Arithmetic

- Bits are just bits (no inherent meaning)
- Designing an ALU
  - It performs many functions
    - Each function may be implemented separately and then a result is selected based on need
    - A multiplexer is useful device for that

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
  - "sign extension"
    - To convert n bits to k bits k>n

An ALU (arithmetic logic unit)

- Let's build an ALU to support AND, OR, ADD, SUB and other functions
- we'll just build a 1 bit ALU, and use 32 of them

32-bit numbers

0000 0000 0000 0000 0000 0000 0000 0000 0000 = 0
0000 0000 0000 0000 0000 0000 0000 0001 = + 1
0000 0000 0000 0000 0000 0000 0000 0010 = + 2
... 0111 1111 1111 1111 1111 1111 1111 1110 = + 2,147,483,646
0111 1111 1111 1111 1111 1111 1111 1111 = + 2,147,483,647
1000 0000 0000 0000 0000 0000 0000 0000 = – 2,147,483,648
1000 0000 0000 0000 0000 0000 0000 0001 = – 2,147,483,647
1000 0000 0000 0000 0000 0000 0000 0010 = – 2,147,483,646
... 1111 1111 1111 1111 1111 1111 1111 1101 = – 3
1111 1111 1111 1111 1111 1111 1111 1110 = – 2
1111 1111 1111 1111 1111 1111 1111 1111 = – 1

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 ?
- An exception (interrupt) occurs when overflow occurs
- Details based on software system / language
  - example: flight control vs. homework assignment
- Don’t always want to detect overflow

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:

```
<table>
<thead>
<tr>
<th>Operation</th>
<th>CarryIn</th>
<th>CarryOut</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add/Sub</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AND</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Other/LESS</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Supporting Other Function

- Can we figure out the idea?

A 32-bit ALU

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

Faster Adder: 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
- Use them to generate block G and P
- CLA principle is used recursively

Use principle to build bigger adders

- A 16 bit adder uses four 4-bit adders
- If 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!

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