

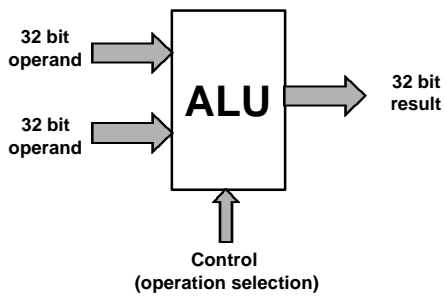
Constructing an ALU

Ref: Chapter 4

Arithmetic Logic Unit

- The device that performs the arithmetic operations and logic operations.
 - arithmetic ops: addition, subtraction
 - logic operations: AND, OR
- For MIPS we need a 32 bit ALU
 - can add 32 bit numbers, etc.

The Goal



Starting Small

- We can start by designing a 1 bit ALU.
- Put a bunch of them together to make larger ALUs.
 - building a larger unit from a 1 bit unit is simple for some operations, can be tricky for others.
- Bottom-Up approach:
 - build small units of functionality and put them together to build larger units.

CompOrg Fall 2001 - Building an ALU

4

1 bit AND/OR machine

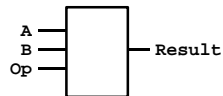
- We want to design a single box that can compute either AND or OR.
- We will use a *control input* to determine which operation is performed.
 - Name the control “Op”.
 - if $Op=0$ do an AND
 - if $Op=1$ do an OR

CompOrg Fall 2001 - Building an ALU

5

Truth Table For 1-bit AND/OR

Op	A	B	Result
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1



Op=0: Result is $A \cdot B$

Op=1: Result is $A + B$

CompOrg Fall 2001 - Building an ALU

6

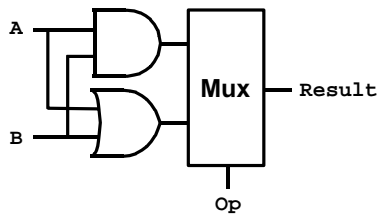
Logic for 1-Bit AND/OR

- We could derive SOP or POS and build the corresponding logic.
- We could also just do this:
 - Feed both **A** and **B** to an OR gate.
 - Feed **A** and **B** to an AND gate.
 - Use a 2-input MUX to pick which one will be used.
 - Op is the selection input to the MUX.

CompOrg Fall 2001 - Building an ALU

7

Logic Design for 1-Bit AND/OR



CompOrg Fall 2001 - Building an ALU

8

Addition

- We need to build a 1 bit *adder*
 - compute binary addition of 2 bits.
- We already know that the result is 2 bits.

A	B	O ₀	O ₁
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

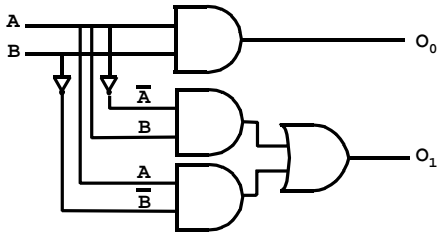
This is addition,
not logical OR!

$$\begin{array}{r} A \\ + B \\ \hline O_0 \ O_1 \end{array}$$

CompOrg Fall 2001 - Building an ALU

9

One Implementation



CompOrg Fall 2001 - Building an ALU

10

Binary addition and our *adder*

$$\begin{array}{r}
 1 1 \leftarrow \text{Carry} \\
 01001 \\
 + 01101 \\
 \hline
 10110
 \end{array}$$

What we really want is something that can be used to implement the binary addition algorithm.

