Arithmetic and Logic Unit

CS 365 Lecture 5
Prof. Yih Huang

Inside a Processor

<table>
<thead>
<tr>
<th>Branch</th>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Control Logic</td>
<td>Integer Arithmetic Circuits</td>
</tr>
<tr>
<td></td>
<td>Floating Point Arithmetic Circuits</td>
</tr>
<tr>
<td>Data Cache</td>
<td>Instruction Cache</td>
</tr>
<tr>
<td></td>
<td>(Internal) Bus</td>
</tr>
</tbody>
</table>
Arithmetic and Logic Unit (ALU)

- The part of a processor circuit that actually gets the computations done.

```
operation

a
32

ALU

b
32

result
32
```

Numbers

- Bits are just bits (no inherent meaning)
- Binary numbers (base 2) ⇒ decimal: 0...2^n-1
- ASCII codes
- Of course it gets more complicated:
  - numbers are finite (overflow)
  - fractions and real numbers
  - negative numbers
- How do we represent negative numbers?
### Possible Representations

<table>
<thead>
<tr>
<th>Sign Magnitude:</th>
<th>One's Complement</th>
<th>Two's Complement</th>
</tr>
</thead>
<tbody>
<tr>
<td>000 = +0</td>
<td>000 = +0</td>
<td>000 = +0</td>
</tr>
<tr>
<td>001 = +1</td>
<td>001 = +1</td>
<td>001 = +1</td>
</tr>
<tr>
<td>010 = +2</td>
<td>010 = +2</td>
<td>010 = +2</td>
</tr>
<tr>
<td>011 = +3</td>
<td>011 = +3</td>
<td>011 = +3</td>
</tr>
<tr>
<td>100 = -0</td>
<td>100 = -3</td>
<td>100 = -4</td>
</tr>
<tr>
<td>101 = -1</td>
<td>101 = -2</td>
<td>101 = -3</td>
</tr>
<tr>
<td>110 = -2</td>
<td>110 = -1</td>
<td>110 = -2</td>
</tr>
<tr>
<td>111 = -3</td>
<td>111 = -0</td>
<td>111 = -1</td>
</tr>
</tbody>
</table>

Most of the modern architectures use two’s complement.

### Two’s Complement Numbers

<table>
<thead>
<tr>
<th>$X_3$</th>
<th>$X_2$</th>
<th>$X_1$</th>
<th>$X_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$-2^3$</td>
<td>$2^2$</td>
<td>$2^1$</td>
<td>$2^0$</td>
</tr>
</tbody>
</table>

- 0010 =
- 1010 =
- -10 in 8-bit two’s complement =
### 32-bit Signed Numbers

<table>
<thead>
<tr>
<th>Two’s Complement</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0000 0000 0000 0000 0000 0000 0000 = 0</td>
<td></td>
</tr>
<tr>
<td>0000 0000 0000 0000 0000 0000 0000 0001 = +1</td>
<td></td>
</tr>
<tr>
<td>0000 0000 0000 0000 0000 0000 0000 0010 = +2</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0111 1111 1111 1111 1111 1111 1110 = +2,147,483,646</td>
<td></td>
</tr>
<tr>
<td>0111 1111 1111 1111 1111 1111 1111 = +2,147,483,647</td>
<td></td>
</tr>
<tr>
<td>1000 0000 0000 0000 0000 0000 0000 0000 = -2,147,483,648</td>
<td></td>
</tr>
<tr>
<td>1000 0000 0000 0000 0000 0000 0000 0001 = -2,147,483,647</td>
<td></td>
</tr>
<tr>
<td>1000 0000 0000 0000 0000 0000 0000 0010 = -2,147,483,646</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>1111 1111 1111 1111 1111 1111 1111 1101 = -3</td>
<td></td>
</tr>
<tr>
<td>1111 1111 1111 1111 1111 1111 1111 1110 = -2</td>
<td></td>
</tr>
<tr>
<td>1111 1111 1111 1111 1111 1111 1111 1111 = -1</td>
<td></td>
</tr>
</tbody>
</table>

---

### Two's Complement Operations

- **Negating a two's complement number:**
  invert all bits and add 1
  - remember: “negate” and “invert” are different!

