CSE 390B, Autumn 2022

Building Academic Success Through Bottom-Up Computing

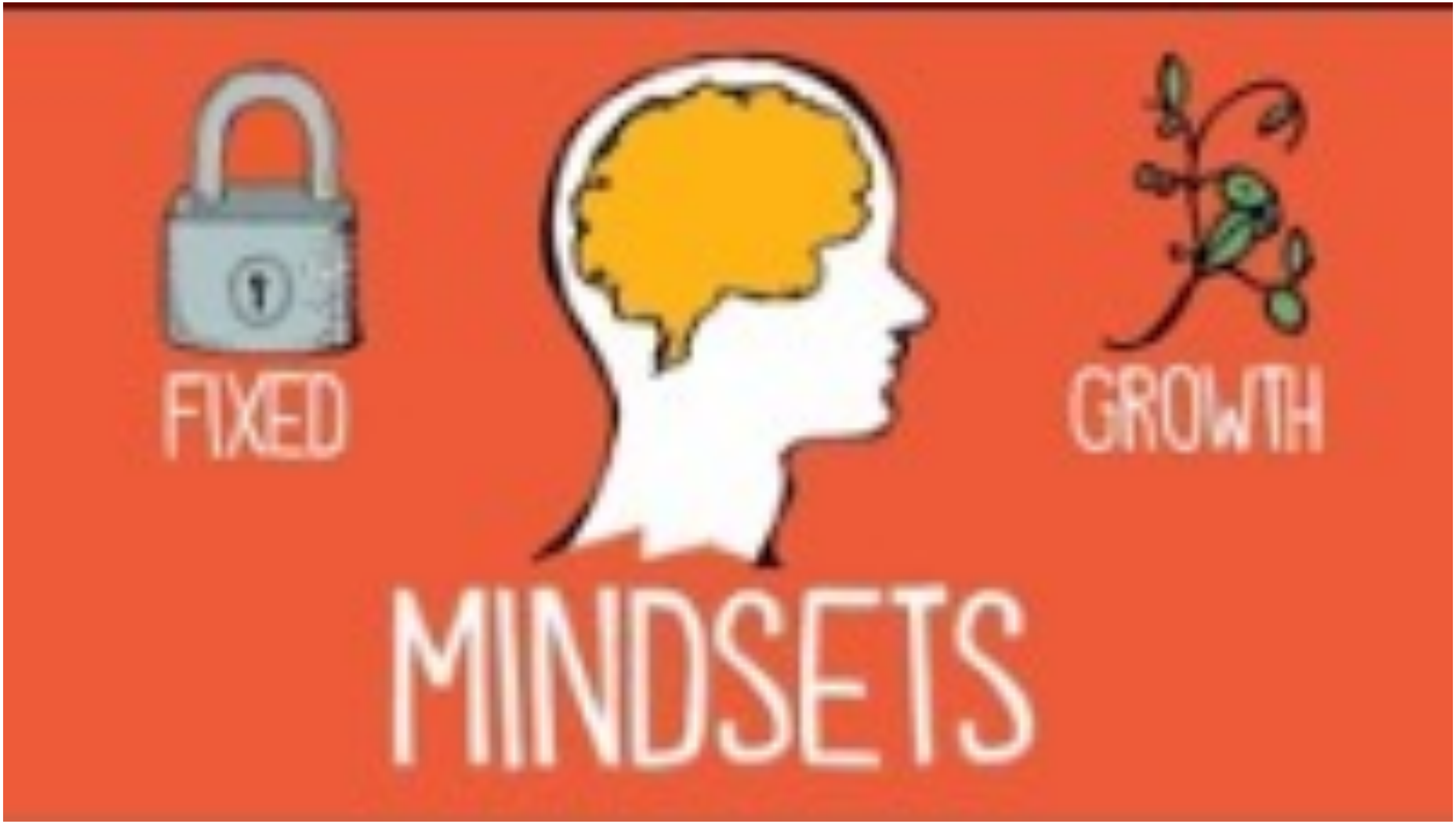
Growth vs. Fixed Mindset & The ALU

Growth vs. Fixed Mindset, Binary Number Representations, The Arithmetic Logic Unit (ALU), Project 3 Overview

Lecture Outline

- **Growth vs. Fixed Mindset**
  - **Setting SMART Goals**
- Binary Number Representations
  - Unsigned, Signed, and Two's Complement
- The Arithmetic Logic Unit (ALU)
  - Specification and ALU Function Examples
- Project 3 Overview
  - ALU Implementation Strategy
  - HDL Tips and Tricks

Growth vs. Fixed Mindset



Setting SMART Goals

- **S** – Be specific, simple and significant.
- **M** – Make sure your goals are measurable. How many times within a week, month, the quarter do you want to do x goal?
- **A** – Make sure your goals are achievable. Is your goal within your scope of control?
- **R** – Be realistic and reasonable.
- **T** – Be time-bound. When will you accomplish your goal?

