

Arithmetic / Logic Unit - ALU Design

Dr. Arjan Durrresi
Louisiana State University
Baton Rouge, LA 70810
Durrresi@Csc.LSU.Edu

These slides are available at:

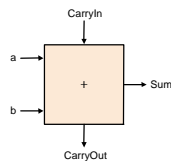
http://www.csc.lsu.edu/~durrresi/CSC3501_07/



- ❑ 1-Bit ALU
- ❑ Full Adder
- ❑ 32-Bit ALU

Different Implementations

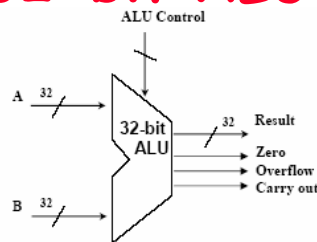
- Not easy to decide the "best" way to build something
 - Don't want too many inputs to a single gate
 - Don't want to have to go through too many gates
 - for our purposes, ease of comprehension is important
- Let's look at a 1-bit ALU for addition:



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

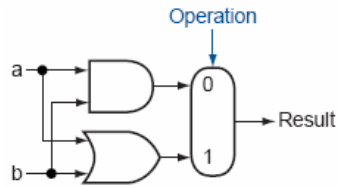
- How could we build a 1-bit ALU for add, and, and or?
- How could we build a 32-bit ALU?

32-bit ALU



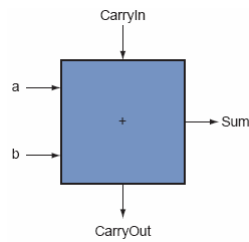
- Our ALU should be able to perform functions:
 - logical **and** function
 - logical **or** function
 - arithmetic **add** function
 - arithmetic **subtract** function
 - arithmetic **slt** (**set-less-then**) function
 - logical **nor** function
- ALU control lines define a function to be performed on A and B.

A 1-Bit ALU



- The 1-bit logical unit for AND and OR

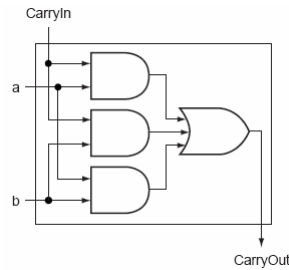
A Full Adder



Inputs			Outputs		Comments
a	b	CarryIn	CarryOut	Sum	
0	0	0	0	0	$0 + 0 + 0 = 00_{two}$
0	0	1	0	1	$0 + 0 + 1 = 01_{two}$
0	1	0	0	1	$0 + 1 + 0 = 01_{two}$
0	1	1	1	0	$0 + 1 + 1 = 10_{two}$
1	0	0	0	1	$1 + 0 + 0 = 01_{two}$
1	0	1	1	0	$1 + 0 + 1 = 10_{two}$
1	1	0	1	0	$1 + 1 + 0 = 10_{two}$
1	1	1	1	1	$1 + 1 + 1 = 11_{two}$

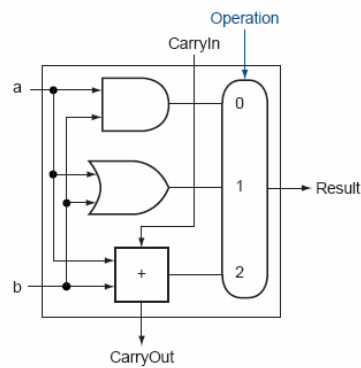
A Full Adder

Inputs		
a	b	CarryIn
0	1	1
1	0	1
1	1	0
1	1	1



- $CarryOut = (b \cdot CarryIn) + (a \cdot CarryIn) + (a \cdot b) + (a \cdot b \cdot CarryIn)$
- $CarryOut = (b \cdot CarryIn) + (a \cdot CarryIn) + (a \cdot b)$

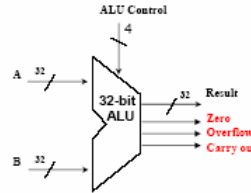
A Full Adder



- $Sum = (\overline{a} \cdot \overline{b} \cdot \overline{CarryIn}) + (\overline{a} \cdot b \cdot \overline{CarryIn}) + (a \cdot \overline{b} \cdot \overline{CarryIn}) + (a \cdot b \cdot \overline{CarryIn}) + (\overline{a} \cdot \overline{b} \cdot CarryIn) + (\overline{a} \cdot b \cdot CarryIn) + (a \cdot \overline{b} \cdot CarryIn) + (a \cdot b \cdot CarryIn)$

