EEC 483 Computer Organization

Chapter 3. Arithmetic for Computers

Chansu Yu

Cleveland State University

Table of Contents

- Ch.1 Introduction
- Ch. 2 Instructions: Language of the Computer
- Ch. 3 CPU Implementation: Arithmetic
  - 3.1 Introduction
  - 3.2 Addition and subtraction
  - 3.3 Multiplication
  - 3.4 Division
  - 3.5 Floating point
  - Appendix C.5 Constructing an Arithmetic Logic Unit (ALU)
- Ch. 4 CPU Implementation: Pipeline
- Ch. 5 Cache and Virtual Memory
- Ch. 6-7 I/O and Multiprocessors

c.yu91@csuohio.edu
Show the truth table for these three functions.

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.

1-bit andi and ori

3-input implementation (a,b,op)
A different implementation with:
The Multiplexor

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

```
   A   B   C
   0   1   S
```

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

- Lets build our ALU using a MUX: (1-bit and/or)

```
   A   B   S
   0   1   C
```

Latency: 1 gate delay (gd) (assuming no MUX delay)

A different implementation with:
The Multiplexor

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

```
   A   B   C
   0   1   S
```

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

- Lets build our ALU using a MUX: (1-bit and/or)

```
   A   B   S
   0   1   C
```

Latency: 1 gate delay (gd) (assuming no MUX delay)

1-bit ALU for Addition

- Boolean equation for carryout ???
- Boolean equation for sum ???
- Implementation ???

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

Latency: 2 gds for Cout & 1 gd for sum
1-bit ALU for And/Or/Addition

- How could we build a 1-bit ALU for add, and, and or?

- How could we build a 32-bit ALU?
What about subtraction \((a - b)\)?

- Two's complement approach: just negate \(b\) and add.
- How do we negate \((-b = b^\text{bar} + 1)\)?
- A very clever solution:
  - \(b^\text{bar}\): invert each bit
  - ++1: carry in to each bit (NO!!!) \(\Rightarrow\) just carry in to the first bit only

Revisit: 2’s or 1’s Complement

- Subtraction: \(a-b\) (\(a,b > 0\))
  - Two’s Complement
    - If \(a\geq b\): answer = \((-b-a)\), where \((b-a)>0\)
    - \(a-b = a+(b^\text{bar}+1) = 2^b-(b-a)\)
      - This is exactly the representation of \((-b-a)\): OK
    - If \(a\geq b\): answer = \((a-b)\), where \((a-b)>0\)
      - \(a-b = a+(b^\text{bar}+1) = 2^b-(b-a)+2^b\): This is not the representation of \((a-b)\), but \(2^b\) is just ignored: OK
  - One's Complement
    - If \(a\geq b\): answer = \((-b-a)\), where \((b-a)>0\)
      - \(a-b = a+(b^\text{bar}+1)=2^b-(b-a)-1\)
        - This is exactly the representation of \((-b-a)\): OK
    - If \(a\geq b\): answer = \((a-b)\), where \((a-b)>0\)
      - \(a-b = a+(b^\text{bar}+1)=2^b-(b-a)+2^b\): This is not the representation of \((a-b)??\)
        - \(2^b\) is just ignored, but “-1” must be compensated
      - \(=>\) if there is a carry out from the last digit, add 1 to the first digit
  - 1’s complement requires one more addition !!!
Remember that all AND/OR/ADD circuits are operating, but it outputs only the result of the selected operation.

What’s more

- MIPS instructions
  - add/ sub/ and/ or/ lw/ sw/ beq/ bne/ slt/ j/ jr/ ...
- We’re covering ALU instructions in Ch.4
  - add/ sub/ and/ or
- What other instructions require ALU operations?
  - All inst: PC calculation: PC=PC+4, PC=PC+jump offset
  - Lw/sw: Address calculation: lw $s0, 100($t1)
  - Slt: slt $s0, $s1, $s2: if $s1<$s2, set $s0=1
    => if ($s1-$s2)<0: “subtraction” + some other activities
  - Beq/bne: beq $s0, $s1, label: if $s0=$s1, jump to label
    => if ($s0=$s1)=0: “subtraction” + some other activities
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 \( a < b \)
  - => perform subtraction & see which bit?? (sign bit or MSB)
  - => set the destination register with the value of MSB!!