SMART Goals Group Discussion

AUTUMN QUARTER GOALS	SPHERE OF CONTROL	SMART GOAL FRAMEWORK
What are skills, practices or habits that are not strengths YET?	Getting a 4.0 in a course vs. Attending course office hours	**S** — Specific **M** — Measurable **A** — Achievable **R** — Realistic **T** — Timebound Attending CSE 390B office hours at least 5x this quarter (or once every other week)

Lecture Outline

- Growth vs. Fixed Mindset
  - Setting SMART Goals
- **Binary Number Representations**
  - **Unsigned, Signed, and Two's Complement**
- The Arithmetic Logic Unit (ALU)
  - Specification and ALU Function Examples
- Project 3 Overview
  - ALU Implementation Strategy
  - HDL Tips and Tricks

Unsigned Binary Representation

- To interpret, we multiply the value of each bit by the power of two that specific bit represents

- This system is unable to represent negative numbers

$$\begin{array}{l}\text{Exponent: } 3\ 2\ 1\ 0\\ \qquad\qquad\quad\ \downarrow\downarrow\downarrow\downarrow\end{array}$$

- Example: 0b1101 in **unsigned binary**
  - $(1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0)$

  $= (1 \times 8) + (1 \times 4) + (0 \times 2) + (1 \times 1)$

  $= 8 + 4 + 0 + 1$

  $= 13$

Signed Binary Representation

- Also called the **sign and magnitude** number encoding
- Most significant bit (MSB) represents the **sign** of the number
  - The remaining bits represent the **weight** of the number

Exponent: 3 2 1 0

- Example: 0b1101 in **signed binary**
  - $-((1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0))$

  $= -((1 \times 4) + (0 \times 2) + (1 \times 1))$

  $= -(4 + 0 + 1)$

  $= -5$

Signed Binary Representation: Limitations

- Good first attempt at encoding negative numbers, but there are two main problems

- First, there exists two representations of zero (a positively and negatively signed zero)

- Second, adding numbers no longer works universally
  - Addition no longer works with negative numbers

```
Carry:   0 0 1 0 0
-----------------
           0 0 1 0     (2)
       +   1 0 1 0     (-2)
       -----------
           1 1 0 0     (-4) ← This should be 0
```

Two's Complement Binary Representation

- Standard for encoding numbers in computers

- Most significant bit (MSB) has a negative weight
  - Add the remaining bits as usual (with positive weights)

Exponent: 3 2 1 0

- Example: 0b1101 in Two's Complement
  - $-(1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0)$

  $= -(1 \times 8) + (1 \times 4) + (0 \times 2) + (1 \times 1)$

  $= -8 + 4 + 0 + 1$

  $= -3$

Benefits of Two's Complement

- Only one representation of zero
- Represents more unique numbers compared to sign and magnitude given a fixed width binary number
- Simple negation procedure: Take the bitwise complement and add one ($\mathbf{-x = \sim x + 1}$)
  - Example: To negate $x = 4$:
  - $\sim 0b0100 + 1 = 0b1011 + 0b1 = 0b1100 = -8 + 4 = -4$

Four-bit Values in Various Representations

Binary Value	Unsigned Binary	Signed Binary	Two's Complement
0b0000	0	0	0
0b0001	1	1	1
0b0010	2	2	2
0b0011	3	3	3
0b0100	4	4	4
0b0101	5	5	5
0b0110	6	6	6
0b0111	7	7	7
0b1000	8	-0	-8
0b1001	9	-1	-7
0b1010	10	-2	-6
0b1011	11	-3	-5
0b1100	12	-4	-4
0b1101	13	-5	-3
0b1110	14	-6	-2
0b1111	15	-7	-1

Two's Complement Addition

- The process for adding binary in Two's Complement is the same as that of unsigned binary

- Hardware performs the exact same calculations
  - It doesn't need to know the sign of the values, it performs the same calculation
  - The only difference is representation of sum

