Appendix C

Review of Boolean Logic and the ALU Design

New Paltz

Baback Izadi
ECE Department
bai@engr.newpaltz.edu
Review: Boolean Algebra & Gates

- 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: The Multiplexor

- 1-out-2 MUX

\[ S \]

\[ A \rightarrow 0 \quad \rightarrow B \rightarrow 1 \rightarrow O \]

\[ \begin{array}{c|c}
S & O \\
0 & A \\
1 & B \\
\end{array} \]

- \( C = \overline{S} A + S B \)
- Let’s build our ALU using MUX’s:
Building a 32 bit ALU

Let's look at a 1-bit ALU for addition:

\[
\text{Sum} = a \oplus b \oplus c_{in}
\]
\[
C_{out} = a \cdot b + (a \oplus b) \cdot c_{in}
\]
\[
C_{out} = a \cdot b + a \cdot c_{in} + b \cdot c_{in}
\]

How could we build a 1-bit ALU for add, and, and or?
What about subtraction \((a - b)\) ?

- Two's complement approach:
- \(a - b = a + \overline{b} + 1\)
Flags

- Carry into MSB XOR Carry out of MSB
- For a N-bit ALU: Overflow = CarryIn[N - 1] ⊕ CarryOut[N - 1]
Supporting \textit{slt}

- Need to support the set-on-less-than instruction (\textit{slt})
  - remember: \textit{slt} is an arithmetic instruction
  - produces a 1 if \( rs < rt \) and 0 otherwise
  - use subtraction: \((a-b) < 0\) implies \( a < b \) and use sign bit
Test for equality

• Notice control lines:

000 = and
001 = or
010 = add
110 = subtract
111 = slt
module MIPSALU (ALUctrl, A, B, ALUOut, Zero);
  input [3:0] ALUctrl;
  input [31:0] A,B;
  output reg [31:0] ALUOut;
  output Zero;
  assign Zero = (ALUOut==0); //Zero is true if ALUOut is 0; goes anywhere
  always @(ALUctrl, A, B) //reevaluate if these change
    case (ALUctrl)
      0: ALUOut <= A & B;
      1: ALUOut <= A | B;
      2: ALUOut <= A + B;
      6: ALUOut <= A - B;
      7: ALUOut <= A < B ? 1:0;
      12: ALUOut <= ~(A | B); // result is nor
    default: ALUOut <= 0; //default to 0, should not happen;
    endcase
  endmodule
MIPS ALU Summary

- 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
  - 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 with Ripple Carry

- 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?

\[
\begin{align*}
  c_1 &= b_0 c_0 + a_0 c_0 + a_0 b_0 \\
  c_2 &= b_1 c_1 + a_1 c_1 + a_1 b_1 \\
  c_3 &= b_2 c_2 + a_2 c_2 + a_2 b_2 \\
  c_4 &= b_3 c_3 + a_3 c_3 + a_3 b_3 \\
\end{align*}
\]

Not feasible! Why?
Carry-lookahead adder

- An approach in-between our two extremes

\[ c_1 = b_0c_0 + a_0c_0 + a_0b_0 = (b_0 + a_0)c_0 + a_0b_0 \]

  - 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?

\[
\begin{align*}
  c_1 &= g_0 + p_0c_0 \\
  c_2 &= g_1 + p_1c_1 \\
  c_3 &= g_2 + p_2c_2 \\
  c_4 &= g_3 + p_3c_3
\end{align*}
\]
Carry Look Ahead Design trick

\[ c_{in} \]

\[ \begin{align*}
    a_0 & \quad g & \quad p & \quad S \\
    b_0 & & & \\
\end{align*} \]

\[ \begin{align*}
    a_1 & \quad g & \quad p & \quad S_1 \\
    b_1 & & & \\
\end{align*} \]

\[ \begin{align*}
    a_2 & \quad g & \quad p & \quad S \\
    b_2 & & & 2 \\
\end{align*} \]

\[ \begin{align*}
    a_3 & \quad g & \quad p & \quad S_3 \\
    b_3 & & & \\
\end{align*} \]

\[ c_1 = g_0 + p_0 c_0 \]

\[ c_2 = g_1 + p_1 g_0 + p_1 p_0 c_0 \]

\[ c_3 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0 \]

\[ c_4 = \ldots \]

\[ g = a \cdot b \]

\[ p = a + b \]
16 Bit Carry Look Ahead

- \( g_i = a_i \cdot b_i \)
- \( p_i = a_i + b_i \)
- \( c_1 = g_0 + p_0 \cdot c_0 \)
- \( c_2 = g_1 + p_1 \cdot g_0 + p_1 \cdot p_0 \cdot c_0 \)
- \( c_3 = g_2 + p_2 \cdot g_1 + p_2 \cdot p_1 \cdot g_0 + p_2 \cdot p_1 \cdot p_0 \cdot c_0 \)
- \( c_4 = g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot p_2 \cdot p_1 \cdot g_0 + p_3 \cdot p_2 \cdot p_1 \cdot p_0 \cdot c_0 \)

\[
\begin{align*}
G_0 &= g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot p_2 \cdot p_1 \cdot g_0 \\
P_0 &= p_3 \cdot p_2 \cdot p_1 \cdot p_0 \\
G_1 &= g_7 + p_7 \cdot g_6 + p_7 \cdot p_6 \cdot g_5 + p_7 \cdot p_6 \cdot p_5 \cdot g_4 \\
P_1 &= p_7 \cdot p_6 \cdot p_5 \cdot p_4 \\
G_2 &= g_{11} + p_{11} \cdot g_{10} + p_{11} \cdot p_{10} \cdot g_9 + p_{11} \cdot p_{10} \cdot p_9 \cdot g_8 \\
P_2 &= p_{11} \cdot p_{10} \cdot p_9 \cdot p_8 \\
G_3 &= g_{15} + p_{15} \cdot g_{14} + p_{15} \cdot p_{14} \cdot g_{13} + p_{15} \cdot p_{14} \cdot p_{13} \cdot g_{12} \\
P_3 &= p_{15} \cdot p_{14} \cdot p_{13} \cdot p_{12}
\end{align*}
\]
16 Bit Carry Look Ahead (Cont.)

- \( C_1 = G_0 + P_0 C_0 \)
- \( C_2 = G_1 + P_1 C_1 \)
- \( C_3 = G_2 + P_2 C_2 \)
- \( C_4 = G_3 + P_3 C_3 \)

- \( C_1 = c_4 = G_0 + P_0 c_0 \)
- \( C_2 = c_8 = G_1 + P_1 G_0 + P_1 P_0 c_0 \)
- \( C_3 = c_{12} = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 c_0 \)
- \( C_4 = c_{16} = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 c_0 \)
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!
Cascaded Carry Look-ahead (16-bit)

\[ C_0 = G_0 + P_0 C_0 \]

\[ C_2 = G_1 + P_1 G_0 + P_1 P_0 C_0 \]

\[ C_3 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0 \]

\[ C_4 = \ldots \]
Example: Determine the $g_i$, $p_i$, $P_i$, and $G_i$ values of the following two 16 bit numbers. What is $C_{out_{15}}$ ($C_{16}$)?

a:
0001 1010 0011 0011

b:
1110 0101 1110 1011

$p_i = a_i + b_i$

$g_i = a_i \cdot b_i$

$c_i$

• Repeat Using $P_i$ and $G_i$

$P_0 =$

$G_0 =$

$G_1 =$

$G_2 =$

$G_3 =$

$P_3 =$

$C_4 =$
Speed of Ripple Carry Versus Carry Lookahead

- One simple way to model time for logic is to assume each AND and OR gate takes the same time for a signal to pass through it. Time is estimated by simply counting the number of gates along the longest path through a piece of logic. Compare the number of gate delays for the critical paths of two 16-bit adders, one using ripple carry and one using two-level carry lookahead.
Other Design Tricks: Guess

Cout

Carry-select adder

n-bit adder

n-bit adder

n-bit adder

n-bit adder

n-bit adder

n-bit adder

n-bit adder

1

0

Cout

Carry-select adder
ALU Summary

- We can build an ALU to support MIPS addition
- Our focus is on comprehension, not performance
- Real processors use more sophisticated techniques for arithmetic
- Where performance is not critical, hardware description languages allow designers to completely automate the creation of hardware!