- **Exercises (in 6 bits)**
  - Negate 12
  - Negate -5
Sign Extensions

- 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

4 bit number  8 bit equivalent

Additions & Subtractions

- Just like regular binary numbers

\[
\begin{array}{ccc}
0010 & + & 0001 & + & 1111 \\
0110 & + & 1110 & + & 1111 \\
1111 & + & 1111 & + & 1111 \\
1111 & + & 1111 & + & 1111 \\
\end{array}
\]
Overflows

- Result too large to store in finite-size computer words
  - e.g., adding two n-bit numbers does not always yield an n-bit number
- Depends on the kind of numbers you have in mind: Signed or unsigned

\[
\begin{array}{ccc}
0010 & + & 0110 \\
0100 & - & 0001 \\
\end{array}
\]

Detecting Overflow

- No overflow when adding a positive and a negative number
- No overflow when signs are the same for subtraction
- Overflows when the value affects the sign:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>A+B</td>
<td>&gt;0</td>
<td>&gt;0</td>
<td>&lt;0</td>
</tr>
<tr>
<td>A+B</td>
<td>&lt;0</td>
<td>&lt;0</td>
<td>&gt;0</td>
</tr>
<tr>
<td>A−B</td>
<td>&gt;0</td>
<td>&lt;0</td>
<td>&lt;0</td>
</tr>
<tr>
<td>A−B</td>
<td>&lt;0</td>
<td>&gt;0</td>
<td>&gt;0</td>
</tr>
</tbody>
</table>
**Effects of Overflow**

- Architecture and case dependent
- Solution 1: just remember it and leave the handing to software.
  - The condition/flag register of IA32
- Solution 2: exception/interrupt
  - Control jumps to predefined address for exception
  - Interrupted address is saved for possible resumption
  - Used by MIPS

**Discussion**

- IA32 provides an addc (add with carry) instruction. What is its use?
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 the three functions.
- Show an implementation consisting of inverters, AND, and OR gates

---

Design An Overflow Detector

- Inputs: SA (sign of A), SB (Sign of B), OP (operation, 0 for add, 1 for sub).
- Output: OF=0 no overflow, 1 overflow
- Truth Table:
- Boolean equation for OF.
- A circuit design of OF according to the equation above.

<table>
<thead>
<tr>
<th>OP</th>
<th>SA</th>
<th>SB</th>
<th>OF</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>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>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Review: The Multiplexer

- Selects one of the inputs to be the output, based on a control input
- Note: we call this 2-input multiplexer even though it actually has three inputs

More Inputs

- The general case: N-input multiplexer needs $\lceil \log_2 N \rceil$ select lines.
- You should be able to design its logic circuit.
Second Exercise

Let us build a one-bit ALU to support addition and logic or.

- Operation: 0 for add 1 for or

Solution

- Truth Table

- Sum of product
Supporting MIPS Logic Instructions

- MIPS provides bit-wise and, or, xor, and nor instructions.
- Input operation (3 bits) determine the output.

32-bit ALU

- Both inputs A and B are 32 bit wide.
  - Size of the truth table?

- Rather we will just cascade 32 1-bit ALU.
  - How about carries?
  - We need to refine the spec of the 1-bit ALU
Two Solutions

- Truth table and sum of product
- Use multiplexer

1-bit Adder

\[ C_{out} = AB + AC_{in} + BC_{in} \]
\[ \text{Sum} = A \text{xor} B \text{xor} C_{in} \]

- How could we build a 1-bit ALU for add, and, or?
- How could we build a 32-bit ALU?
What about subtraction \((a - b)\)?

- Two's complement approach: negate \(b\) and add.
- How do we negate?

- A clever solution:
Tailoring the ALU to the MIPS

- Need to support the set-on-less-than instruction (slt)
  - remember: slt is an arithmetic instruction
  - produces a 1 if rs < rt and 0 otherwise
  - use subtraction: \((a - b) < 0\) implies \(a < b\)

- Need to support test for equality (beq \(t5, t6, offset\))
  - use subtraction: \((a - b) = 0\) implies \(a = b\)
Test for equality

- Notice control lines:
  - 000 = and
  - 001 = or
  - 010 = add
  - 110 = subtract
  - 111 = slt

- Output zero=1 when result is 0.
Important points about hardware

- all of the gates are always working
- the speed of a gate is affected by the number of inputs to the gate
- the speed of a circuit is affected by the number of gates in series
  (on the “critical path” or the “deepest level of logic”)
- What is the critical path in our 32-bit ALU?