Supporting slt

- 32-bit result with slt inst.
  - 000...0000 or 000...0001
  - All other bits = 0
  - Bit 0 (LSB) = 1 or 0 depending on the comparison (subtraction)

- One more input for operation
  - 0:and, 1:or, 2:add/sub, 3:slt
  - For (3:slt) operation, “less” input is selected
    - “Less” input for all other bits = 0
    - “Less” input for LSB = sign bit (MSB) after the subtraction = “Set” bit

- But how to “sub” circuit (2) operate when we select the “slt” operation (3)
  - All circuits are always working
  - Just need to input as if it is subtract operation (binvert=1 & carryin =1)
Notice that “Set” from MSB 1-bit ALU is fed to “Less” input of LSB 1-bit ALU.

How many gate delays to get r0?

\[ c1 = 2gd \\
\[ c2 = 4gd \\
\[ \ldots \\
\[ c32 = 64gd \]

Thus, \( r0' = 63gd \)

---

Test for equality (BEQ)

- Need to support test for equality (beq $t5, $t6, $t7)
  - Use subtraction: \((a-b) = 0\) implies \(a = b\)
  - Which means all bits is zero
  - “Zero” output signal !!!
  - Operation = add/sub
  - carryin = 1
  - binvert = 1

Is that all? Then, what?

\( r0\rightarrow r31 \) must carry the result of subtraction (in slt, \( r0\rightarrow r31 \) Do not carry the result of the subtraction)
Supporting MIPS ISA

- 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 (AND/ OR/ ADD or SUB)
  - 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

Latency: 1 gd for and/or & 64 gds for add
(63 for r’s, 64 for r’s & cout’s)
⇒ “add” limits the ALU performance

Ripple carry adder in 32-bit ALU

\[
\begin{align*}
c_{\text{out}} &= a \oplus b + a \oplus c_{\text{in}} + b \oplus c_{\text{in}} \\
\text{sum} &= a \oplus b \oplus a \oplus c_{\text{in}} \\
&\Rightarrow 2 \text{ gate delays (gds)}
\end{align*}
\]

Overall 64 gds from carryin (c0) to carryout (c31) & r31
Problem: ripple carry adder is slow (RCA)

- Can you see the ripple? How could you get rid of the dependency?
  - Remove ci’s by substitution !!!
    
    \[
    \begin{align*}
    c_1 &= b_0c_0 + a_0c_0 + a_0b_0 \\
    c_2 &= b_1c_1 + a_1c_1 + a_1b_1 \\
    c_3 &= b_2c_2 + a_2c_2 + a_2b_2 \\
    c_4 &= b_3c_3 + a_3c_3 + a_3b_3 \\
    \vdots \\
    c_{31} &= b_{31}c_{31} + a_{31}c_{31} + a_{31}b_{31} \\
    c_{31} &= f(a, b, c, 0) = \text{“TOO MANY TERMS !!!”}
    \end{align*}
    \]

Not feasible! Why?

- => Gate delays ???? (=2) : OK
- => But, many-input gate is required
  (e.g. 100-input and gate)

Carry-lookahead adder (CLA)

- An approach in-between our two extremes
  - 64 gds with 2-input “and” + 3-input “or”
  - 2 gds with 33-input “and” + (2^32-1)-input “or”

- Motivation:
  - If we didn’t know the value of carry-in, what could we do?
    - \( c_{i+1} = b_i c_i + a_i c_i + a_i b_i \)
    - \( g_i = a_i \cdot b_i \)
  - When would we always generate a carry?
    - \( p_i = a_i + b_i \)
    - \( c_{i+1} = (a_i + b_i) c_i + (a_i b_i) = p_i c_i + g_i \)
    - If \( g_i = 1 \), there always is a carry out
    - If \( g_i = 1 \), carryout is propagated from the previous stage

- Did we get rid of the ripple?
  \[
  \begin{align*}
  c_1 &= g_1 + p_1 c_0 \\
  c_2 &= g_1 + p_1 c_1 \\
  c_3 &= g_1 + p_1 c_2 \\
  c_4 &= g_1 + p_1 c_3 \\
  c_5 &= g_1 + p_1 c_4 \quad (5 \text{ terms})
  \end{align*}
  \]

Feasible! Why?
4-bit adder: RCA vs CLA

This is important because the next stage can start whenever \( c4 \) is available.