Functioning of 32-bit ALU

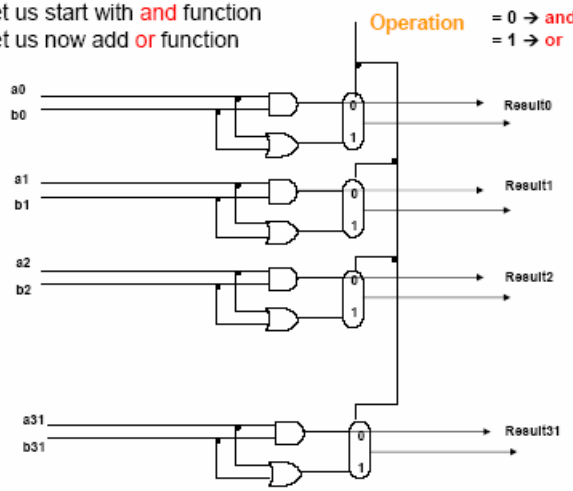
Function	ALU Control lines		
	Ainvert	Binvert	Operation
and	0	0	00
or	0	0	01
add	0	0	10
subtract	0	1	10
sll	0	1	11
nor	1	1	00



- Result lines provide result of the chosen function applied to values of A and B. Since this ALU operates on 32-bit operands, it is called 32-bit ALU.
- Zero output indicates if all Result lines have value 0.
- Overflow indicates a sign integer overflow of add and subtract functions; for unsigned integers, this overflow indicator does not provide any useful information.
- Carry out indicates carry out and unsigned integer overflow.

Designing 32-bit ALU: Beginning

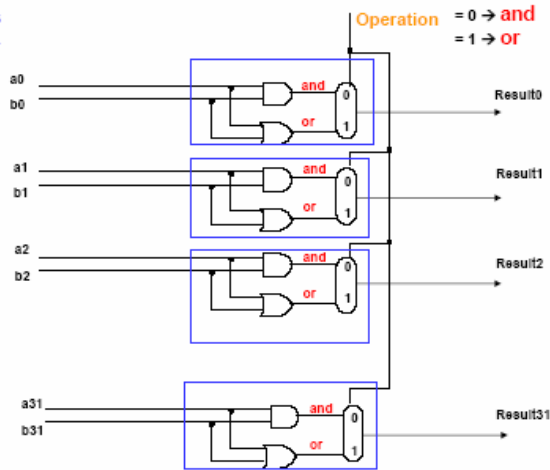
- Let us start with **and** function
- Let us now add **or** function



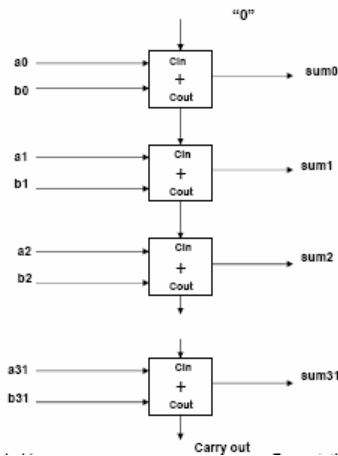
Designing 32-bit ALU: Principles

- Number of functions are performed internally, but only one result is chosen for the output of ALU

