Arithmetic Logic Unit (ALU)

Introduction to Computer
Yung-Yu Chuang

with slides by Sedgewick & Wayne (introcs.cs.princeton.edu), Nisan & Schocken (www.nand2tetris.org) and Harris & Harris (DDCA)
Let's Make an Adder Circuit

**Goal.** \( x + y = z \) for 4-bit integers.
- We build 4-bit adder: 9 inputs, 4 outputs.
- Same idea scales to 128-bit adder.
- Key computer component.

\[
\begin{array}{cccc}
1 & 1 & 1 & 0 \\
2 & 4 & 8 & 7 \\
+ & 3 & 5 & 7 & 9 \\
\hline
6 & 0 & 6 & 6 \\
\end{array}
\]
Binary addition

Assuming a 4-bit system:

\[
\begin{array}{cccc}
0 & 0 & 0 & 1 \\
+ & 1 & 0 & 0 & 1 \\
\hline
& 1 & 0 & 1 & 1 \\
\end{array}
\]

\[
\begin{array}{cccc}
1 & 1 & 1 & 1 \\
+ & 0 & 1 & 1 & 1 \\
\hline
& 1 & 0 & 0 & 1 & 0 \\
\end{array}
\]

no overflow

Algorithm: exactly the same as in decimal addition

Overflow (MSB carry) has to be dealt with.
Representing negative numbers (4-bit system)

- The codes of all positive numbers begin with a “0”
- The codes of all negative numbers begin with a “1”
- To convert a number: leave all trailing 0’s and first 1 intact, and flip all the remaining bits

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>1111</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>1110</td>
<td>-2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>1101</td>
<td>-3</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>1011</td>
<td>-5</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>1010</td>
<td>-6</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>1001</td>
<td>-7</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>1000</td>
<td>-8</td>
<td></td>
</tr>
</tbody>
</table>

Example: $2 - 5 = 2 + (-5) = \begin{array}{c}
0010 \\
+ 1011 \\
\hline
1101
\end{array} = -3$
Let's Make an Adder Circuit

**Step 1.** Represent input and output in binary.
Let's Make an Adder Circuit

Goal. \( x + y = z \) for 4-bit integers.

Step 2. [first attempt]
  1. Build truth table.

<table>
<thead>
<tr>
<th>( c_0 )</th>
<th>( x_3 )</th>
<th>( x_2 )</th>
<th>( x_1 )</th>
<th>( x_0 )</th>
<th>( y_3 )</th>
<th>( y_2 )</th>
<th>( y_1 )</th>
<th>( y_0 )</th>
<th>( z_3 )</th>
<th>( z_2 )</th>
<th>( z_1 )</th>
<th>( z_0 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>...</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\( 2^{8+1} = 512 \) rows!

Q. Why is this a bad idea?
A. 128-bit adder: \( 2^{256+1} \) rows $\gg$ # electrons in universe!
1-bit half adder

We add numbers one bit at a time.

\[
\begin{array}{cc}
  x & y \\
  \hline
  s & c \\
\end{array}
\]

\[
\begin{array}{cc}
  x & y \\
  \hline
  s & c \\
\end{array}
\]
1-bit full adder

\[ x \quad y \quad \text{C}_{\text{in}} \quad \text{C}_{\text{out}} \quad s \]

\[ \begin{array}{cccc}
0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 \\
0 & 1 & 0 & 1 \\
0 & 1 & 1 & 0 \\
1 & 0 & 0 & 1 \\
1 & 0 & 1 & 0 \\
1 & 1 & 0 & 0 \\
1 & 1 & 1 & 1 \\
\end{array} \]
8-bit adder
Let’s Make an Adder Circuit

Goal. \( x + y = z \) for 4-bit integers.

Step 2. [do one bit at a time]
  - Build truth table for carry bit.
  - Build truth table for summand bit.

### Carry Bit

<table>
<thead>
<tr>
<th>( x_i )</th>
<th>( y_i )</th>
<th>( c_i )</th>
<th>( c_{i+1} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

### Summand Bit

<table>
<thead>
<tr>
<th>( x_i )</th>
<th>( y_i )</th>
<th>( c_i )</th>
<th>( z_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Let's Make an Adder Circuit

**Goal.** \( x + y = z \) for 4-bit integers.

**Step 3.**
- Derive (simplified) Boolean expression.
Let’s Make an Adder Circuit

**Goal.** $x + y = z$ for 4-bit integers.

**Step 4.**
- Transform Boolean expression into circuit.
- Chain together 1-bit adders.
Adder: Interface

\[ x_3 \rightarrow y_3 \rightarrow x_2 \rightarrow y_2 \rightarrow x_1 \rightarrow y_1 \rightarrow x_0 \rightarrow y_0 \]

\[ \text{carry in} \]

\[ \text{carry out} \]

ADD

\[ z_3, z_2, z_1, z_0 \]
Adder: Component Level View

carry in

x₃ y₃ x₂ y₂ x₁ y₁ x₀ y₀

carry out

z₃ z₂ z₁ z₀
Subtractor

Subtractor circuit: \( z = x - y \).

- One approach: design like adder circuit
Subtractor

Subtractor circuit: \( z = x - y \).

- One approach: design like adder circuit
- Better idea: reuse adder circuit
  - 2’s complement: to negate an integer, flip bits, then add 1
Subtractor

Subtractor circuit: \( z = x - y \).

- One approach: design like adder circuit
- Better idea: reuse adder circuit
  - 2's complement: to negate an integer, flip bits, then add 1
Only one of them will be on at a time.

