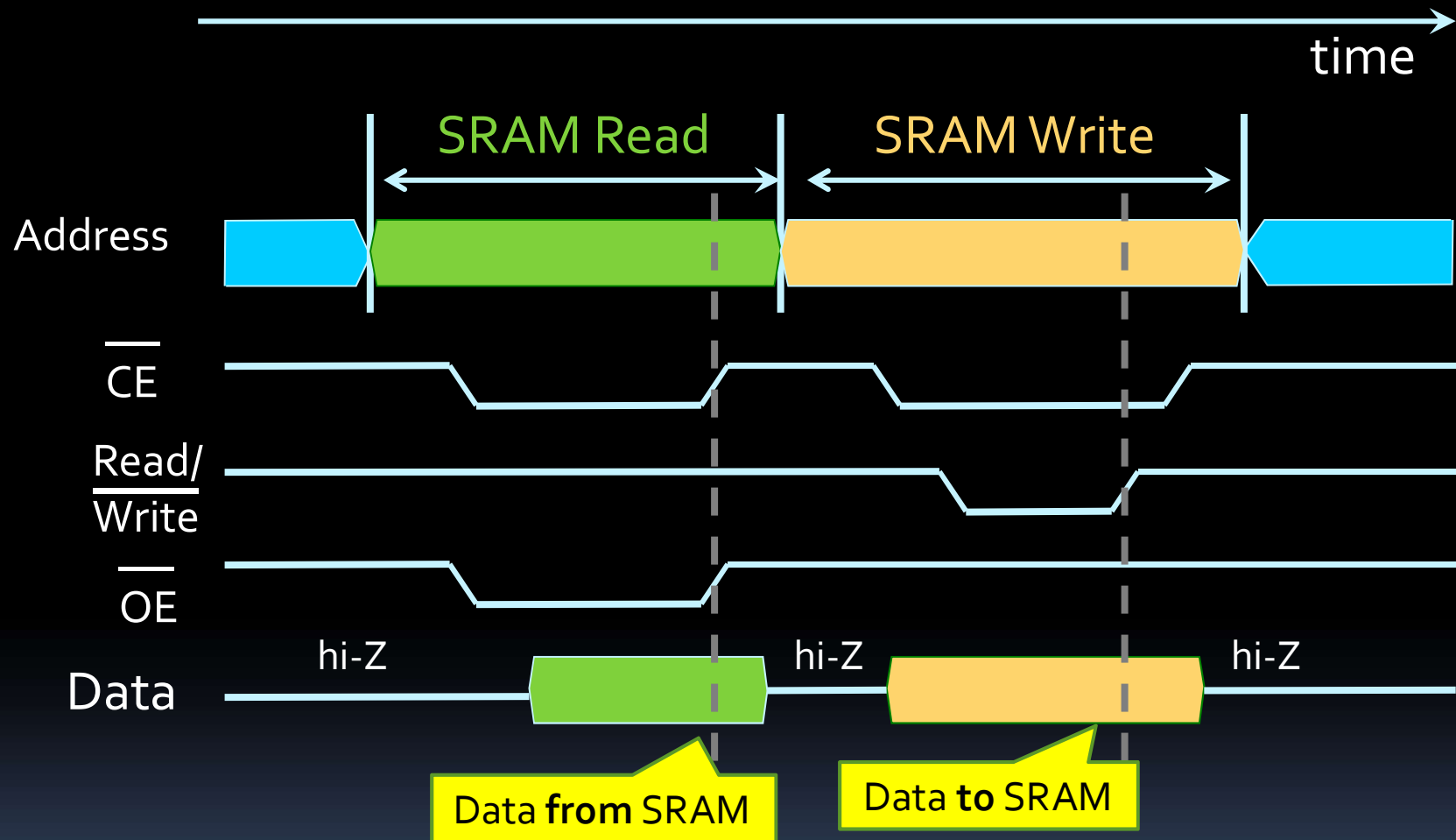
- Each memory read and write is done in stages.
- Each stage takes a certain amount of time.

