COEN-2710 Microprocessors - Lecture 6

Arithmetic for Computers – Building the ALU (Ch.3 + Appendices A,C)

Cristinel Ababei
Marquette University
Department of Electrical and Computer Engineering

Outline

- Basics
- ALU
- Carry look ahead
- Shifting

Design process summary

- 1. Divide and Conquer:
  - Outline what needs to be accomplished
  - Formulate solution in terms of simpler components
  - Design each component
- 2. Connect, test, and verify:
  - Put together the basic building blocks
  - Verify that every possible input gives valid output
- 3. Successive refinement:
  - Evaluate results, correct errors, improve design

Design Methodologies

- Hierarchical Design to manage complexity
- Top Down vs. Bottom Up
  - Block Diagrams
  - Decomposition into Bit Slices
  - Truth Tables, K-Maps
  - Circuit Diagrams
  - Other: state & timing diagrams, ...
- Measurement Criteria:
  - Design Performance
  - Design Cost
  - Design Time
  - Gate count
  - Power dissipation
  - ...

First step: arithmetic circuits

Questions to be addressed:

- **How do we represent numbers?**
  - **Integers vs. floating-point (i.e., real numbers)**
  - **Negative numbers**
- **How do we implement:**
  - **Addition?**
  - **Subtraction?**
  - **Multiplication?**
  - **Division?**
- **How do we handle errors? (overflow, etc.)**

5

Possible Representations

Sign Magnitude:	One's Complement	Two's Complement
000 = +0	000 = +0	000 = +0
001 = +1	001 = +1	001 = +1
010 = +2	010 = +2	010 = +2
011 = +3	011 = +3	011 = +3
100 = −0	100 = −3	100 = −4
101 = −1	101 = −2	101 = −3
110 = −2	110 = −1	110 = −2
111 = −3	111 = −0	111 = −1

- **Issues: balance, number of zeros, ease of operations**
- **Which one is best? Why?**

6

RISC-V - Two's complement

- Positive numbers - regular binary
- Negative numbers
  - Take equivalent positive binary number
  - Flip the bits
  - Add 1
- Significant design advantages
  - Easy to negate numbers (steps given above)
  - Easy to check positive/negative (still have sign bit)
  - Biggest one: adding and subtracting work right!

Sign extension

We will often need to take a 16-bit number and put it into a 32-bit (or 64-bit) location.

Doing this in two's complement requires some slight additional care:

- Must always replicate the left-most bit (sign bit) into the other positions.
- This process is called sign-extension, and is built into most commands

Unsigned numbers

- Sometimes we have variables (like counters) that can only be positive. By **not** using two's complement for these cases, we get a whole extra bit of representation.
- Positive-only numbers are called ***unsigned***.
- Math commands on unsigned numbers will have to be handled carefully, both in design and in programming (especially with the built-in sign extension)

Addition & Subtraction

- Two's complement operations easy
  - subtraction using addition of negative numbers

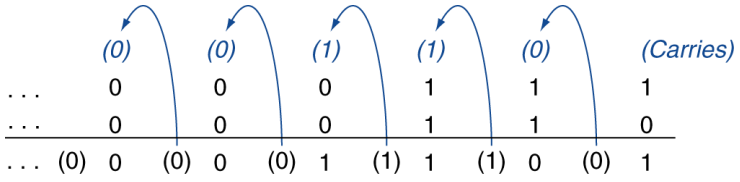
```
  0111
+ 1010
```

- Overflow (result too large for finite computer word):
  - e.g., adding two n-bit numbers does not yield an n-bit number

```
  0111
+ 0001
  1000
```

Integer Addition

- **Example: 7 + 6**



- Overflow if result out of range
  - Adding positive and negative operands, no overflow
  - Adding two positive operands
    - Overflow if result sign is 1
  - Adding two negative operands
    - Overflow if result sign is 0

11

Integer Subtraction

- Add negation of second operand
- **Example: 7 – 6 = 7 + (–6)**

  +7: 0000 0000 … 0000 0111
  –6: 1111 1111 … 1111 1010
  +1: 0000 0000 … 0000 0001

- Overflow if result out of range
  - Subtracting two positive or two negative operands, no overflow
  - Subtracting positive from negative operand
    - Overflow if result sign is 0
  - Subtracting negative from positive operand
    - Overflow if result sign is 1

12

Practice

1. Represent +14, -14, +25, and –25 in *6-bit* two's complement notation
2. Using these representations,
   a) Add -14 + -14
   b) Add 14 + -25
   c) Add 14 + 25
3. Any overflows?

