Performance is specific to a particular program/s
- Total execution time is a consistent summary of performance

For a given architecture performance increases come from:
- Increases in clock rate (without adverse CPI affects)
- Improvements in processor organization that lower CPI
- Compiler enhancements that lower CPI and/or instruction count
- Algorithm/Language choices that affect instruction count

Pitfall: expecting improvement in one aspect of a machine’s performance to affect the total performance

Remember

Almost ready to move into chapter 5 and start building a processor
First, let’s review Boolean Logic and build the ALU we’ll need
(Material from Appendix B)

Let’s Build a Processor

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 these three functions.
Show an implementation consisting of inverters, AND, and OR gates.

Review: Boolean Algebra & Gates

An ALU (arithmetic logic unit)

Let’s build an ALU to support the andi and orl instructions
- We’ll just build a 1 bit ALU, and use 32 of them

Possible Implementation (sum-of-products):
Review: The Multiplexor

- Selects one of the inputs to be the output, based on a control input

\[ \text{MUX input select} \]

\[ A \rightarrow \text{S} \rightarrow B \rightarrow C \]

note: we call this a 2-input mux even though it has 3 inputs!

- Lets build our ALU using a MUX:

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:

\[ \text{cout} = a \cdot b + a \cdot \text{cin} + b \cdot \text{cin} \]

\[ \text{sum} = a \oplus b \oplus \text{cin} \]

- 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)?

- Two’s complement approach: just negate b and add.
- How do we negate?
- A very clever solution:
Adding a NOR function

• Can also choose to invert a. How do we get "a NOR b"?

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?

Use this ALU for most significant bit

all other bits
Test for equality

- Notice control lines:

  0000 = and
  0001 = or
  0010 = add
  0110 = subtract
  0111 = slt
  1100 = NOR

  *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 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)
  - We saw this in multiplication, let's look at addition now

---

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?

\[
\begin{align*}
c_1 &= b_2c_0 + a_2c_0 + a_2b_2 \\
c_2 &= b_2c_1 + a_2c_1 + a_2b_1c_2 \\
c_3 &= b_2c_3 + a_2c_3 + a_2b_2c_1 \\
c_4 &= b_2c_4 + a_2c_4 + a_2b_2c_2 \\
\end{align*}
\]

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? \( g_i = a_i b_i \)
  - When would we propagate the carry? \( p_i = a_i + b_i \)

- Did we get rid of the ripple?

\[
\begin{align*}
c_1 &= g_0 + p_0c_0 \\
c_2 &= g_1 + p_0c_1 c_2 \\
c_3 &= g_2 + p_0c_2 c_3 \\
c_4 &= g_3 + p_0c_3 c_4 \\
\end{align*}
\]

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!