- 32-bit ALU is built out of 32 identical 1-bit ALU's

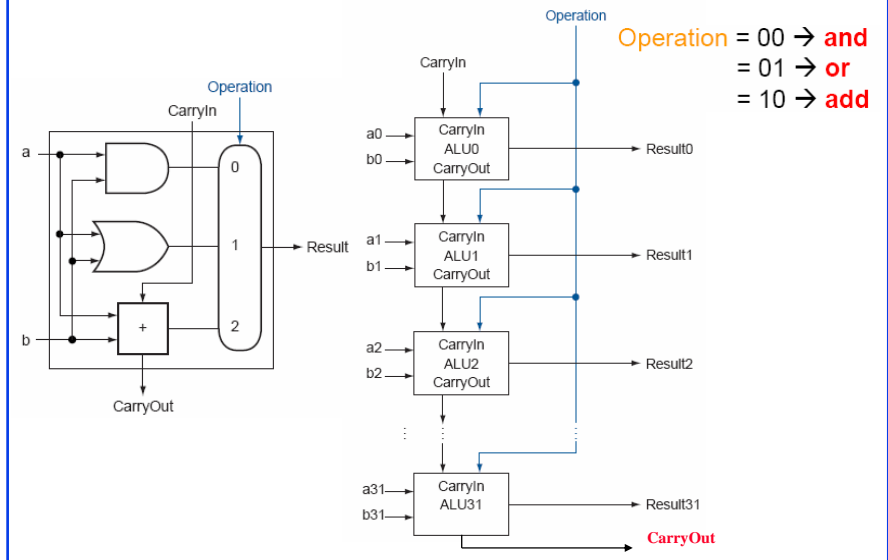


32-bit Adder

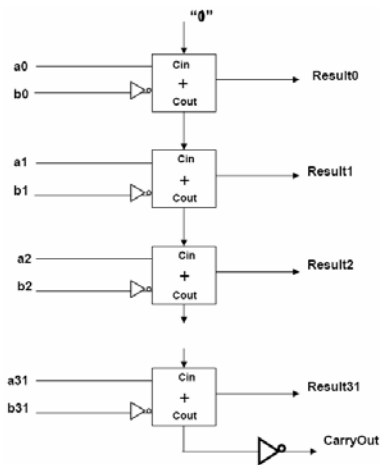


- ❑ This is a ripple carry adder.
- ❑ The key to speeding up addition is determining carry out in the higher order bits sooner.
- ❑ Result: **Carry look-ahead adder.**

32-bit ALU With 3 Functions

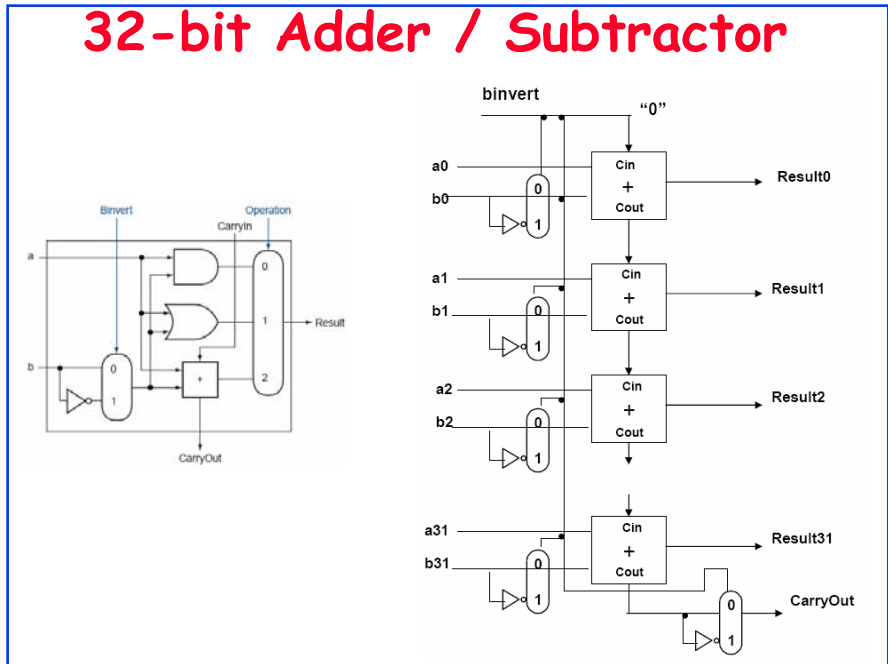


32-bit Subtractor

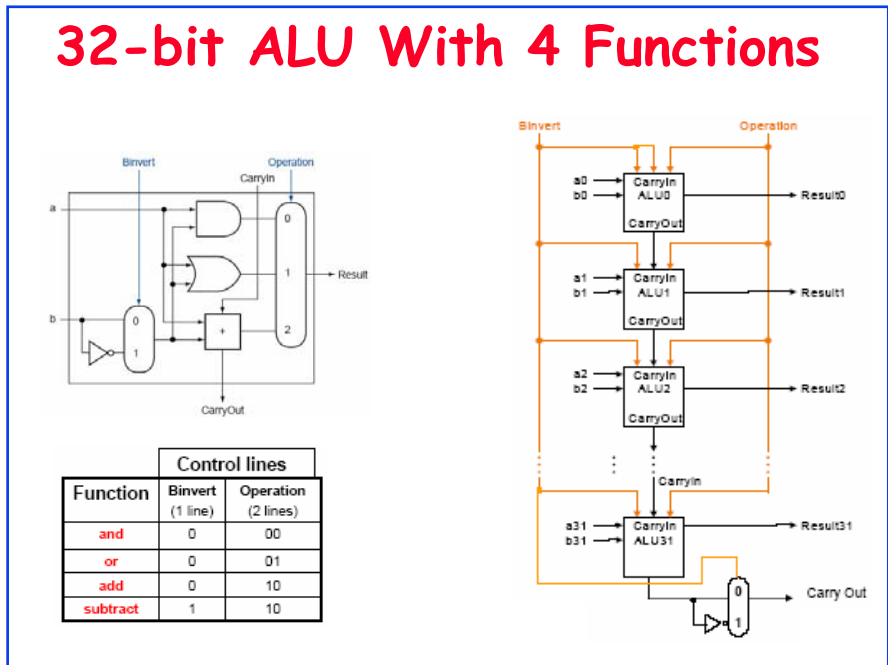


$$a + \bar{b} + 1 = a + (\bar{b} + 1) = a + (-b) = a - b$$

32-bit Adder / Subtractor



32-bit ALU With 4 Functions

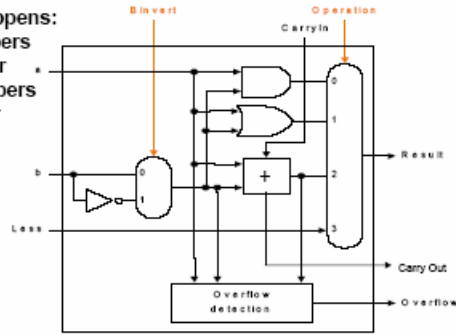


2's Complement Overflow

1-bit ALU for the most significant bit

2's complement overflow happens:

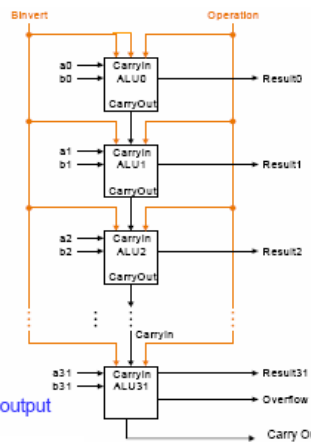
- if sum of two positive numbers results in a negative number
- if sum of two negative numbers results in a positive number



Other 1-bit ALUs, i.e. non-most significant bit ALUs, are not affected.

32-bit ALU With 4 Functions and Overflow

Function	Control lines	
	Binvert (1 line)	Operation (2 lines)
and	0	00
or	0	01
add	0	10
subtract	1	10



Missing: slt & nor functions and Zero output

Add correction for CarryOut

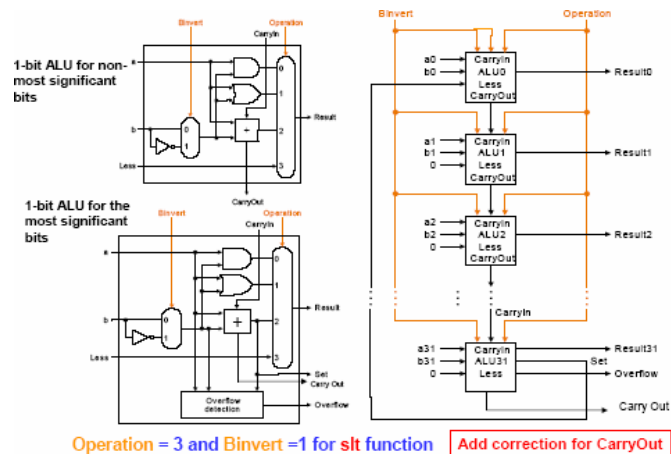
Set Less Than (slt) Function

- **slt** function is defined as:

$$A \text{ slt } B = \begin{cases} 000 \dots 001 & \text{if } A < B, \text{ i.e. if } A - B < 0 \\ 000 \dots 000 & \text{if } A \geq B, \text{ i.e. if } A - B \geq 0 \end{cases}$$

- Thus each 1-bit ALU should have an additional input (called "Less"), that will provide results for **slt** function. This input has value 0 for all but 1-bit ALU for the least significant bit.
- For the least significant bit **Less** value should be sign of $A - B$

32-bit ALU With 5 Functions

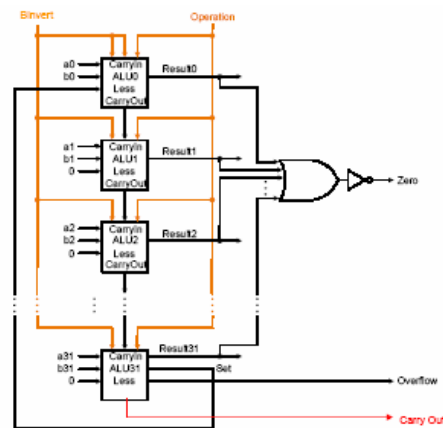


Zero

- $A - B = 0 \Rightarrow A = B$
- To test fore Zero:
- $Zero = (\text{Result}_{31} + \text{Result}_{30} + \dots + \text{Result}_2 + \text{Result}_1 + \text{Result}_0)$

32-bit ALU with 5 Functions and Zero

Function	Control lines	
	Binvert (1 line)	Operation (2 lines)
and	0	00
or	0	01
add	0	10
subtract	1	10
sll	1	11



Faster Addition: Carry Lookahead

- The key to speeding up addition is determining the carry in to the high-order bits sooner.
 - Fast Carry Using "Infinite" Hardware
 - $\text{CarryIn}_2 = (b_1 \cdot \text{CarryIn}_1) + (a_1 \cdot \text{CarryIn}_1) + (a_1 \cdot b_1)$
 - $\text{CarryIn}_1 = (b_0 \cdot \text{CarryIn}_0) + (a_0 \cdot \text{CarryIn}_0) + (a_0 \cdot b_0)$
- And
- $C_2 = (a_1 \cdot a_0 \cdot b_0) + (a_1 \cdot a_0 \cdot c_0) + (a_1 \cdot b_0 \cdot c_0) + (b_1 \cdot a_0 \cdot b_0) + (b_1 \cdot a_0 \cdot c_0) + (b_1 \cdot b_0 \cdot c_0) + (a_1 \cdot b_1)$

Carry-lookahead Adder

- $c_{i+1} = (b_i \cdot c_i) + (a_i \cdot c_i) + (a_i \cdot b_i) = (a_i \cdot b_i) + (a_i + b_i) \cdot c_i$
- Generate (g_i) and propagate (p_i)
 - $g_i = a_i \cdot b_i$
 - $p_i = a_i + b_i$
- $c_{i+1} = g_i + p_i \cdot c_i$
- If $g_i = 1$. That is, the adder generates a CarryOut (c_{i+1}) independent of the value of CarryIn (c_i)
 - $c_{i+1} = g_i + p_i \cdot c_i = 1$
- If $g_i = 0$ and $p_i = 1$
 - $c_{i+1} = g_i + p_i \cdot c_i = 0 + 1 \cdot c_i = c_i$
- The adder propagates CarryIn to a CarryOut.
- CarryIn_{i+1} is a 1 if either g_i is 1 or both p_i is 1 and CarryIn_i is 1.

Carry-lookahead Adder

$$c1 = g0 + (p0 \cdot c0)$$

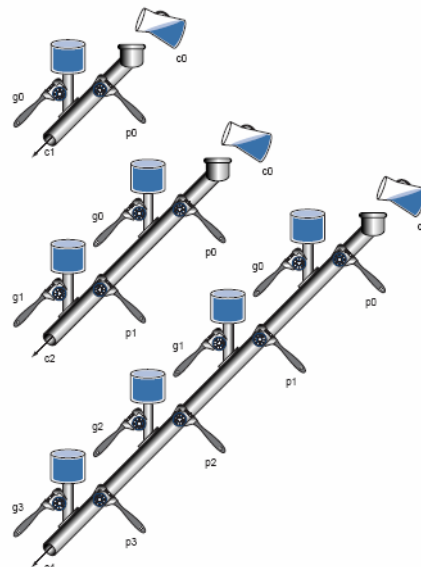
$$c2 = g1 + (p1 \cdot g0) + (p1 \cdot p0 \cdot c0)$$

$$c3 = g2 + (p2 \cdot g1) + (p2 \cdot p1 \cdot g0) + (p2 \cdot p1 \cdot p0 \cdot c0)$$

$$c4 = g3 + (p3 \cdot g2) + (p3 \cdot p2 \cdot g1) + (p3 \cdot p2 \cdot p1 \cdot g0) + (p3 \cdot p2 \cdot p1 \cdot p0 \cdot c0)$$

Carry-lookahead Adder

- A plumbing analogy for carry lookahead for 1 bit, 2 bits, and 4 bits using water pipes and valves.