- O_0 is the *carry*
- O_1 is the *sum*

CompOrg Fall 2001 - Building an ALU

11

What about the second column?

$$\begin{array}{r}
 1 1 \leftarrow \text{Carry} \\
 01001 \\
 + 01101 \\
 \hline
 10110
 \end{array}$$

- We are adding 3 bits
 - new bit is the *carry* from the first column.
 - The output is still 2 bits, a *sum* and a *carry*

CompOrg Fall 2001 - Building an ALU

12

Revised Truth Table for Addition

A	B	Carry In	Carry Out	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

CompOrg Fall 2001 - Building an ALU

13

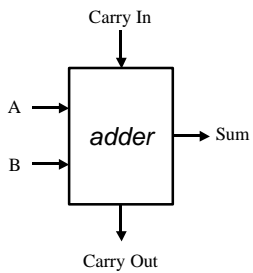
Logic Design for new adder

- We can derive SOP expressions from the truth table.
- We can build a combinational circuit that implements the SOP expressions.
- We can put it in a box and give it a name.

CompOrg Fall 2001 - Building an ALU

14

New Component: Adder



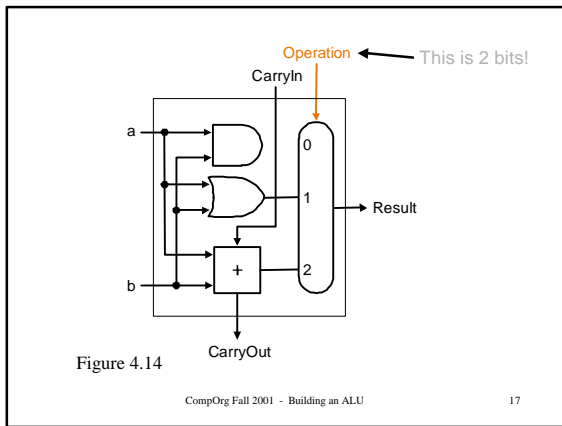
CompOrg Fall 2001 - Building an ALU

15

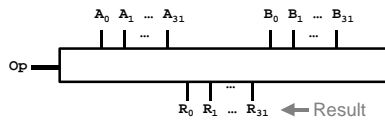
1 Bit ALU

- Combine the AND/OR with the adder.
- We must now use a 4-input MUX with 2 selection inputs.

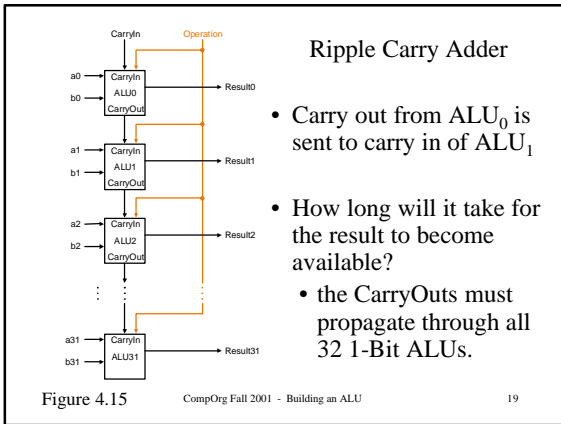
AND OR add



Building a 32 bit ALU



- 64 inputs
- 3 different Operations (AND,OR,add).
- 32 bit output



New Operation: Subtraction

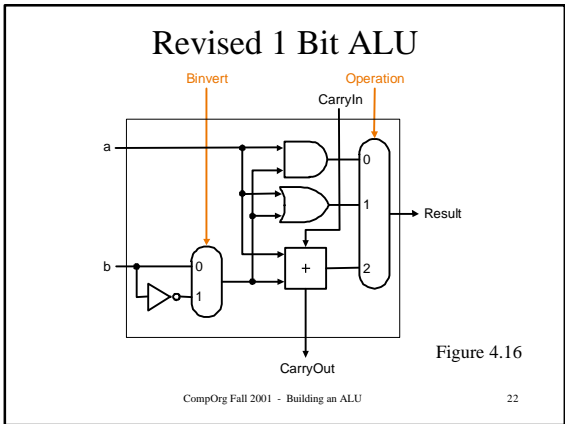
- Subtraction can be done with an adder:
 $A - B$ can be computed as $A + -B$
- To negate B we need to:
 - invert the bits.
 - add 1

CompOrg Fall 2001 - Building an ALU 20

Negating B in the ALU

- We can negate B by in the ALU by:
 - providing \overline{B} to the adder.
 - need a selection bit to do this.
 - To add 1, just set the initial carry in to 1!

CompOrg Fall 2001 - Building an ALU 21



Uses for our ALU

- addition, subtraction, OR and AND instructions can be implemented with our ALU.
 - we still need to get the right values to the ALU and set control lines.
- We can also support the **slt** instruction.
 - need to add a little more to the 1 bit ALU.

CompOrg Fall 2001 - Building an ALU 23

Supporting **slt**

slt needs to compare 2 numbers.

- comparison requires a subtraction.

if **A-B** is negative, then **A<B** is true.
 otherwise **A<B** is false.

This is what slt stores in the dest. register

True: output should be **0000000...001**
 False: output should be **0000000...000**

CompOrg Fall 2001 - Building an ALU 24

Strategies for speeding things up.

- We could derive the truth table for each of the 32 result bits as a function of 64 inputs.
- We know we can build SOP expressions for each and implement using 2 levels of gates.
- This might be a good test question!
 - don't worry, you would need so much paper I couldn't carry the tests to class...

A more realistic approach

- The problem is the *ripple*
 - The last carry-in is takes a long time to compute.
- We can try to compute the carry-in bits as fast as possible
 - this is called *carry lookahead*
 - It turns out we can easily compute the carry-in bits much faster (but not in constant time).

Carry In Analysis

- CarryIn_i is an input to the ith 1 bit adder.
- CarryOut_{i-1} is connected to CarryIn_i
- We know about how to compute the CarryOuts

A	B	Carry In	Carry Out	Sum
0	0	0	0	0
0	0	1	0	0
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Computing Carry Bits

- CarryIn₀ is an input to the adder.
 - we don't compute this – it's an input.
- CarryIn₁ depends on A₀, B₀ and CarryIn₀:

$$\text{CarryIn}_1 = (B_0 \cdot \text{CarryIn}_0) + (A_0 \cdot \text{CarryIn}_0) + (A_0 \cdot B_0)$$

↑
SOP: Requires 2 levels of gates

CarryIn₂

$$\text{CarryIn}_2 = (B_1 \cdot \text{CarryIn}_1) + (A_1 \cdot \text{CarryIn}_1) + (A_1 \cdot B_1)$$

We can substitute for CarryIn₁ and get this mess:

$$\begin{aligned} \text{CarryIn}_2 = & (B_1 \cdot B_0 \cdot \text{CarryIn}_0) + (B_1 \cdot A_0 \cdot \\ & \text{CarryIn}_0) + (B_1 \cdot A_0 \cdot B_0) + (A_1 \cdot B_0 \cdot \text{CarryIn}_0) + \\ & (A_1 \cdot A_0 \cdot \text{CarryIn}_0) + (A_1 \cdot A_0 \cdot B_0) + (A_1 \cdot B_1) \end{aligned}$$

The size of these expressions will get too big (that's the whole problem!).

Another way to describe CarryIn

$$\begin{aligned} C_{i+1} &= (B_i \cdot C_i) + (A_i \cdot C_i) + (A_i \cdot B_i) \\ &= (A_i \cdot B_i) + (A_i + B_i) \cdot C_i \end{aligned}$$

A_i • B_i : Call this *Generate* (G_i)

A_i + B_i : Call this *Propagate* (P_i)

$$C_{i+1} = G_i + P_i \cdot C_i$$

Generate and Propagate

$$C_{i+1} = G_i + P_i \cdot C_i$$
$$G_i = A_i \cdot B_i$$
$$P_i = A_i + B_i$$

- When A_i and B_i are both 1, G_i becomes a 1.
– a CarryOut is *generated*.
- If P_i is a 1, any Carry in is *propagated* to Carry Out.

