Computer Arithmetic

Chapter Three P&H

Data Representation

- Why do we not encode numbers as strings of ASCII digits inside computers?

Data Representation

- What is ***overflow*** when applied to binary operations on data?

Data Representation

- Why do we not use signed magnitude to represent numbers inside computers?

Data Representation

- What is the two’s compliment number representation?

Data Representation

- How is a two’s compliment number sign extended?

Data Representation

- Why does MIPS have:
  - lb and lbu instructions?
  - slt and sltu instructions?

Addition and Subtraction

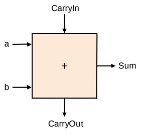
- No overflow possible when:
  - Adding numbers with different signs
  - Subtracting numbers with same sign
  - one of numbers is zero
- Overflow occurs when:
  - Adding two numbers with same sign and sign of result is different
  - Subtracting numbers with different signs & result is the same sign as second number
- MIPS handles overflow with an exception

MIPS ALU Design

- MIPS ALU requirements
  - add, addu, sub, subu, addi, addiu
    - => 2's complement adder/sub with overflow detection
  - and, or, andi, ori, xor, xori, nor
    - => Logical AND, logical OR, XOR, nor
  - SLTI, SLTIU (set less than)
- => 2's complement adder with inverter, check sign bit of result

Different Implementations

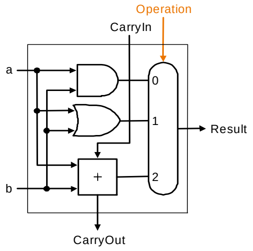
- Not easy to decide the "best" way to build something
  - Don't want too many inputs to a single gate
  - Don't want to have to go through too many gates
  - for our purposes, ease of comprehension is important
- Let's look at a 1-bit ALU for addition:



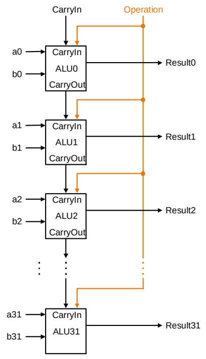
$$c_{out} = a\ b + a\ c_{in} + b\ c_{in}$$
$$sum = a \text{ xor } b \text{ xor } c_{in}$$

- 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



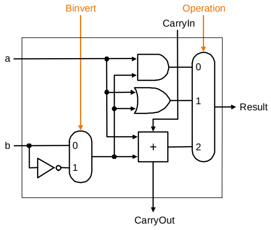
Building a 32 bit ALU



What about subtraction (a – b) ?

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

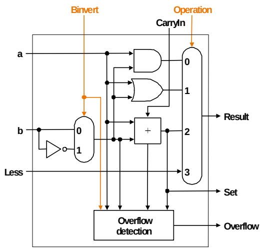
- A very clever solution:

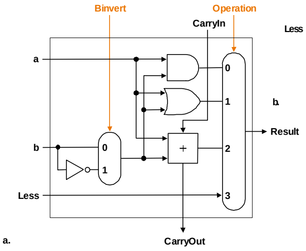


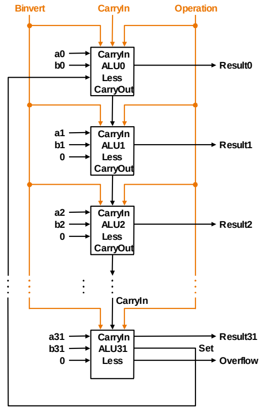
Tailoring the ALU to the MIPS

- 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





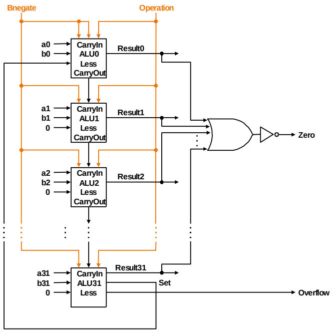


Test for equality

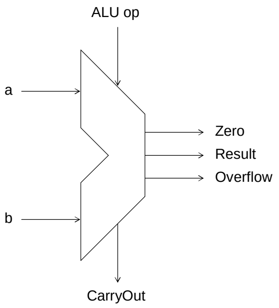
- Notice control lines:

```
000 = and
001 = or
010 = add
110 = subtract
111 = slt
```

•Note: zero is a 1 when the result is zero!

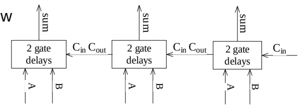


ALU symbol



Addition

- Ripple adders are slow



- What about sum-of products representation?

$c_1 = b_0c_0 + a_0c_0 + a_0b_0$

$c_2 = b_1c_1 + a_1c_1 + a_1b_1$ $\qquad c_2 =$

$c_3 = b_2c_2 + a_2c_2 + a_2b_2$ $\qquad c_3 =$

$c_4 = b_3c_3 + a_3c_3 + a_3b_3$ $\qquad c_4 =$

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? $\qquad g_i = a_i\ b_i$
  - When would we propagate the carry? $\qquad p_i = a_i + b_i$

$c_1 = g_0 + p_0c_0$

$c_2 = g_1 + p_1c_1$ $\qquad c_2 =$

$c_3 = g_2 + p_2c_2$ $\qquad c_3 =$

$c_4 = g_3 + p_3c_3$ $\qquad c_4 =$

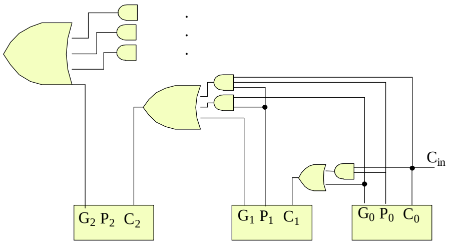
Example

```
a:  0001 1010 0011 0011
b:  1110 0101 1110 1011

gi:
pi:

P0, P1, P2, P3 ← super propagate.
G0, G1, G2, G3 ← super generate.
C4             ← what’s that?
```

Carry Look-ahead Adder



Conclusion

- We can build an ALU to support the MIPS instruction set
  - key idea: use multiplexer to select the output we want
  - we can efficiently perform subtraction using two’s complement
  - we can replicate a 1-bit ALU to produce a 32-bit ALU
- 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”)
- Our primary focus: comprehension, however,
  - Clever changes to organization can improve performance (similar to using better algorithms in software)