- Example: $0b1001 + 0b0010$
  - Unsigned interpretation:
  - Signed interpretation:
  - Two's Complement interpretation:

carry				
a	1	0	0	1
b	0	0	1	0
sum				

Two’s Complement Addition

- The process for adding binary in Two’s Complement is the same as that of unsigned binary

- Hardware performs the exact same calculations
  - It doesn’t need to know the sign of the values, it performs the same calculation
  - The only difference is representation of sum

- Example: $0b1001 + 0b0010 = 0b1011$
  - Unsigned interpretation: $9 + 2 = 11$
  - Signed interpretation: $-1 + 2 = -3$ (?)
  - Two’s Complement interpretation: $-7 + 2 = -5$

carry				
a	1	0	0	1
b	0	0	1	0
sum	1	0	1	1

Poll Everywhere

Vote at https://pollev.com/cse390b

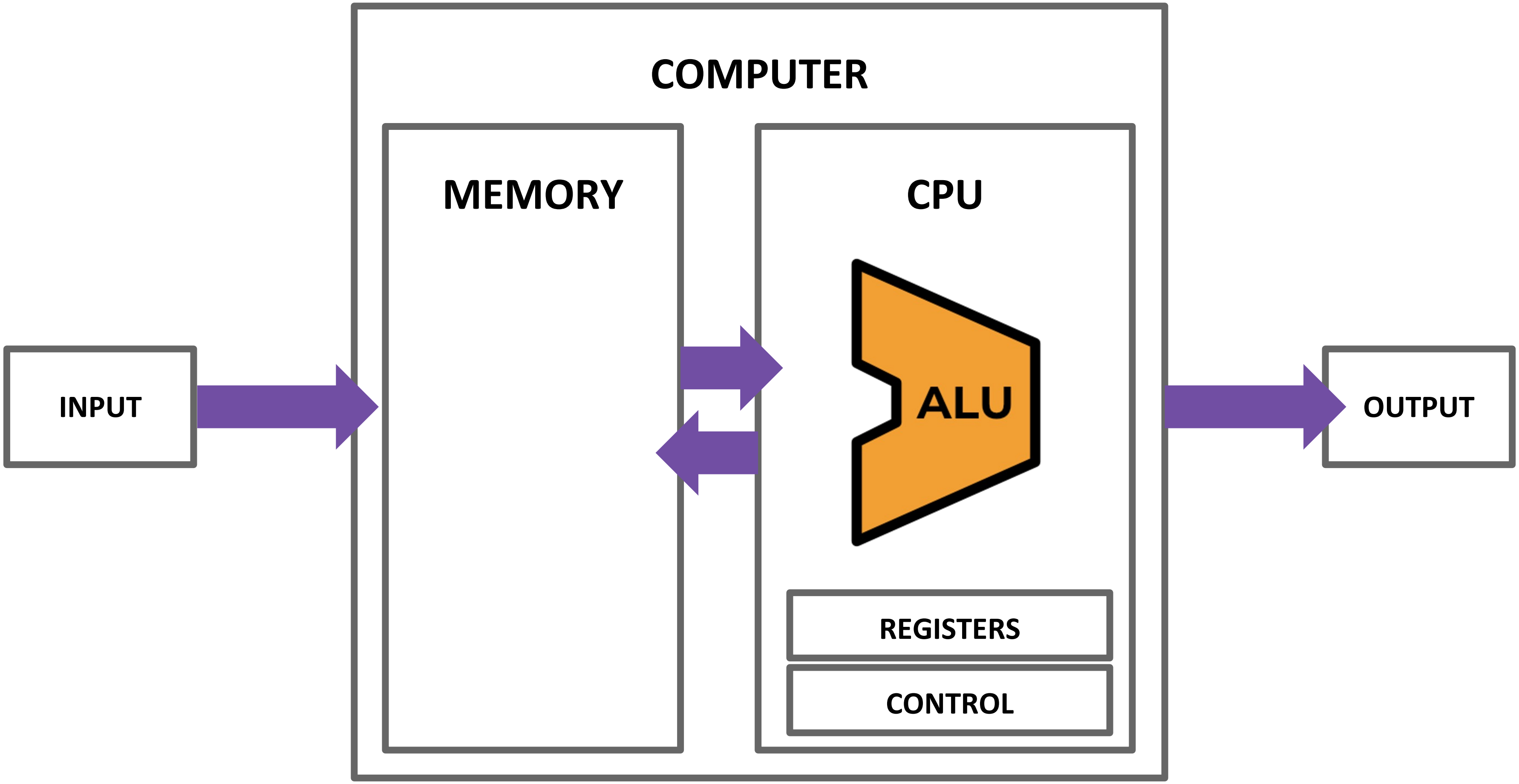
What are the Sign and Magnitude (signed) and Two's Complement binary representations of the number -7?