Detecting Overflow

- No overflow when adding a positive and a negative number
- No overflow when signs are the same for subtraction
- Overflow occurs when the value affects the sign:
  - adding two positives yields a negative
  - adding two negatives gives a positive
  - subtract a negative from a positive and get a negative
  - subtract a positive from a negative and get a positive
- How to handle overflow:
  - Control jumps to predefined address for exception
  - Interrupted address is saved for possible resumption
  - Ignore it (several commands for unsigned arithmetic do not cause exceptions on overflow)

Outline

- Basics
- ALU
- Carry look ahead
- Shifting

The Arithmetic Logic Unit (ALU)

- Will use a bottom-up approach to design a 32-bit ALU (64-bit is similar)
- Based on the chosen instruction set, we need to be able to implement:
  - Addition
  - Subtraction
  - Bit-wise AND
  - Bit-wise OR
  - Set on less-than
  - Zero-flag (BNE)
  - Nor

  } Will be integrated into ALU

  - Shift left
  - Shift right
  - Multiplication
  - Division

  } (Will do later)

Building blocks: Start with 1-bit ALU

Start with just AND, OR, ADD functions

ALU inputs:

- **Input A, Input B, Carry In, Control flags (for deciding what operation is implemented)**

ALU outputs:

- **Result, Carry Out**

❖ ***Practice*: Work out the logic formulas for Add from the basic concept of addition**

ALU formulas

AND

- **Result = AB**

OR

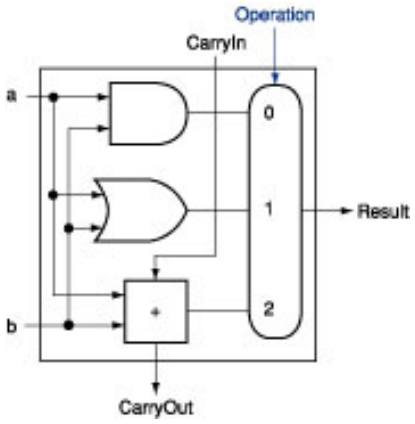
- **Result = A+B**

ADD

- **Result =** $\overline{A}\overline{B}C_i + A\overline{B}\overline{C_i} + \overline{A}B\overline{C_i} + ABC_i$
- **Carry Out =** $AB + AC_i + BC_i$

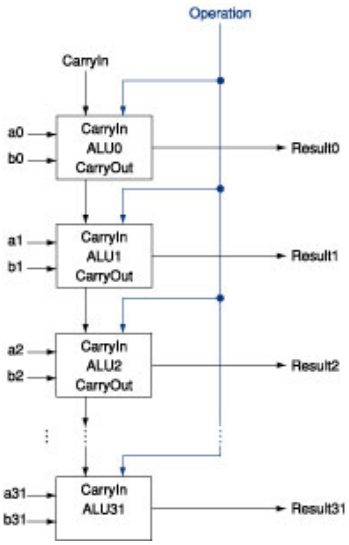
First-pass ALU Implementation

- Implement AND, OR, ADD logic
- Use 3-input mux to select output function (Operation)



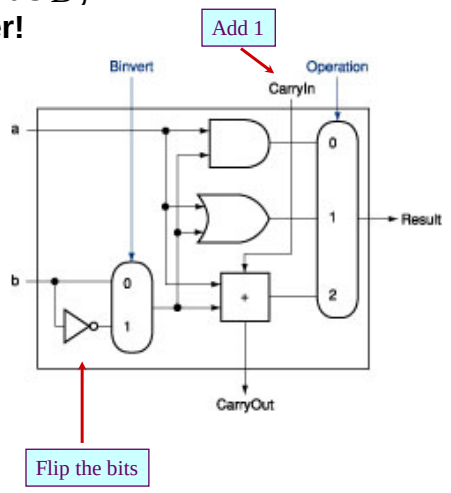
Extending to full 32-bit ALU

- Just need to hook up in series (in chain) so the carries will propagate.
- This is an example of hierarchical design (using building blocks) - what the book calls "abstraction"



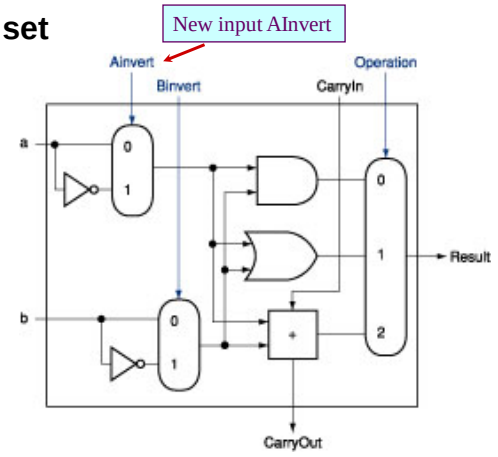
Including subtraction

- Subtraction is just A plus $\overline{B}$, so just need an inverter!
- Implement AND, OR, ADD logic
- Use 2-input mux to choose ADD/SUB (Binvert)
- Use 3-input mux to select output function (Operation)



