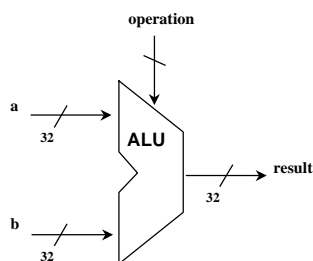


## Arithmetic

---

- Where we've been:
  - Performance (seconds, cycles, instructions)
  - Abstractions:
    - Instruction Set Architecture
    - Assembly Language and Machine Language
- What's up ahead:
  - Implementing the Architecture



## Numbers

---

- Bits are just bits (no inherent meaning)
  - conventions define relationship between bits and numbers
- Binary numbers (base 2)
  - 0000 0001 0010 0011 0100 0101 0110 0111 1000 1001...
  - decimal:  $0 \dots 2^n - 1$
- Of course it gets more complicated:
  - numbers are finite (overflow)
  - fractions and real numbers
  - negative numbers
  - e.g., no MIPS `subi` instruction; `addi` can add a negative number)
- How do we represent negative numbers?
  - i.e., which bit patterns will represent which numbers?

## Possible Representations

| Sign Magnitude: | One's Complement | Two's Complement |
|-----------------|------------------|------------------|
| 000 = +0        | 000 = +0         | 000 = +0         |
| 001 = +1        | 001 = +1         | 001 = +1         |
| 010 = +2        | 010 = +2         | 010 = +2         |
| 011 = +3        | 011 = +3         | 011 = +3         |
| 100 = -0        | 100 = -3         | 100 = -4         |
| 101 = -1        | 101 = -2         | 101 = -3         |
| 110 = -2        | 110 = -1         | 110 = -2         |
| 111 = -3        | 111 = -0         | 111 = -1         |

- Issues: balance, number of zeros, ease of operations
- Which one is best? Why?

## MIPS

- 32 bit signed numbers:

|   |   |                 |
|---|---|-----------------|
| 0000 0000 0000 0000 0000 0000 0000 0000 | $0000_{\text{two}} = 0_{\text{ten}}$              |                 |
| 0000 0000 0000 0000 0000 0000 0000 0001 | $0001_{\text{two}} = +1_{\text{ten}}$             |                 |
| 0000 0000 0000 0000 0000 0000 0000 0010 | $0010_{\text{two}} = +2_{\text{ten}}$             |                 |
| ...                                     |   |                 |
| 0111 1111 1111 1111 1111 1111 1111 1110 | $1110_{\text{two}} = +2,147,483,646_{\text{ten}}$ | / <i>maxint</i> |
| 0111 1111 1111 1111 1111 1111 1111 1111 | $1111_{\text{two}} = +2,147,483,647_{\text{ten}}$ |                 |
| 1000 0000 0000 0000 0000 0000 0000 0000 | $0000_{\text{two}} = -2,147,483,648_{\text{ten}}$ | \ <i>minint</i> |
| 1000 0000 0000 0000 0000 0000 0000 0001 | $0001_{\text{two}} = -2,147,483,647_{\text{ten}}$ |                 |
| 1000 0000 0000 0000 0000 0000 0000 0010 | $0010_{\text{two}} = -2,147,483,646_{\text{ten}}$ |                 |
| ...                                     |   |                 |
| 1111 1111 1111 1111 1111 1111 1111 1101 | $1101_{\text{two}} = -3_{\text{ten}}$             |                 |
| 1111 1111 1111 1111 1111 1111 1111 1110 | $1110_{\text{two}} = -2_{\text{ten}}$             |                 |
| 1111 1111 1111 1111 1111 1111 1111 1111 | $1111_{\text{two}} = -1_{\text{ten}}$             |                 |

## Two's Complement Operations

---

- Negating a two's complement number: invert all bits and add 1
  - remember: “negate” and “invert” are quite different!
- Converting n bit numbers into numbers with more than n bits:
  - MIPS 16 bit immediate gets converted to 32 bits for arithmetic
  - copy the most significant bit (the sign bit) into the other bits
    - 0010 -> 0000 0010
    - 1010 -> 1111 1010
  - “sign extension” (lbu vs. lb)