Using G_i and P_i

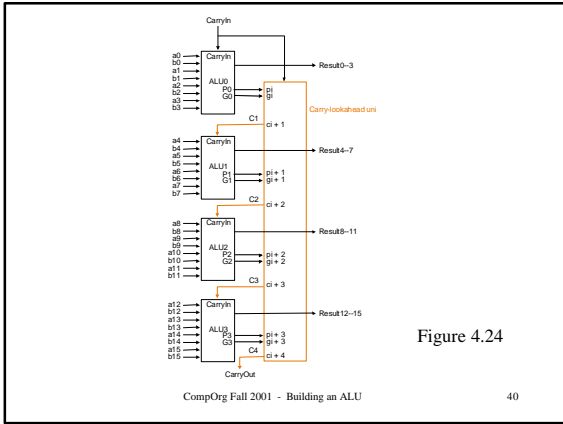
$$C_1 = G_0 + P_0 \cdot C_0$$

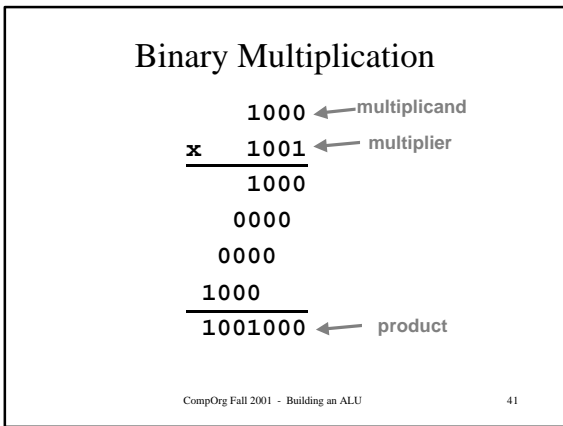
$$C_2 = G_1 + P_1 \cdot C_1$$
$$= G_1 + P_1 \cdot (G_0 + P_0 \cdot C_0)$$
$$= G_1 + P_1 \cdot G_0 + P_1 \cdot P_0 \cdot C_0$$

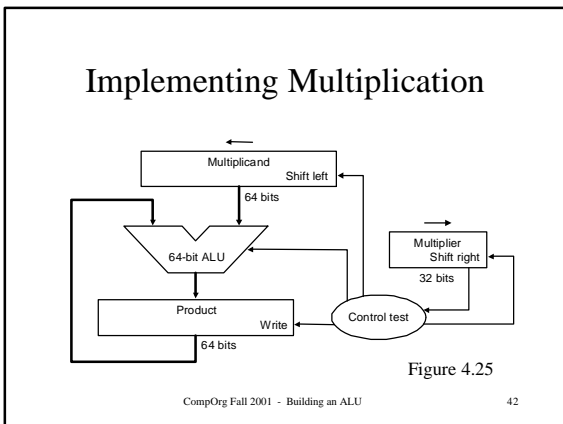
$$C_3 = G_2 + P_2 \cdot G_1 + P_2 \cdot P_1 \cdot G_0 + P_2 \cdot P_1 \cdot P_0 \cdot C_0$$

Implementation

- Expression is still too big to handle (for 32 bits).
- We *can* minimize the time needed to compute all the CarryIn bits for a 4 bit adder.
- Connect a bunch of 4 bit adders together and treat CarryIns to these adders in the same manner.







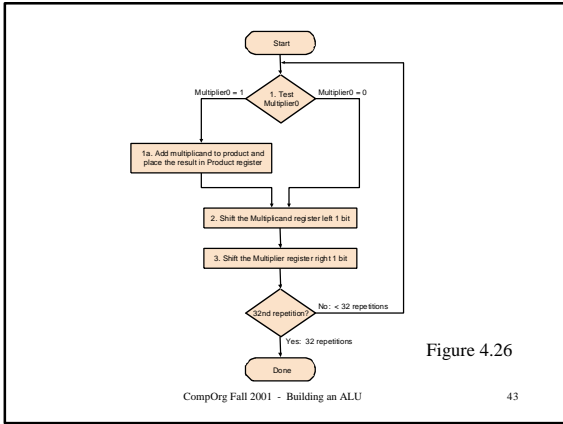


Figure 4.26