Including NOR

- A NOR operation can be written as $\overline{A} \bullet \overline{B}$ Therefore, we can use the AND gate. Just need another inverter!
- To implement NOR, set
  - AInvert and
  - BInvert and
  - Operation = AND

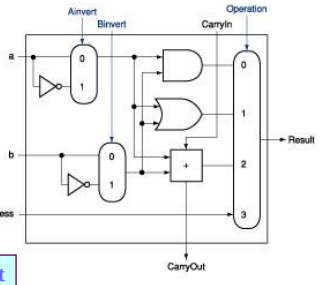


Including Set on Less Than (SLT)

- SLT sets output = 1 if A < B, 0 else
- Bitwise formula for slt:
  Result = 0, unless A-B < 0 *and* this is LSB
- So:
  - Set Binvert (to perform A-B)
  - Add a new input called Less
    - Connect to logical 0, for all bits except LSB
    - Connect LSB Less input to sign bit (MSB) from A-B
  - Question: Where do we get that sign bit? (We will be setting the result output of the MSB to logic 0)
  - Answer: Need slightly modified ALU for MSB; add an output called Set connected to MSB adder

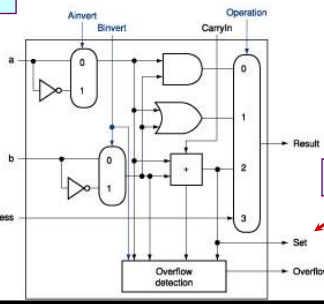
Final Standard and MSB Units

Standard 1-bit units
Chain 31 of them (for bits Index 0..30)



LSB: Connect to Set
Else: Hardwire to 0

MSB unit
Use for last bit (bit Index 31)



MSB sign bit

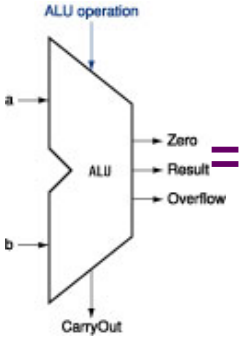
(Note we've also added overflow detection to the MSB.)

Final Version with Zero-check flag

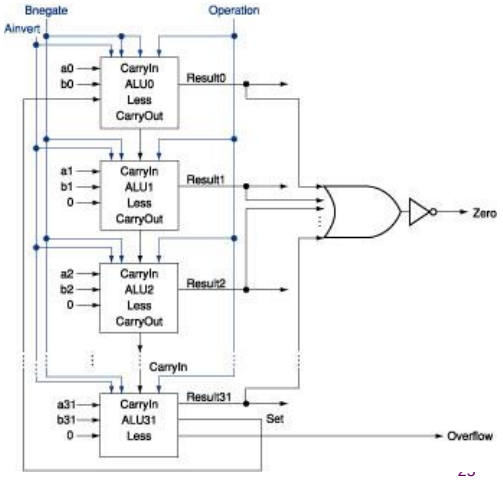
Control lines: "ALU operation", 4 bits:

- Ainvert
- Bnegate
- Operation1
- Operation0

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



=



25

Problem for You

- Analyze the performance of this design
  - Does it implement the required tasks? YES
  - How long does it take to do it?
    - Assume: The logic in a 1-bit adder takes 2 levels of logic to produce the carry output, and each gate has a 1 ns delay in this particular design.
    - What's the total amount of delay from IN to OUT for a 32-bit ALU to do an ADD?
- This design is called a ripple carry adder. Not the best
- What is the minimum delay possible?

26

Outline

- Basics
- ALU
- Carry look ahead
- Shifting

Carry look-ahead

So, what do we know without any delay?

- The values of the a_i and b_i inputs

What can we do with them?

- Figure out when a Carry Out will be *generated* (regardless of the Carry In value).

  $g_i = a_i\, b_i$

- Figure out when a Carry In will be *propagated* to the Carry Out. (when c_{out} will equal c_{in})

  $p_i = a_i + b_i$

- At each bit, we get $c_{out} = g + p\, c_{in}$

4-bit case, expanded

$$c_1 = g_0 + p_0c_0$$

$$\begin{aligned} c_2 &= g_1 + p_1c_1 \\ &= g_1 + p_1(g_0 + p_0c_0) \\ &= g_1 + p_1g_0 + p_1p_0c_0 \end{aligned}$$

$$\begin{aligned} c_3 &= g_2 + p_2c_2 \\ &= g_2 + p_2(g_1 + p_1g_0 + p_1p_0c_0) \\ &= g_2 + p_2g_1 + p_2p_1g_0 + p_2p_1p_0c_0 \end{aligned}$$

$$\begin{aligned} c_4 &= g_3 + p_3c_3 \\ &= g_3 + p_3(g_2 + p_2g_1 + p_2p_1g_0 + p_2p_1p_0c_0) \\ &= g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0 + p_3p_2p_1p_0c_0 \end{aligned}$$

29

- Can we build a 16-bit adder this way?
  - How much delay would it have?
  - How many gate inputs for the MSB Carry Out?
- How about a compromise?