A. Signed: 0b1001, Two's Complement: 0b1111
B. Signed: 0b0111, Two's Complement: 0b1001
C. Signed: 0b1111, Two's Complement: 0b0111
D. Signed: 0b1111, Two's Complement: 0b1001
E. We're lost…

Lecture Outline

- Growth vs. Fixed Mindset
  - Setting SMART Goals
- Binary Number Representations
  - Unsigned, Signed, and Two's Complement
- **The Arithmetic Logic Unit (ALU)**
  - **Specification and ALU Function Examples**
- Project 3 Overview
  - ALU Implementation Strategy
  - HDL Tips and Tricks

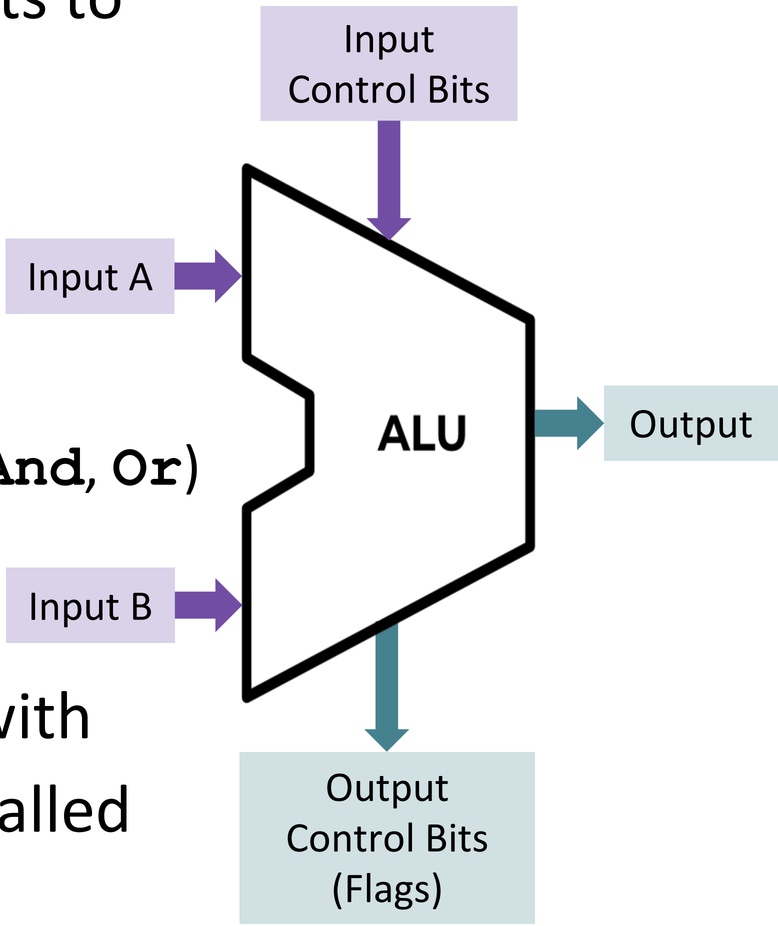
The Von Neumann Architecture



(This picture will get more detailed as we go!)