## Addition & Subtraction

---

- Just like in grade school (carry/borrow 1s)
  - $$\begin{array}{r} 0111 \\ + 0110 \\ \hline \end{array}$$
  - $$\begin{array}{r} 0111 \\ - 0110 \\ \hline \end{array}$$
  - $$\begin{array}{r} 0110 \\ - 0101 \\ \hline \end{array}$$
- Two's complement operations easy
  - subtraction using addition of negative numbers
    - $$\begin{array}{r} 0111 \\ + 1010 \\ \hline \end{array}$$
- Overflow (result too large for finite computer word):
  - e.g., adding two n-bit numbers does not yield an n-bit number
    - $$\begin{array}{r} 0111 \\ + 0001 \\ \hline 1000 \end{array}$$
 *note that overflow term is somewhat misleading, it does not mean a carry “overflowed”*

## Detecting Overflow

---

- No overflow when adding a positive and a negative number
- No overflow when signs are the same for subtraction
- Overflow occurs when the value affects the sign:
  - overflow when adding two positives yields a negative
  - or, adding two negatives gives a positive
  - or, subtract a negative from a positive and get a negative
  - or, subtract a positive from a negative and get a positive
- Consider the operations  $A + B$ , and  $A - B$ 
  - Can overflow occur if  $B$  is 0 ?
  - Can overflow occur if  $A$  is 0 ?

## Effects of Overflow

---

- An exception (interrupt) occurs
  - Control jumps to predefined address for exception
  - Interrupted address is saved for possible resumption
- Details based on software system / language
  - example: flight control vs. homework assignment
- Don't always want to detect overflow
  - new MIPS instructions: `addu`, `addiu`, `subu`

*note: addiu still sign-extends!*

*note: sltu, sltiu for unsigned comparisons*

## Review: Boolean Algebra & Gates

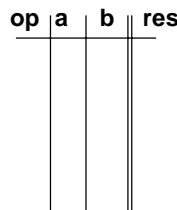
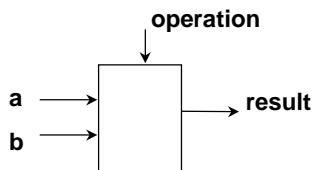
- Problem: Consider a logic function with three inputs: A, B, and C.

Output D is true if at least one input is true  
 Output E is true if exactly two inputs are true  
 Output F is true only if all three inputs are true

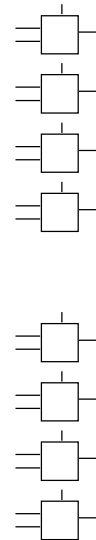
- Show the truth table for these three functions.
- Show the Boolean equations for these three functions.
- Show an implementation consisting of inverters, AND, and OR gates.

## An ALU (arithmetic logic unit)

- Let's build an ALU to support the `andi` and `ori` instructions
  - we'll just build a 1 bit ALU, and use 32 of them



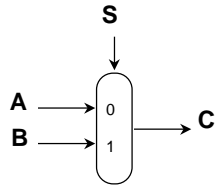
- Possible Implementation (sum-of-products):



## Review: The Multiplexor

---

- Selects one of the inputs to be the output, based on a control input



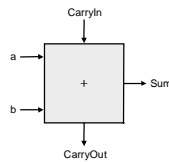
*note: we call this a 2-input mux  
even though it has 3 inputs!*

- Lets build our ALU using a MUX:

## Different Implementations

---

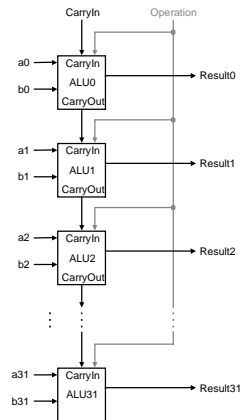
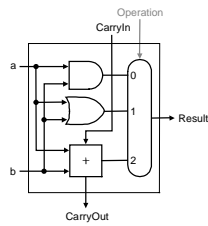
- Not easy to decide the “best” way to build something
  - Don't want too many inputs to a single gate
  - Dont want to have to go through too many gates
  - for our purposes, ease of comprehension is important
- Let's look at a 1-bit ALU for addition:



$$c_{out} = a b + a c_{in} + b c_{in}$$
$$sum = a \text{ xor } b \text{ xor } c_{in}$$

- How could we build a 1-bit ALU for add, and, and or?
- How could we build a 32-bit ALU?

## Building a 32 bit ALU



## What about subtraction ( $a - b$ ) ?

- Two's complement approach: just negate  $b$  and add.
- How do we negate?
- A very clever solution:

