Bits are just bits (no inherent meaning)
- conventions define relationship between bits and numbers
- Binary numbers (base 2)
  \[0000 0001 0010 0011 0100 0101 0110 0111 1000 1001\ldots\]
  decimal: \(0, 2^n-1\)
- Of course it gets more complicated:
  numbers are finite (overflow)
  fractions and real 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
  \[
  \begin{array}{cccc}
  000 & = +0 & 000 & = +0 & 000 & = +0 \\
  001 & = +1 & 001 & = +1 & 001 & = +1 \\
  010 & = +2 & 010 & = +2 & 010 & = +2 \\
  011 & = +3 & 011 & = +3 & 011 & = +3 \\
  100 & = -0 & 100 & = -3 & 100 & = -4 \\
  101 & = -1 & 101 & = -2 & 101 & = -3 \\
  110 & = -2 & 110 & = -1 & 110 & = -2 \\
  111 & = -3 & 111 & = -0 & 111 & = -1 \\
  \end{array}
  \]
- Issues: balance, number of zeros, ease of operations
- Which one is best? Why?

MIPS

- 32 bit signed numbers:
  \[
  \begin{array}{cccccccc}
  0000 0000 0000 0000 0000 0000 0000 0000 \rightarrow & +0 \\
  0000 0000 0000 0000 0000 0000 0000 0001 \rightarrow & +1 \\
  0000 0000 0000 0000 0000 0000 0000 0010 \rightarrow & +2 \\
  \ldots \\
  0111 1111 1111 1111 1111 1111 1111 1101 \rightarrow & +2,147,483,646 \\
  0111 1111 1111 1111 1111 1111 1111 1110 \rightarrow & +2,147,483,647 \\
  \end{array}
  \]
  \[
  \begin{array}{cccccccc}
  1000 0000 0000 0000 0000 0000 0000 0000 \rightarrow & -2,147,483,648 \\
  1000 0000 0000 0000 0000 0000 0000 0001 \rightarrow & -2,147,483,647 \\
  1000 0000 0000 0000 0000 0000 0000 0010 \rightarrow & -2,147,483,646 \\
  \ldots \\
  1111 1111 1111 1111 1111 1111 1111 1111 \rightarrow & -1 \\
  \end{array}
  \]
  \[\text{maxint} \quad \text{minint}\]

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
      \[
      \begin{array}{ll}
      0010 & \rightarrow 0000 0010 \\
      1010 & \rightarrow 1111 1010 \\
      \end{array}
      \]
    - “sign extension” (lbu vs. lb)

Addition & Subtraction

- Just like in grade school (carry/borrow 1s)
  \[
  \begin{array}{llllll}
  0111 & + 0110 & = 1001 \\
  \end{array}
  \]
- Two's complement operations easy
  - subtraction using addition of negative numbers
    \[
    \begin{array}{llll}
    0111 & + 1001 & = 1010 \\
    \end{array}
    \]
- Overflow (result too large for finite computer word):
  - e.g., adding two n-bit numbers does not yield an n-bit number
    \[
    \begin{array}{llllll}
    1111 & \rightarrow 1000 & \rightarrow 1001 \\
    \end{array}
    \]
  - note that overflow term is somewhat misleading,
    \[
    \begin{array}{llllll}
    1111 & \rightarrow 1000 & \rightarrow 1001 \\
    \end{array}
    \]
    \[\text{it does not mean a carry “overflowed”}\]

One-Bit Adder

- Takes three input bits and generates two output bits
- Multiple bits can be cascaded
Adder Boolean Algebra

• $A \cdot B \cdot C + \overline{A} \cdot \overline{B} \cdot \overline{C}$
• $0 \ 0 \ 0 \ 0 \ 0$
• $0 \ 0 \ 1 \ 0 \ 1$
• $0 \ 1 \ 0 \ 0 \ 1$
• $1 \ 0 \ 0 \ 0 \ 1$
• $1 \ 1 \ 0 \ 0 \ 0$
• $1 \ 1 \ 1 \ 1 \ 1$

$C = A \cdot B + A \cdot \overline{C} \cdot B \cdot \overline{C}$

$S = A \cdot B \cdot \overline{C} + A' \cdot B' \cdot \overline{C} + A \cdot B \cdot C' + A' \cdot B \cdot C'$

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 positive 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 $E$ 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

Real Design

• $A \quad B \quad C \quad D \quad E \quad F$
• $0 \ 0 \ 0 \ 0 \ 0 \ 0$
• $0 \ 0 \ 1 \ 1 \ 0 \ 0$
• $0 \ 1 \ 0 \ 1 \ 0 \ 0$
• $0 \ 1 \ 1 \ 1 \ 0 \ 0$
• $0 \ 1 \ 1 \ 1 \ 1 \ 0$
• $1 \ 0 \ 0 \ 1 \ 0 \ 0$
• $1 \ 0 \ 1 \ 1 \ 1 \ 0$
• $1 \ 1 \ 1 \ 1 \ 1 \ 1$

An ALU (arithmetic logic unit)

• Let’s build an ALU to support the $\text{andi}$ and $\text{ori}$ instructions
  – We’ll just build a 1-bit ALU, and use 32 of them

• Possible Implementation (sum-of-products):

Different Implementations

• Not easy to decide the “best” way to build something
  – Don’t want too many inputs to a single gate
  – Don’t 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:

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

What about subtraction \((a - b)\)?

What about subtraction \((a - b)\)?

- Two's complement approach: just negate \(b\) and add.
- How do we negate?
- A very 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, $t7)
  - use subtraction: \((a-b) = 0\) implies \(a = b\)

Supporting slt

- Can we figure out the idea?

A 32-bit ALU

- A Ripple carry ALU
- Two bits decide operation
  - Add/Sub
  - AND
  - OR
  - LESS
- 1 bit decide add/sub operation
- A carry in bit
- Bit 31 generates overflow and set bit

Test for equality

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

<Note: zero is a 1 when the result is zero!>
Problem: ripple carry adder is slow

- Is a 32-bit ALU as fast as a 1-bit ALU?
- Is there more than one way to do addition?
  - two extremes: ripple carry and sum-of-products

Can you see the ripple? How could you get rid of it?

\[
c_1 = b_0c_0 + a_0c_0 + a_0b_0
\]
\[
c_2 = b_1c_1 + a_1c_1 + a_1b_1
\]
\[
c_3 = b_2c_2 + a_2c_2 + a_2b_2
\]
\[
c_4 = b_3c_3 + a_3c_3 + a_3b_3
\]

Not feasible! Why?

Carry-look-ahead adder

- An approach in-between our two extremes
- Motivation:
  - If we didn’t know the value of carry-in, what could we do?
  - When would we always generate a carry?
    \[ g_i = a_i \times b_i \]
  - When would we propagate the carry?
    \[ p_i = a_i + b_i \]
- Did we get rid of the ripple?

\[
c_1 = g_0 + p_0c_0
\]
\[
c_2 = g_1 + p_1c_1
\]
\[
c_3 = g_2 + p_2c_2
\]
\[
c_4 = g_3 + p_3c_3
\]

Feasible! Why?

A 4-bit carry look-ahead adder

- Generate g and p term for each bit
- Use g’s, p’s and carry in to generate all C’s
- Also use them to generate block G and P
- CLA principle can be used recursively

Delays in carry look-ahead adders

- 4-Bit case
  - Generation of g and p: 1 gate delay
  - Generation of carries (and G and P): 2 more gate delay
  - Generation of sum: 1 more gate delay
- 16-Bit case
  - Generation of g and p: 1 gate delay
  - Generation of block G and P: 2 more gate delay
  - Generation of block carries: 2 more gate delay
  - Generation of bit carries: 2 more gate delay
  - Generation of sum: 1 more gate delay
- 64-Bit case
  - 12 gate delays

Use principle to build bigger adders

- A 16 bit adder uses four 4-bit adders
- It takes block g and p terms and cin to generate block carry bits out
- Block carries are used to generate bit carries
  - could use ripple carry of 4-bit CLA adders
  - Better: use the CLA principle again!

Multiplication

- More complicated than addition
  - accomplished via shifting and addition
- More time and more area
- Let’s look at 3 versions based on grade school algorithm

\[
01010010 \quad (\text{multiplicand})
\]
\[
\times 01101101 \quad (\text{multiplier})
\]

- Negative numbers: convert and multiply
- Use other better techniques like Booth’s encoding
**Multiplication**

```
01010010 (multiplicand) 01010010 (multiplicand)
01101101 (multiplier) 01101101 (multiplier)
```

```
00000000 00000000
01010010 01010010
00000000 00000000
01010010 01010010
00000000 00000000
01010010 01010010
10000101 10000101
00000000 00000000
01000010 01000010
01010010 01010010
00000000 00000000
01010010 01010010
```

```
00000000 00000000
01010010 01010010
00000000 00000000
01010010 01010010
00000000 00000000
01010010 01010010
```

**Multiplication Example**

<table>
<thead>
<tr>
<th>Iteration</th>
<th>multiplicand</th>
<th>Original algorithm</th>
<th>Step</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0010</td>
<td>Initial values</td>
<td>0100 0110</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0010</td>
<td>1:0 =&gt; no operation</td>
<td>0000 0110</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0010</td>
<td>2: Shift right Product</td>
<td>0000 0011</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>1:a:1 =&gt; prod = Prod + Mcand</td>
<td>0010 0011</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0010</td>
<td>2: Shift right Product</td>
<td>0011 0001</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0010</td>
<td>1:a:1 =&gt; prod = Prod + Mcand</td>
<td>0011 0001</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0010</td>
<td>2: Shift right Product</td>
<td>0011 1000</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0010</td>
<td>1:0 =&gt; no operation</td>
<td>0011 1000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0010</td>
<td>2: Shift right Product</td>
<td>0010 1100</td>
<td></td>
</tr>
</tbody>
</table>

**Signed Multiplication**

- Let Multiplier be Q[n-1:0], multiplicand be M[n-1:0]
- Let F = 0 (shift flag)
- Let result A[n-1:0] = 0...00
- For n-1 steps do
  - F = F .or. (M[n-1] .and. Q[0]) /* determine shift bit */
  - Shift A and Q with F, i.e.,
  - Do the correction step
  - Shift A and Q while retaining A[n-1]

This works in all cases excepts when both operands are 10...00

**Booth's Encoding**

- Numbers can be represented using three symbols, 1, 0, and -1
  - One representation is 11111111
  - Another possible one 00000000
- Another example +14
  - One representation is 00011110
  - Another possible one 00011001
- We do not explicitly store the sequence
- Look for transition from previous bit to next bit
  - 0 to 0 is 0; 0 to 1 is -1; 1 to 1 is 0; and 1 to 0 is 1
  - Multiplication by 1, 0, and -1 can be easily done
  - Add all partial results to get the final answer

**Using Booth's Encoding for Multiplication**

- Convert a binary string in Booth's encoded string
- Multiply by two bits at a time
- For n bit by n-bit multiplication, n/2 partial product
- Partial products are signed and obtained by multiplying the multiplicand by 0, +1, -1, +2, and -2 (all achieved by shift)
- Add partial products to obtain the final result
- Example, multiply 0111 (+7) by 1010 (-6)
- With 2-bit groupings, multiplication needs to be carried by -1 and -2
- Booth's encoding of 1010 is -1 -1 -1 0
- Add the two partial products to get 11010110 (-42) as result
**Booth's algorithm (Neg. multiplier)**