Reading From Memory – Timing Constraints
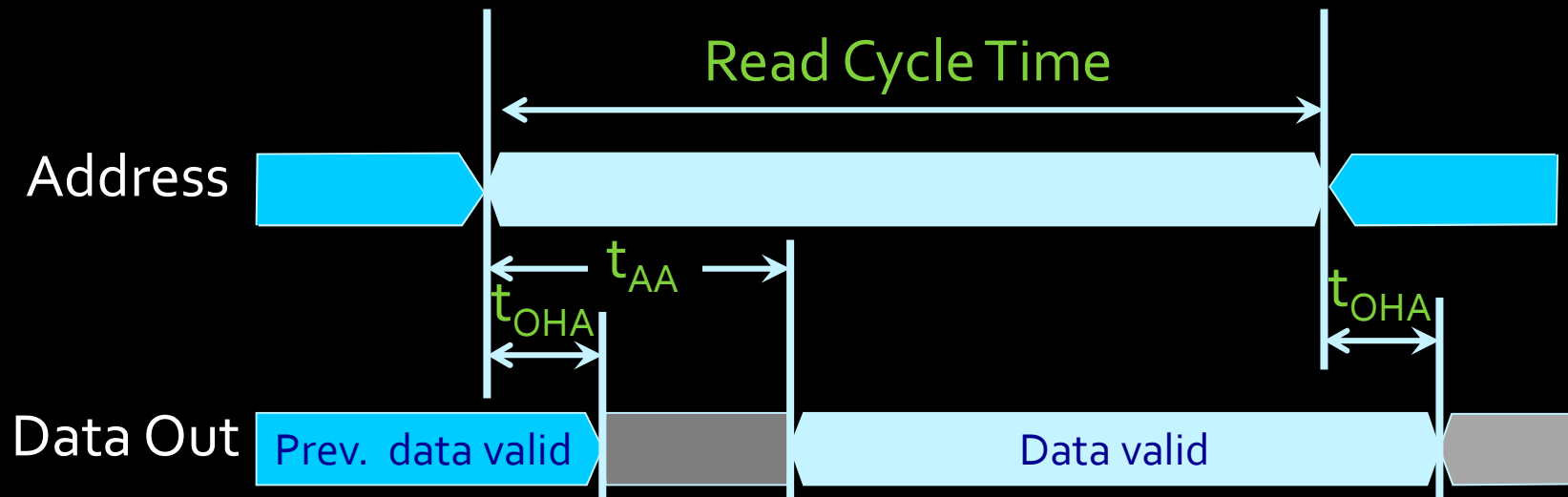
- t_{AA} = Address Access time
  - Time needed for address to be stable before reading data values.

- t_{OHA} = Output Hold time
  - Time output data is held after change of address.

Writing To Memory – Timing Constraints
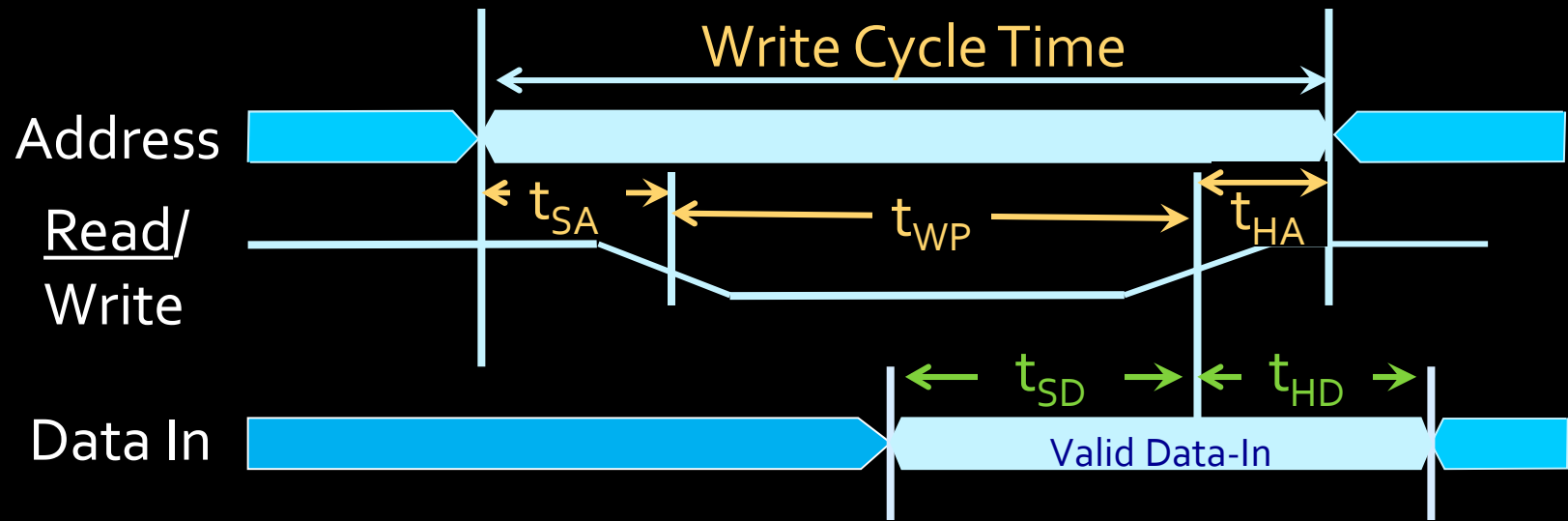
- t_{SA} = **Address Setup Time**
  - Time for address to be stable before enabling write signal.
- t_{HA} = **Address Hold Time**
  - Time for address to be stable before enabling write signal.
- t_{WP} = **Write Pulse Width**