\( \text{RCA} \)

\[ a0-a3, b0-b3, c0 \]

\[ \text{c1} = \text{g0} + \text{p0} \text{c0} \]
\[ \text{c2} = \text{f(} g1, \text{p1, c0)} \]
\[ \text{c3} = \text{f(} g2, \text{p2, c0)} \]
\[ \text{c4} = \text{f(} g3, \text{p3, c0)} \]

\( \text{CLA} \)

\[ \text{c16: 17-input “and” + 17-input “or”} \]

But 16-bit CLA is too BIG to implement!!
16-bit adder: 16-bit RCA vs (RCA of CLAs) vs (CLA of CLAs)

Cleveland State University

c.yu91@csuohio.edu

16-bit adder: 16-bit RCA vs (RCA of CLAs) vs (CLA of CLAs)

Cleveland State University

c.yu91@csuohio.edu
16-bit adder: CLA of CLAs

- Could use ripple carry of 4-bit CLA adders
- Better: use the CLA principle again!

16-bit adder: 16-bit RCA vs (RCA of CLAs) vs (CLA of CLAs)

- CLA of CLA
Carry Look-ahead Adder (CLA)

- Basic quantities and equations
  - \( a_i \leftarrow a_{i+1}, b_i \leftarrow b_{i+1}, c_i \)
  - \( c_{i+1} = a_i b_i + a_i c_i + b_i c_i \) and \( t_i = a_i \oplus b_i \oplus c_i \)

- Generation and propagation signals
  - \( c_{i+1} = a_i b_i + a_i c_i + b_i c_i = (a_i + b_i) c_i + (a_i b_i) = p_i c_i + g_i \)
  - Where, \( p_i = a_i + b_i \) and \( g_i = a_i b_i \)

- RCA of CLAs
  - \( c_1 = g_1 + p_1 c_0 \) (2 terms)
  - \( c_2 = g_1 + p_1 c_1 \)
  - \( c_3 = g_2 + p_2 c_1 \)
  - \( c_4 = g_3 + p_3 c_1 \)

  \( d_i = \frac{c_i + g_i}{2} \) for \( i = 1, 2, 3 \)

  \( c_i - c_i \) depends on \( p_i, g_i \), and \( c_{i-1} \). So does \( r_i - r_{i-1} \).

16-bit adder: RCA of CLAs

Every fourth carry is “ripple propagated” to the next bit but other carries are calculated four at a time.
16-bit adder: CLA of CLAs

Equations

From above, \( c_8 = g_3 + p_3, g_4 + p_4, g_5 + p_5, g_6 + p_6, g_7 + p_7, c_0 = G_6 + P_6, c_0 \) (2 terms)

where, \( G_6 = g_3 + p_3, g_4 + p_4, g_5 + p_5, g_6, P_6 = p_3, p_4, p_5 \)

Similarly, \( c_3 = G_1 + P_1, C_3 = G_1 + P_1, C_3 = G_1 + P_1, G_0 + P_1, C_3 = 3 \) terms

\( c_1 = G_1 + P_1, C_1 = G_1 + P_1, G_1 + P_1, P_1, G_1 + P_1, 4 \) terms

\( c_1 = G_1 + P_1, C_1 = G_1 + P_1, G_1 + P_1, P_1, G_1 + P_1, P_1, P_1, P_1, P_1, c_0, 5 \) terms

ALU Summary

- 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!
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!

```verilog
module MIPSALU (a, b, aluOut, zero);
    input [3:0] aluOut;
    output [11:0] aluOut;
    output zero;
    assign zero = !aluOut[15]; // Zero if true if ALUOut is 0 or goes anywhere
    always @(*) begin
        case(aluOut)
            0: aluOut = a & b;
            1: aluOut = a | b;
            2: aluOut = a & b;
            4: aluOut = a - b;
            7: aluOut = a <= b ? 1'd0 : 1'd1;
            12: aluOut = a > b ? 1'd0 : 1'd1; // result is err
        default: aluOut = 0; // defaults to 0, should not happen;
        endcase
    endmodule
```

Cleveland State University

c.yu91@csuohio.edu