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?
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!
Conclusion

- We can build an ALU to support the MIPS instruction set
  - key idea: use multiplexor 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)
  - we’ll look at two examples for addition and multiplication

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?

\[
\begin{align*}
    c_1 &= b_0c_0 + a_0c_0 + a_0b_0 \\
    c_2 &= b_1c_1 + a_1c_1 + a_1b_1 \\
    c_3 &= b_2c_2 + a_2c_2 + a_2b_2 \\
    c_4 &= b_3c_3 + a_3c_3 + a_3b_3
\end{align*}
\]

Not feasible! Why?
Carry-lookahead 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?  
  - When would we propagate the carry?  
- Did we get rid of the ripple?

\[
\begin{align*}
c_1 &= g_0 + p_0 c_0 \\
c_2 &= g_1 + p_1 c_1 \\
c_3 &= g_2 + p_2 c_2 \\
c_4 &= g_3 + p_3 c_3 \\
\end{align*}
\]

Feasible! Why?

Use principle to build bigger adders

- Can’t build a 16 bit adder this way... (too big)
- Could use ripple carry of 4-bit CLA adders
- Better: use the CLA principle again!