- 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 vs registers

- Memory houses most of the data values being used by a program.
- Registers are more local data stores, meant to be used to execute an instruction.
  - Registers are not meant to host memory between instructions (like scrap paper for a calculation).
  - Exception is the stack pointer register, which is sometimes in the same register file as the others.

Using RAM template

- Create new Verilog file in Quartus.

- Right click on the empty file and select "Insert Template"

  - Verilog HDL -> Full Designs -> RAMs and ROMs -> Single Port Ram. Insert and close.

RAM Interface (Show template)

```verilog
module single_port_ram
 #(parameter DATA_WIDTH=8, parameter ADDR_WIDTH=6)
(
        input [(DATA_WIDTH-1):0] data,
        input [(ADDR_WIDTH-1):0] addr,
        input we, clk,
        output [(DATA_WIDTH-1):0] q
);

        // Declare the RAM variable
        reg [DATA_WIDTH-1:0] ram[2**ADDR_WIDTH-1:0];

        // Variable to hold the registered read address
        reg [ADDR_WIDTH-1:0] addr_reg;

        always @ (posedge clk)
        begin
                // Write
                if (we)
```

Processor datapath diagram
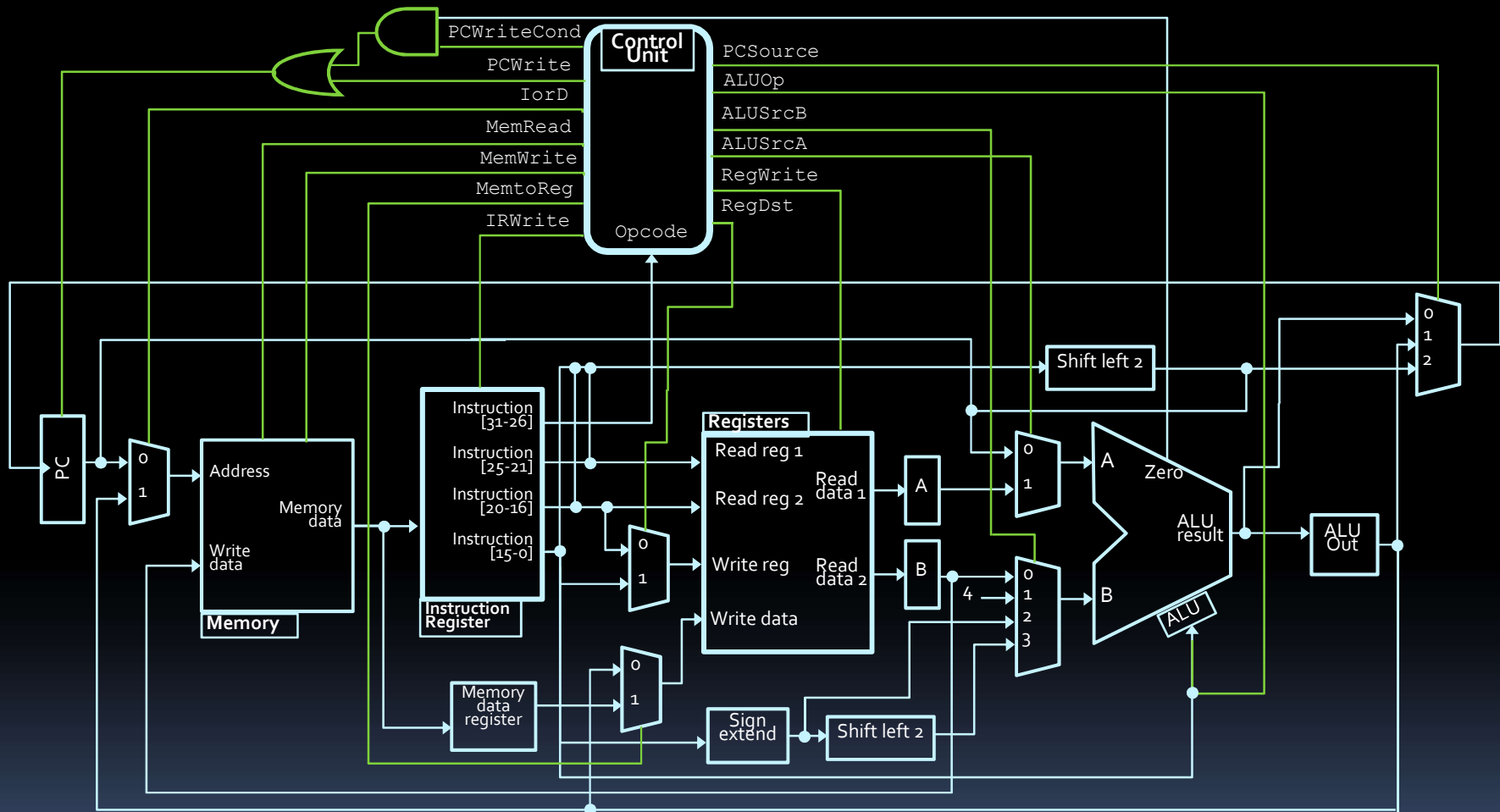
- What controls the memory, registers and ALU?