Ripple Carry Adder Is Slow

- Logic circuit speed is determined by the number of gates a signal have to pass in the worst case.
- Assuming each 1-bit ALU adds \( x \)-gate delay, what is the delay of a 32-bit ALU?
16-Bit Adder

Carry Look Ahead

\[ G = A \land B \]
\[ P = A \oplus B \]
\[ C_i = G_{i-1} + G_i + C_{i-1} \]
\[ C_i = P_i + G_i + C_{i-1} \]

\[ C_i = G_i + C_{i-1} + A_i \land B_i \]

Carry Look Ahead Unit
### Exercise

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Cout</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Cin “propagate”</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Cin “propagate”</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1 “generate”</td>
</tr>
</tbody>
</table>

A = B Cout

- $G = A \text{ and } B$
- $P = A \text{ xor } B$

---

### Multiplications

- $N \text{ bits } \times N \text{ bits } \rightarrow 2N \text{ bits result}$
- Paper and pencil example (unsigned):
  - Multiplicand: 0 0 1 1
  - Multiplier: 0 1 0 1
Stage $i$ accumulates $A \times 2^i$ if $B_i = 1$

Discussions

- Multiplication is expensive
- A combinational multiplier uses a great deal of silicon
  - 32 32-bit adders needed
- We will discuss designs that are slower but less silicon demanding.
  - Due to its complexity, we will first present a basic but suboptimal design, and refine it twice.
Shift and Add

One step per clock tick; \( n \) clock cycles needed for \( n \)-bit multiplications

Example: 0101 \( \times \) 0011

<table>
<thead>
<tr>
<th>Multiplicand</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Product</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiplier</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>( \text{To add or not to add} )</td>
<td></td>
</tr>
</tbody>
</table>
Unsigned Shift-Add Multiplier

- 64-bit Multiplicand reg, 64-bit ALU, 64-bit Product reg, 32-bit multiplier reg

Multiplier = datapath + control

Observations

- Half bits in multiplicand always 0
  - 64 bit adder is a waste

- Improvement:
  - Use 32 bit multiplicand
  - Don’t shift multiplicand left; shift the product right instead
Example: $0101 \times 0011$

<table>
<thead>
<tr>
<th>Multiplicand</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Product</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiplier</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Shift-Add Multiplier Version 2

- 32-bit Multiplicand reg, 32-bit ALU, 64-bit Product reg, 32-bit Multiplier reg
A Second Example

<table>
<thead>
<tr>
<th>Product</th>
<th>Multiplier</th>
<th>Multiplicand</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0000</td>
<td>0011</td>
<td>0010</td>
</tr>
<tr>
<td>0010 0000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001 0000</td>
<td>0001</td>
<td>0010</td>
</tr>
<tr>
<td>0011 0000</td>
<td>0001</td>
<td>0010</td>
</tr>
<tr>
<td>0001 1000</td>
<td>0000</td>
<td>0010</td>
</tr>
<tr>
<td>0000 1100</td>
<td>0000</td>
<td>0010</td>
</tr>
<tr>
<td>0000 0110</td>
<td>0000</td>
<td>0010</td>
</tr>
</tbody>
</table>

Observations

- Product register wastes space that exactly matches size of multiplier

- Improvement: combine Multiplier register and Product register
Example: $0101 \times 0011$

<table>
<thead>
<tr>
<th>Multiplicand</th>
<th>Product</th>
<th>To add or not to add</th>
</tr>
</thead>
</table>

Multiplier

A Second Example

<table>
<thead>
<tr>
<th>Multiplicand</th>
<th>Initial Product</th>
<th>Product after 1st shift</th>
<th>Product after 2nd shift</th>
<th>Product after 3rd shift</th>
<th>Product after 4th shift</th>
<th>Product after 5th shift</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0 1</td>
<td>0 0 0 0 0 1 0 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Multiplier Hardware Version 3**

- 32-bit Multiplicand reg, 32-bit ALU, 64-bit Product reg, (0-bit Multiplier reg)

![Diagram](image)

**Discussions**

- Can you see where the MIPS Hi and Lo registers come from?

- Can you see the special hardware associated with IA32 EAX and EDX?