4-bit Shifter
### Shifter

<table>
<thead>
<tr>
<th>$s_0$</th>
<th>$z_0$</th>
<th>$z_1$</th>
<th>$z_2$</th>
<th>$z_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$s_2$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$s_3$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Shifter

<table>
<thead>
<tr>
<th>(s)</th>
<th>(z_0)</th>
<th>(z_1)</th>
<th>(z_2)</th>
<th>(z_3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(s_0)</td>
<td>(x_0)</td>
<td>(x_1)</td>
<td>(x_2)</td>
<td>(x_3)</td>
</tr>
<tr>
<td>(s_1)</td>
<td>0</td>
<td>(x_0)</td>
<td>(x_1)</td>
<td>(x_2)</td>
</tr>
<tr>
<td>(s_2)</td>
<td>0</td>
<td>0</td>
<td>(x_0)</td>
<td>(x_1)</td>
</tr>
<tr>
<td>(s_3)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>(x_0)</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
  z_0 &= s_0 \cdot x_0 + s_1 \cdot 0 + s_2 \cdot 0 + s_3 \cdot 0 \\
  z_1 &= s_0 \cdot x_1 + s_1 \cdot x_0 + s_2 \cdot 0 + s_3 \cdot 0 \\
  z_2 &= s_0 \cdot x_2 + s_1 \cdot x_1 + s_2 \cdot x_0 + s_3 \cdot 0 \\
  z_3 &= s_0 \cdot x_3 + s_1 \cdot x_2 + s_2 \cdot x_1 + s_3 \cdot x_0
\end{align*}
\]
Shifter

Right-shifter

\[
\begin{align*}
z_0 &= s_0 \cdot x_0 + s_1 \cdot 0 + s_2 \cdot 0 + s_3 \cdot 0 \\
z_1 &= s_0 \cdot x_1 + s_1 \cdot x_0 + s_2 \cdot 0 + s_3 \cdot 0 \\
z_2 &= s_0 \cdot x_2 + s_1 \cdot x_1 + s_2 \cdot x_0 + s_3 \cdot 0 \\
z_3 &= s_0 \cdot x_3 + s_1 \cdot x_2 + s_2 \cdot x_1 + s_3 \cdot x_0
\end{align*}
\]
N-bit Decoder

N-bit decoder
- N address inputs, $2^N$ data outputs
- Addresses output bit is 1; all others are 0
N-bit Decoder

N-bit decoder
- N address inputs, $2^N$ data outputs
- Addresses output bit is 1; all others are 0

3-Bit Decoder Interface

Decoder
2-Bit Decoder Controlling 4-Bit Shifter

Ex. Put in a binary amount $r_0r_1$ to shift.

Right-shifter with decoder
Arithmetic Logic Unit

**Arithmetic logic unit (ALU).** Computes all operations in parallel.
- Add and subtract.
- Xor.
- And.
- Shift left or right.

**Q.** How to select desired answer?
1 Hot OR

1 hot OR.

- All devices compute their answer; we pick one.
- Exactly one select line is on.
- Implies exactly one output line is relevant.

\[ x \cdot 1 = x \]
\[ x \cdot 0 = 0 \]
\[ x + 0 = x \]
1 Hot OR

$x \cdot 1 = x$

$x \cdot 0 = 0$

$x + 0 = x$
16-bit bus
- Bundle of 16 wires
- Memory transfer
- Register transfer

8-bit bus
- Bundle of 8 wires
- TOY memory address

4-bit bus
- Bundle of 4 wires
- TOY register address
Bitwise AND, XOR, NOT

Bitwise logical operations
- Inputs $x$ and $y$: $n$ bits each
- Output $z$: $n$ bits
- Apply logical operation to each corresponding pair of bits

 Bitwise And Interface

 Bitwise And Implementation
TOY ALU

- Big combinational logic
- 16-bit bus
- Add, subtract, and, xor, shift left, shift right.

<table>
<thead>
<tr>
<th>op</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>+,-</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>&amp;</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>^</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>&lt;&lt;,&gt;&gt;</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>input 2</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Device Interface Using Buses

Device. Processes a word at a time.
Input bus. Wires on top.
Output bus. Wires on bottom.
Control. Individual wires on side.

16-bit words for TOY memory
Arithmetic logic unit.
- Add and subtract.
- Xor.
- And.
- Shift left or right.

How to convert opcode to 1-hot OR signal?
Hack ALU

\[
\text{out}(x, y, \text{control bits}) = \begin{align*}
& x+y, x-y, y-x, \\
& 0, 1, -1, \\
& x, y, -x, -y, \\
& x!, y!, \\
& x+1, y+1, x-1, y-1, \\
& x\& y, x|y
\end{align*}
\]
# Hack ALU

These bits instruct how to preset the x input

These bits instruct how to preset the y input

This bit selects between + / \& And

This bit inst. how to postset out

Resulting ALU output

<table>
<thead>
<tr>
<th>zx</th>
<th>nx</th>
<th>zy</th>
<th>ny</th>
<th>f</th>
<th>no</th>
<th>out=</th>
</tr>
</thead>
<tbody>
<tr>
<td>if zx then x=0</td>
<td>if nx then x=!x</td>
<td>if zy then y=0</td>
<td>if ny then y=!y</td>
<td>if f then out=x+y else out=x&amp;y</td>
<td>if no then out=!out</td>
<td>f(x, y)=</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>y</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>!x</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>!y</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-x</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>-y</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>x+1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>y+1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>x-1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>y-1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>x+y</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>x-y</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>y-x</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>x&amp;y</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>x</td>
</tr>
</tbody>
</table>
The ALU in the CPU context (a sneak preview of the Hack platform)

Perspective

- Combinational logic
- Our adder design is very basic: no parallelism
- It pays to optimize adders
- Our ALU is also very basic: no multiplication, no division
- Where is the seat of more advanced math operations? a typical hardware/software tradeoff.