The Control Unit

The Processor Datapath

- The datapath of a processor is a description/illustration of how the data flows between processor components during the execution of an 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.

The Control Unit

- All datapath operations need to determine:
  - Where the data is coming from (the source),
  - Where it's going to (the destination), and
  - How the data is being processed (the operation).
- How does the control unit know what operation to perform?
  - It gets information from an instruction.
  - This instruction also passes other information about the operation to the rest of the processor.
  - The control unit is responsible for loading the next instruction to run, after completing the current one.

Data sources and destinations
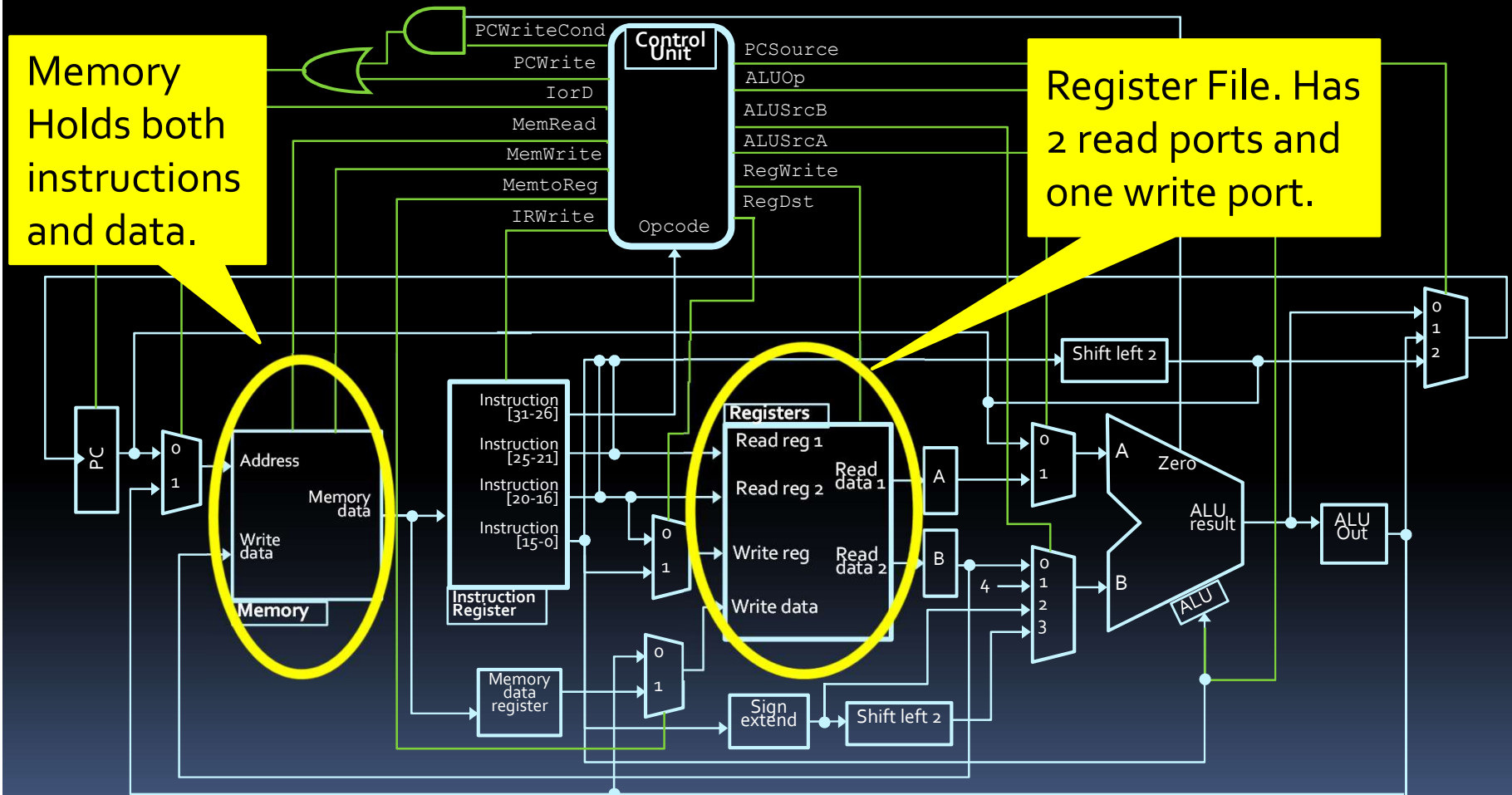
How Instructions Work

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

- 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

- To know what signals to send to the datapath, the control unit is continually performing the following set of 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.

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.
- But how does the PC get updated?

Updating The Program Counter

- Usually instructions are executed in sequential order (i.e., one after the other).
  - We assume byte-addressable memory (i.e., memory where every byte has its own unique address).
  - Also assume that instructions are 32 bits long (i.e. 4 bytes, where 1 byte = 8 bits).
- 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 output of the ALU can write to the PC.
  - Branches, jumps and function calls are executed this way.
- We will come back to these special instructions later, after talking more about decoding instructions.

Decoding Instructions

- Let's say we've fetched this 4B (32-bit) instruction:

  ```
  00000000 00000001 00111000 00100011
  ```

  What does it mean?