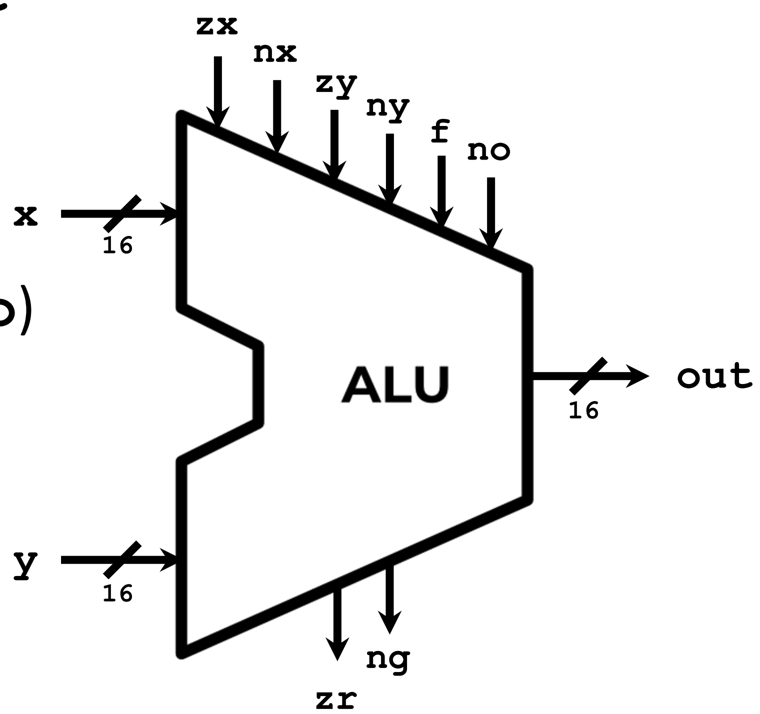
The Arithmetic Logic Unit

- Computes a function on two inputs to produce an output
- Input Control Bits specific which function should be computed
  - Supports a combination of logical (**`And`**, **`Or`**) and arithmetic operations (+, –)
- Indicate properties of the result with **Output Control Bits** (commonly called **Flags**)



Our ALU Implementation

- Inputs & Outputs
  - 16-bit inputs **`x`** and **`y`** and output **`out`**
  - Interpret in Two's Complement
- Input Control Bits
  - Six control bits (**`zx`**, **`nx`**, **`zy`**, **`ny`**, **`f`**, **`no`**) specify which function to compute
  - $2^6 = 64$ different possible functions to choose from (only 18 of interest)
- Output Control Bits (Flags)
  - 2 bits (**`zr`** and **`ng`**) describing the properties of the output



ALU Functions: Client's View

- We support 18 different functions of interest
  - 3 that simply give constant values (ignoring operands)
  - 10 that change a single input, possibly with a constant
  - 5 that perform an operation using both inputs

out
0
1
-1
x
y
!x
!y
-x
-y
x+1
y+1
x-1
y-1
x+y
x-y
y-x
x&y
x|y

ALU Functions: Client's View

- We support 18 different functions of interest
  - 3 that simply give constant values (ignoring operands)
  - 10 that change a single input, possibly with a constant
  - 5 that perform an operation using both inputs
- To select a function, set the control bits to the corresponding combination

zx	nx	zy	ny	f	no	out
1	0	1	0	1	0	0
1	1	1	1	1	1	1
1	1	1	0	1	0	-1
0	0	1	1	0	0	x
1	1	0	0	0	0	y
0	0	1	1	0	1	!x
1	1	0	0	0	1	!y
0	0	1	1	1	1	-x
1	1	0	0	1	1	-y
0	1	1	1	1	1	x+1
1	1	0	1	1	1	y+1
0	0	1	1	1	0	x-1
1	1	0	0	1	0	y-1
0	0	0	0	1	0	x+y
0	1	0	0	1	1	x-y
0	0	0	1	1	1	y-x
0	0	0	0	0	0	x&y
0	1	0	1	0	1	x|y

Five-minute Break!

- Feel free to stand up, stretch, use the restroom, drink some water, review your notes, or ask questions
- We'll be back at:

ALU Functions: Implementer's View

- These 18 functions are really a clever combination of 6 core operations:

1 PREPROCESS INPUTS

if **zx==1**, set **x** to 0

if **nx==1**, negate **x**

if **zy==1**, set **y** to 0

if **ny==1**, negate **y**

2 COMPUTE

if **f==1**, result is **x + y**
else, result is **x & y**

3 POSTPROCESS OUTPUT

if **no==1**, negate result

zx	nx	zy	ny	f	no	out
1	0	1	0	1	0	0
1	1	1	1	1	1	1
1	1	1	0	1	0	-1
0	0	1	1	0	0	x
1	1	0	0	0	0	y
0	0	1	1	0	1	!x
1	1	0	0	0	1	!y
0	0	1	1	1	1	-x
1	1	0	0	1	1	-y
0	1	1	1	1	1	x+1
1	1	0	1	1	1	y+1
0	0	1	1	1	0	x-1
1	1	0	0	1	0	y-1
0	0	0	0	1	0	x+y
0	1	0	0	1	1	x-y
0	0	0	1	1	1	y-x
0	0	0	0	0	0	x&y
0	1	0	1	0	1	x|y

ALU Functions: Implementer's View

zx	nx	zy	ny	f	no	out
...						...
0	0	1	1	1	0	x-1
...						...