  Hierarchical design approach:
  - Put together 4 4-bit carry-lookahead adders
  - Create 4-bit signals: propagates (P0-3), generates (G0-3), carries (C1- 4). Example formulas:

$$P_0 = p_3p_2p_1p_0$$

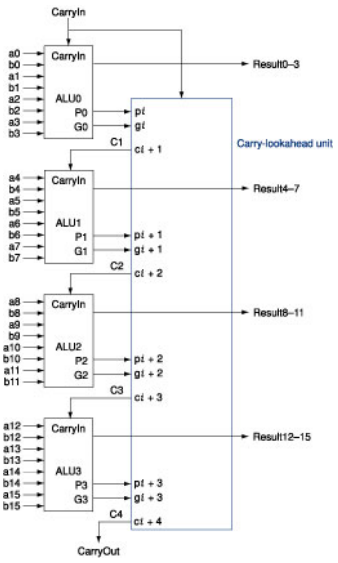
$$G_0 = g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0$$

$$C_1 = G_0 + P_0c_0$$

30

16-bit hierarchical carry-lookahead

- How much total delay is there in this design?
  - 1st level p & g in (a & b in →p & g out)
  - 1st level P & G out? (p & g in→P & G out)
  - 2nd level Carry Out? (P & G in→C out)
  - 1st level carry in? (C in + p & g in → c in)
  - Result out? (a & b & c in → Result out)
- What is the performance improvement?



Outline

- Basics
- ALU
- Carry look ahead
- Shifting

Shifting

Shifting we need to implement (6 kinds)

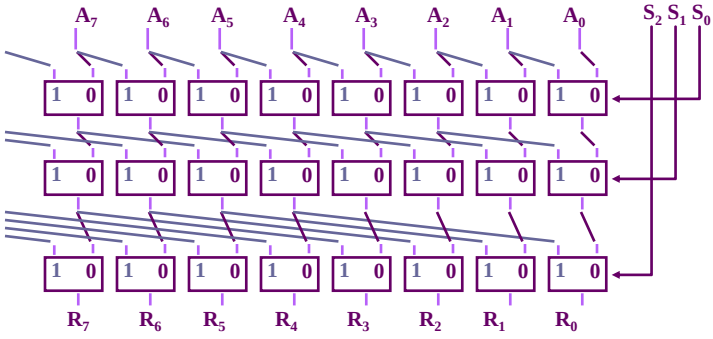
- **Direction: Left or right**
- **Extension:**
  - **Logical (Zero extended)**
  - **Arithmetic (Sign extended)**
  - **Rotate (extended by rotation of bits)**

❖ **Goal – shift in 1 clock cycle. Common types:**

- **Barrel Shifter (Full mux into flip-flop for each bit)**
- **Combinational Shifter (Hierarchical, MUX-based)**
- **Funnel Shifter (Selects a region of a doubled register)**

Combinational Shifter from MUXes

❖ **8-bit right shifter**



Issues/questions

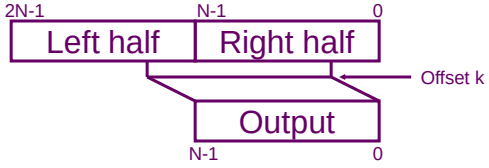
❖ **What comes in the MSBs?**

❖ **How many levels for 32-bit shifter?**

❖ **How can we do rotates (circular shifting)?**

Funnel Shifter

- **Advantage: Can do all 6 kinds of shifts with same hardware (can still use combinational mux design).**



Type	Left half	Right half	Offset
Logical Left	$R_{N-1} – R_0$	0…0	N-k
Logical Right	0…0	$R_{N-1} – R_0$	k
Arithmetic Left	$R_{N-1} – R_0$	0…0	N-k
Arithmetic Right	$R_{N-1}…R_{N-1}$	$R_{N-1} – R_0$	k
Rotate Left	$R_{N-1} – R_0$	$R_{N-1} – R_0$	N-k
Rotate Right	$R_{N-1} – R_0$	$R_{N-1} – R_0$	k