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Multiplier</th>
<th>Booth's algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0010</td>
<td>Initial values 0000 1101 0</td>
</tr>
<tr>
<td>1</td>
<td>0010</td>
<td>1c: Prod = Prod - Mcand 1110 1101 0</td>
</tr>
<tr>
<td></td>
<td>0010</td>
<td>2. Shift right Product 1111 0101 0</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>1b: Prod = Prod + Mcand 0001 0110 1</td>
</tr>
<tr>
<td></td>
<td>0010</td>
<td>2. Shift right Product 0000 1011 0</td>
</tr>
<tr>
<td>3</td>
<td>0010</td>
<td>1c: Prod = Prod - Mcand 1110 1011 0</td>
</tr>
<tr>
<td></td>
<td>0010</td>
<td>2. Shift right Product 1111 0101 0</td>
</tr>
<tr>
<td>4</td>
<td>0010</td>
<td>1d: No operation 1110 0101 1</td>
</tr>
<tr>
<td></td>
<td>0010</td>
<td>2. Shift right Product 1111 0101 1</td>
</tr>
</tbody>
</table>

**Carry-Save Addition**

- Consider adding six set of numbers (4 bits each in the example)
- The numbers are 1001, 0110, 1111, 0111, 1010, 0110 (all positive)
- One way is to add them pair wise, getting three results, and then adding them again

```
1001 1111 1010 10110 10001 10101
0110 0110 1110 01010 00100 00100
1011 1010 1011 10010 10100 10100
```

- Other method is add them three at a time by saving carry
  
```
 0001 0111 00000 010100 001000
 0111 0100 010100 101000 110101
```

**Division**

- Even more complicated
  - can be accomplished via shifting and addition/subtraction
- More time and more area
- We will look at 3 versions based on grade school algorithm

```
0011 0010 0010 (Dividend)
```

- Negative numbers: Even more difficult
- There are better techniques, we won’t look at them

**Non-Restoring Division**

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Divisor</th>
<th>Divide algorithm</th>
<th>Step</th>
<th>Remainder</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0010</td>
<td>Initial values</td>
<td>0000 1110</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0010</td>
<td>1a: Rem = Rem - Div (Rem &gt; 0)</td>
<td>1110 1110</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0010</td>
<td>1b: Rem = Rem + Div (Rem &lt; 0)</td>
<td>1111 1100</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>2a: Rem = Rem - Div (Rem &gt; 0)</td>
<td>1111 1000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0010</td>
<td>2b: Rem = Rem + Div (Rem &lt; 0)</td>
<td>1111 0000</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0010</td>
<td>3a: Rem = Rem - Div (Rem &gt; 0)</td>
<td>1101 0000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0010</td>
<td>3b: Rem = Rem + Div (Rem &lt; 0)</td>
<td>1101 0000</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0010</td>
<td>4a: Rem = Rem - Div (Rem &gt; 0)</td>
<td>0010 0000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0010</td>
<td>4b: Rem = Rem + Div (Rem &lt; 0)</td>
<td>0010 0000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0010</td>
<td>5a: Rem = Rem - Div (Rem &gt; 0)</td>
<td>0010 0000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0010</td>
<td>5b: Rem = Rem + Div (Rem &lt; 0)</td>
<td>0010 0000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0010</td>
<td>6a: Rem = Rem - Div (Rem &gt; 0)</td>
<td>0010 0000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0010</td>
<td>6b: Rem = Rem + Div (Rem &lt; 0)</td>
<td>0010 0000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0010</td>
<td>Shift left half of Rem right 1</td>
<td>0001 0011</td>
<td></td>
</tr>
</tbody>
</table>

**Division**

- Even more complicated
  - can be accomplished via shifting and addition/subtraction
- More time and more area
- We will look at 3 versions based on grade school algorithm

```
0011 0010 0010 (Dividend)
```

- Negative numbers: Even more difficult
- There are better techniques, we won’t look at them