- 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 ddddd000 00100011
```

```
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** takes 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
```
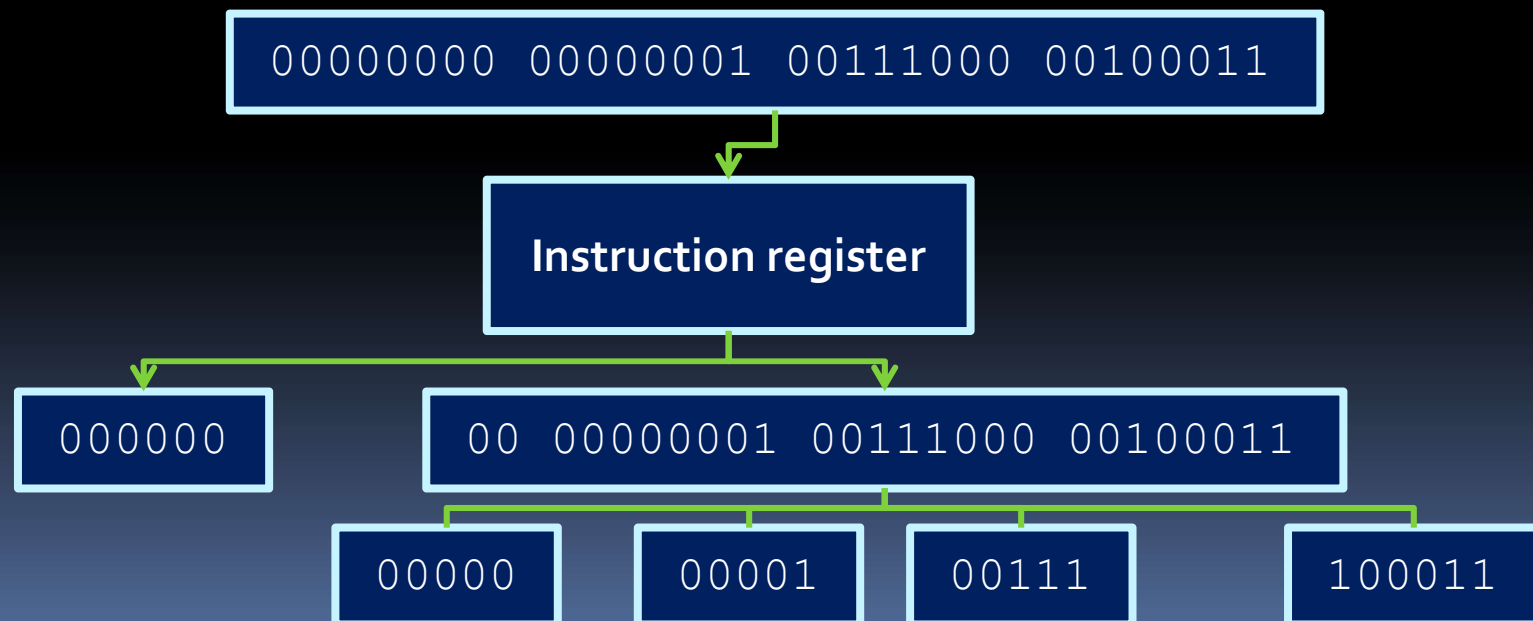
Opcodes

- The first six digits of the instruction (the opcode) are sent to the control unit to signal the operation to perform.

  - For register operations ("R-type" instructions, marked in yellow), the opcode is actually `000000`.

  - In those cases, the last six digits (listed in the table here) further specify the function.

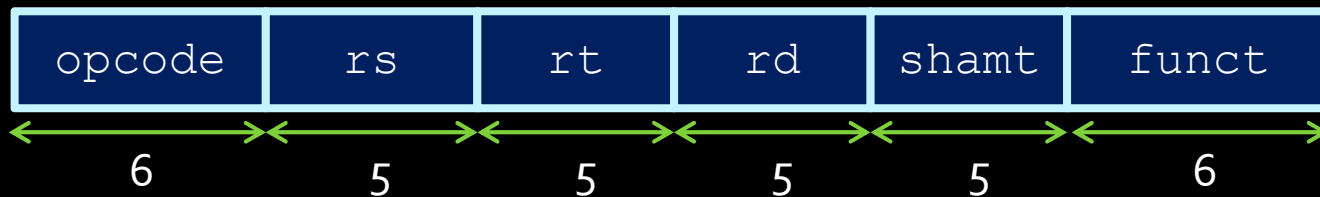| Instruction	Op/Func	Instruction	Op/Func
add	100000	srav	000111
addu	100001	srl	000010
addi	001000	srlv	000110
addiu	001001	beq	000100
div	011010	bgtz	000111
divu	011011	blez	000110
mult	011000	bne	000101
multu	011001	j	000010
sub	100010	jal	000011
subu	100011	jalr	001001
and	100100	jr	001000