- Example: Compute **`x - 1`**
  - Given inputs **`x=0b0101`** (5), **`y=0b0010`** (2)

ALU Functions: Implementer's View

zx	nx	zy	ny	f	no	out
...						...
0	0	1	1	1	0	x-1
...						...

- Example: Compute **x - 1**
  - Given inputs **x=0b0101** (5), **y=0b0010** (2)

1
PREPROCESS INPUTS

if **zx==1**, set **x** to 0	if **zy==1**, set **y** to 0	**x=0b0101** (5) *Unchanged*	**y=0b0000** (0) *Zeroed*
if **nx==1**, negate **x**	if **ny==1**, negate **y**	**x=0b0101** (5) *Unchanged*	**y=0b1111** (-1) *Negated*

ALU Functions: Implementer's View

zx	nx	zy	ny	f	no	out
...						...
0	0	1	1	1	0	x-1
...						...

- Example: Compute **`x - 1`**
  - Given inputs **`x=0b0101`** (5), **`y=0b0010`** (2)

1
PREPROCESS INPUTS

if **`zx==1`**, set **`x`** to 0 → **`x=0b0101`** (5) *Unchanged*

if **`zy==1`**, set **`y`** to 0 → **`y=0b0000`** (0) *Zeroed*

if **`nx==1`**, negate **`x`** → **`x=0b0101`** (5) *Unchanged*

if **`ny==1`**, negate **`y`** → **`y=0b1111`** (-1) *Negated*

2
COMPUTE

if **`f==1`**, result is **`x + y`**
else, result is **`x & y`**

`out=0b0100` (4)
x (5) + y (-1)

ALU Functions: Implementer's View

- Example: Compute `x - 1`
  - Given inputs `x=0b0101` (5), `y=0b0010` (2)

zx	nx	zy	ny	f	no	out
...						...
0	0	1	1	1	0	x-1
...						...

1 PREPROCESS INPUTS

if `zx==1`, set `x` to 0 → `x=0b0101` (5) *Unchanged*

if `zy==1`, set `y` to 0 → `y=0b0000` (0) *Zeroed*

if `nx==1`, negate `x` → `x=0b0101` (5) *Unchanged*

if `ny==1`, negate `y` → `y=0b1111` (-1) *Negated*

2 COMPUTE

if `f==1`, result is `x + y`
else, result is `x & y`

`out=0b0100` (4)
x (5) + y (-1)

3 POSTPROCESS OUTPUT

if `no==1`, negate result

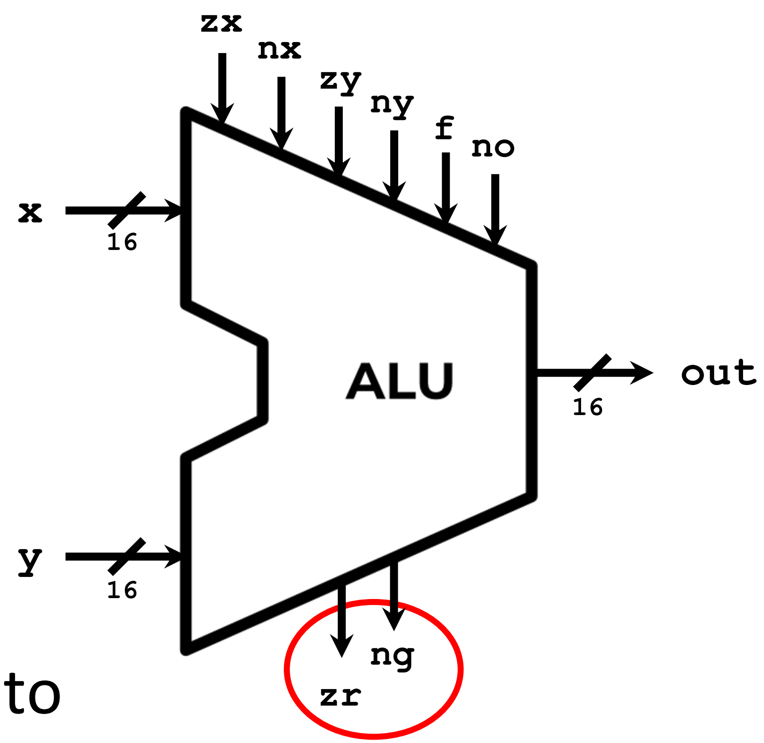
`out=0b0100` (4)
Unchanged

ALU Output Control Bits

- `zr` is 1 if `out == 0`
- `ng` is 1 if `out < 0`

- We'll use these in a later project
  - The basis of **comparison**
  - Example: To evaluate if `x == 4`, compute `x - 4` and check `zr` flag