andi	001100	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
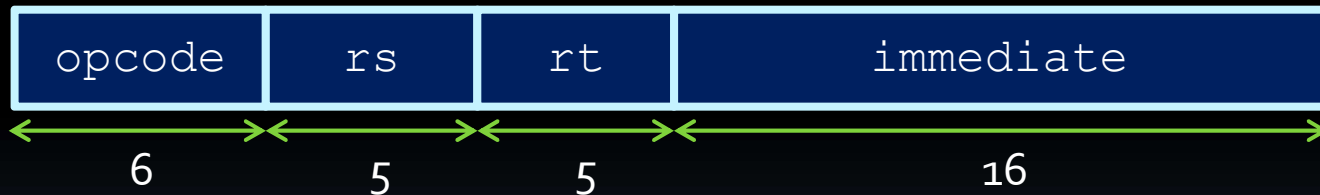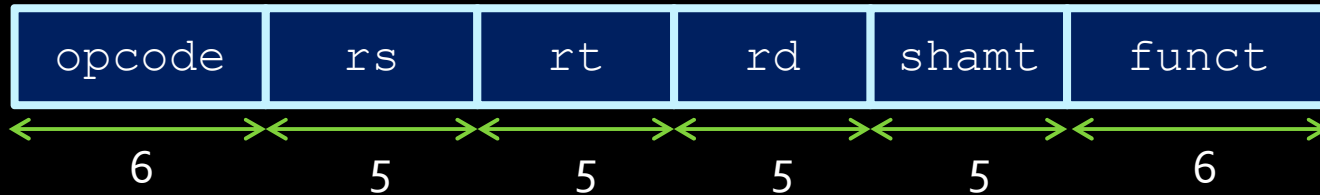
- Short for "register-type" instructions.
  - Because they operate on the registers, naturally.
- These instructions have fields for specifying up to three registers and a shift amount.
  - Three registers: two source registers (`rs` & `rt`) and one destination register (`rd`).
  - A field is usually 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
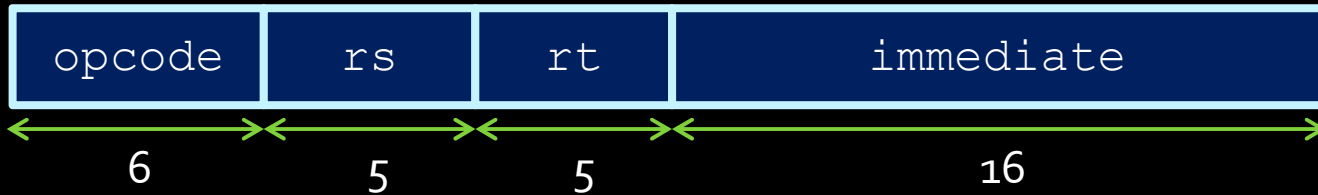
- These instructions have a 16-bit immediate field.
- This field a constant value, which is used for:
  - an immediate operand,
  - a branch target offset, or
  - a displacement for a memory operand.
- For branch target offset operations, the immediate field contains the signed difference between the 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

opcode	address
6	26

- 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).
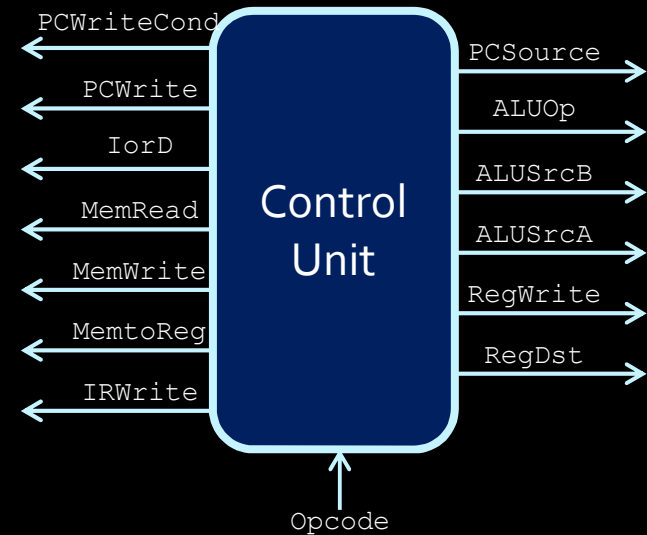
Back to the Control Unit

- These instructions 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?
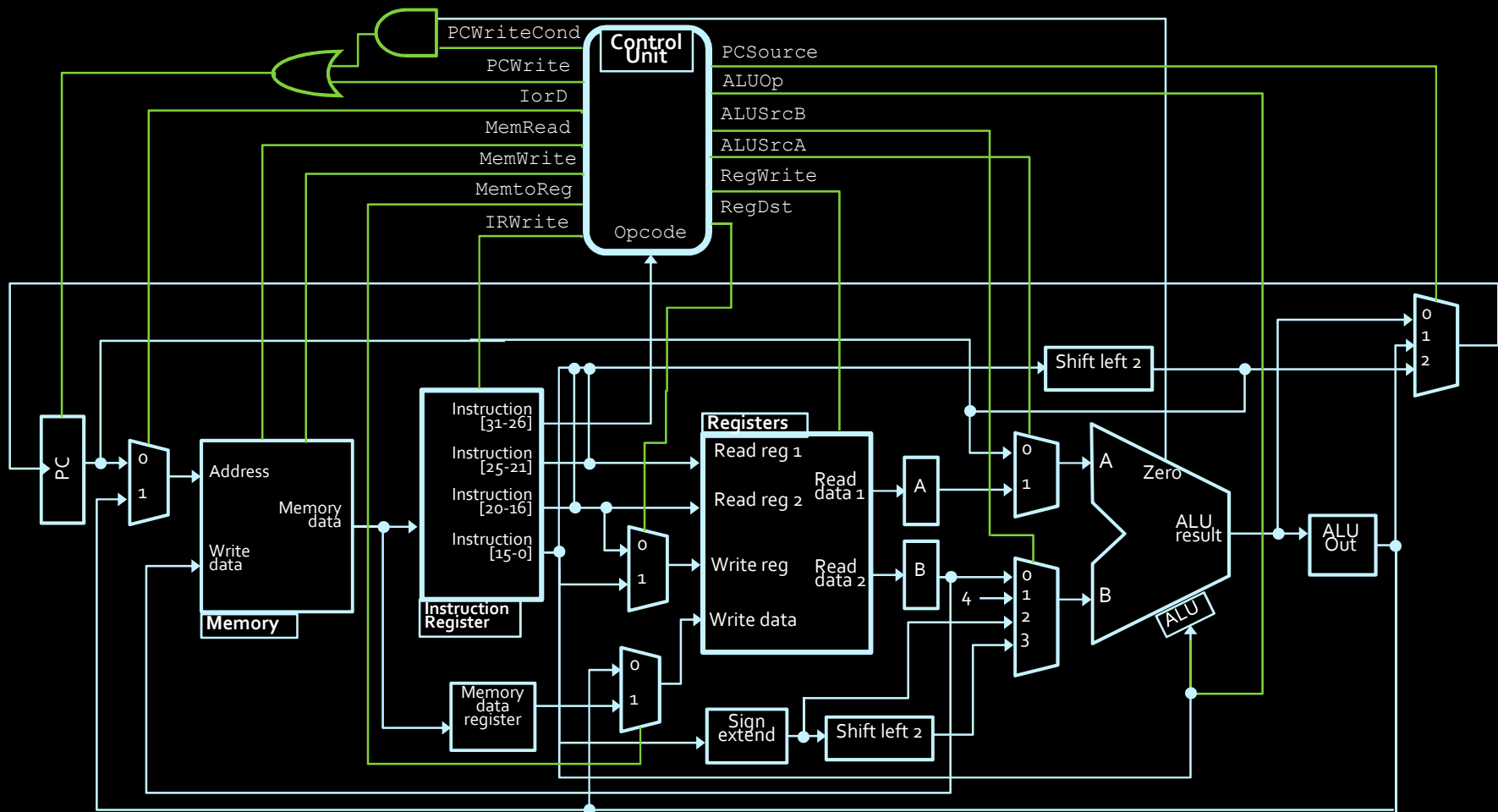
Control unit signals

- The control unit takes in the opcode from the current instruction, and sends signals to the rest of the processor.

Control Unit

Inputs (left): PCWriteCond, PCWrite, IorD, MemRead, MemWrite, MemtoReg, IRWrite

Outputs (right): PCSource, ALUOp, ALUSrcB, ALUSrcA, RegWrite, RegDst

Bottom input: Opcode

- Within the control unit is a finite state machine that 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.

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

Control unit signals

- `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.

More control unit signals

- **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 instruction

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

  - PCWrite = 0
  - PCWriteCond = 0
  - IorD = X
  - MemWrite = 0
  - MemRead = 0
  - MemToReg = 0
  - IRWrite = 0

  - PCSource = X
  - ALUOp = 001 (add)
  - ALUSrcA = 1
  - ALUSrcB = 10
  - RegWrite = 1
  - RegDst = 0

Intro to Machine Code

- Instructions are 0s and 1s that don't make sense to us, but make sense to the processor.

- Remember: operations in the processor are performed as follows.
  - The instruction in the instruction register is decoded according to the opcode (in the first 6 bits) .
  - The control unit then sends a sequence of signals to the rest of the processor, according to the opcode value passed in from the instruction register.

Brainstorming!

- If you were to create your own low-level language that did a couple 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 to come....☺