- These can be deceptively difficult to implement
  - Start early on Project 3



Poll Everywhere

Vote at https://pollev.com/cse390b

Given inputs `x=0b1010` and `y=0b0110` and the following input control bits, what is the resulting output and output control bits?

zx	nx	zy	ny	f	no	out
...						...
0	1	1	1	1	1	x+1
...						...

```
IN
  x[16], y[16], // 16-bit inputs
  zx, // zero the x input?
  nx, // negate the x input?
  zy, // zero the y input?
  ny, // negate the y input?
  f,  // compute out = x + y
         if 1 or x & y (if 0)
  no; // negate the out output?

OUT
  out[16], // 16-bit output
  zr, // 1 if (out == 0),
         0 otherwise
  ng; // 1 if (out < 0),
         0 otherwise
```

A. out=0b1110, zr=0, ng=0
B. out=0b1011, zr=0, ng=1
C. out=0b1101, zr=1, ng=0
D. out=0b1001, zr=1, ng=1
E. We're lost…

Lecture Outline

- Growth vs. Fixed Mindset
  - Setting SMART Goals
- Binary Number Representations
  - Unsigned, Signed, and Two's Complement
- The Arithmetic Logic Unit (ALU)
  - Specification and ALU Function Examples
- **Project 3 Overview**
  - **ALU Implementation Strategy**
  - **HDL Tips and Tricks**

Project 3 Overview

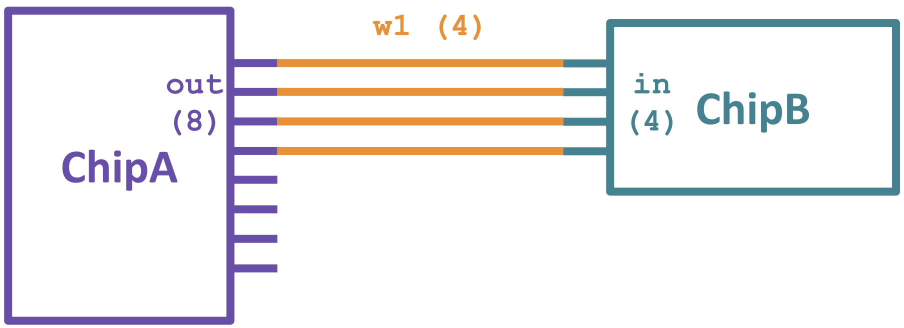
- Part I: 24-Hour Time Audit
- Part II: Boolean Arithmetic
  - Goal: Implement the ALU, which performs the core computations we need (+ and &)
  - First, implement `HalfAdder.hdl`, `FullAdder.hdl`, and `Add16.hdl`
  - Then, implement the ALU in the order suggested by the specification
  - Chapter 2 of the textbook has more details on the adders and ALU
- Part III: Project 3 Reflection

ALU Implementation Strategy

- First, handle zeroing out and negating inputs **`x`** and **`y`** and negating the output
  - Ignore the **`f`** bit (only compute **`And`**) and ignore flag outputs
  - Test your implementation using `ALU-nostat-noadd.tst`
- Next, implement the **`And`** and **`Add`** operations using **`f`**
  - How do we make decisions in hardware?
  - Test your implementation using `ALU-nostat.tst`
- Lastly, implement the logic for the status flags (**`zr`** and **`ng`**)
  - Test your full ALU using `ALU.tst`

HDL Tips: Slicing

- Sometimes want to connect only part of a multi-bit bus
- HDL lets us with **slicing notation**
- Example: **ChipA** has eight output pins, and we want to connect the first four to **ChipB**’s four inputs:

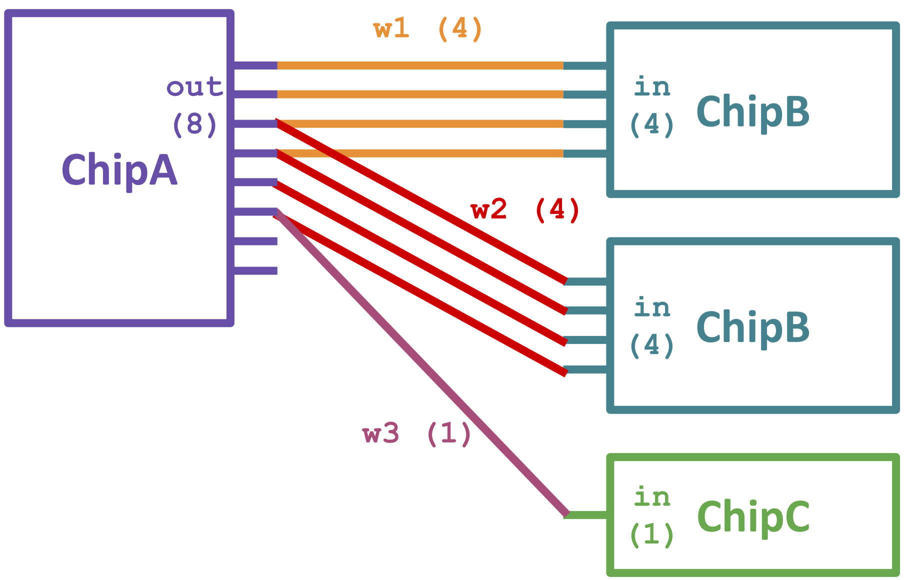



```
ChipA (out[0..3]=w1);
ChipB (in=w1);
```

- Note: We can only slice chip connections, not internal wires (e.g., **`w1[0..3]`** is not allowed)
  - If we need to use half an 8-bit wire, make two 4-bit wires and slice the output they’re connected to

HDL Tips: Connections

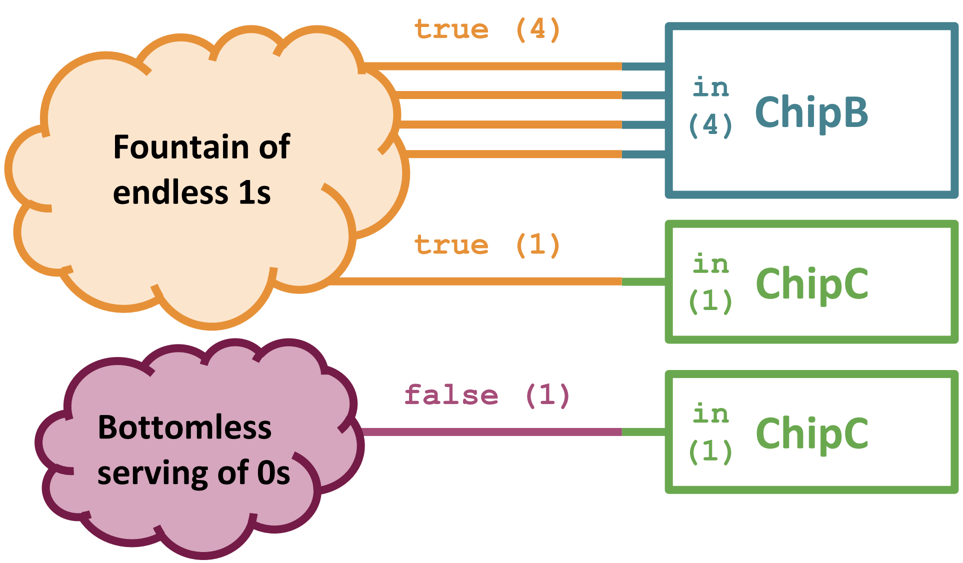
- Can connect a chip output multiple times, or not at all!
  - Hint: In `Add16.hdl`, do we need to use the last carry bit?




```
ChipA (out[0..3]=w1,
       out[2..5]=w2,
       out[5]=w3);
ChipB (in=w1);
ChipB (in=w2);
ChipC (in=w3);
```

HDL Tips: Constants

- A bus of `true` or `false` contain all 1s or all 0s, respectively, and implicitly act as whatever width is needed
- Example: `ChipB` has four inputs and `ChipC` has one input
  - `ChipB (in=true)` assigns four input bits a value of 1 (`true`)
  - `ChipC (in=true)` assigns one input bit a value of 1 (`true`)
  - `ChipC (in=false)` assigns one input bit a value of 0 (`false`)




```
ChipB (in=true);
ChipC (in=true);
ChipC (in=false);
```

Post-Lecture 5 Reminders

- **Project 2 due tonight (10/13) at 11:59pm**

- Project 3 (24-Hour Time Audit & Boolean Arithmetic) released today, due next Thursday (10/20)

- Course Staff Support
  - Eric has office hours in CSE2 153 today after lecture
  - Post your questions on the Ed discussion board