Fast Carry Using the Second Level of Abstraction

- To go faster, we'll need carry lookahead at a higher level.
- For the four 4-bit adder blocks:

$$P_0 = p_3 \cdot p_2 \cdot p_1 \cdot p_0$$

$$P_1 = p_7 \cdot p_6 \cdot p_5 \cdot p_4$$

$$P_2 = p_{11} \cdot p_{10} \cdot p_9 \cdot p_8$$

$$P_3 = p_{15} \cdot p_{14} \cdot p_{13} \cdot p_{12}$$

- That is, the "super" propagate signal for the 4-bit abstraction (P_i) is true only if each of the bits in the group will propagate a carry.

Fast Carry Using the Second Level of Abstraction

- For the "super" generate signal (G_i), we care only if there is a carry out of the most significant bit of the 4-bit group.
- This obviously occurs if generate is true for that most significant bit; it also occurs if an earlier generate is true *and* all the intermediate propagates, including that of the most significant bit, are also true:

$$G_0 = g_3 + (p_3 \cdot g_2) + (p_3 \cdot p_2 \cdot g_1) + (p_3 \cdot p_2 \cdot p_1 \cdot g_0)$$

$$G_1 = g_7 + (p_7 \cdot g_6) + (p_7 \cdot p_6 \cdot g_5) + (p_7 \cdot p_6 \cdot p_5 \cdot g_4)$$

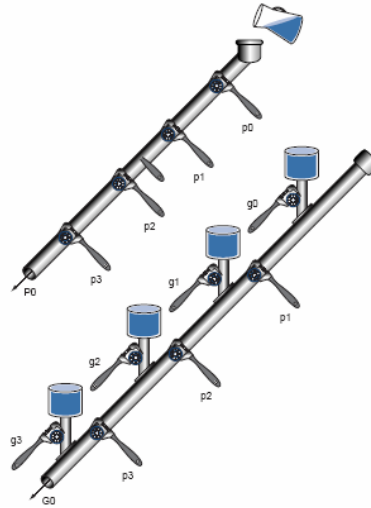
$$G_2 = g_{11} + (p_{11} \cdot g_{10}) + (p_{11} \cdot p_{10} \cdot g_9) + (p_{11} \cdot p_{10} \cdot p_9 \cdot g_8)$$

$$G_3 = g_{15} + (p_{15} \cdot g_{14}) + (p_{15} \cdot p_{14} \cdot g_{13}) + (p_{15} \cdot p_{14} \cdot p_{13} \cdot g_{12})$$

Fast Carry Using the Second Level of Abstraction

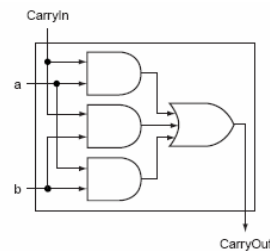
$$\begin{aligned}
 C1 &= G0 + (P0 \cdot c0) \\
 C2 &= G1 + (P1 \cdot G0) + (P1 \cdot P0 \cdot c0) \\
 C3 &= G2 + (P2 \cdot G1) + (P2 \cdot P1 \cdot G0) + (P2 \cdot P1 \cdot P0 \cdot c0) \\
 C4 &= G3 + (P3 \cdot G2) + (P3 \cdot P2 \cdot G1) + (P3 \cdot P2 \cdot P1 \cdot G0) \\
 &\quad + (P3 \cdot P2 \cdot P1 \cdot P0 \cdot c0)
 \end{aligned}$$

- A plumbing analogy for the next-level carry-lookahead signals $P0$ and $G0$.
- $P0$ is open only if all four propagates (p_i) are open, while water flows in $G0$ only if at least one generate (g_i) is open and all the propagates downstream from that generate are open.



Example: Speed of Ripple Carry versus Carry Lookahead

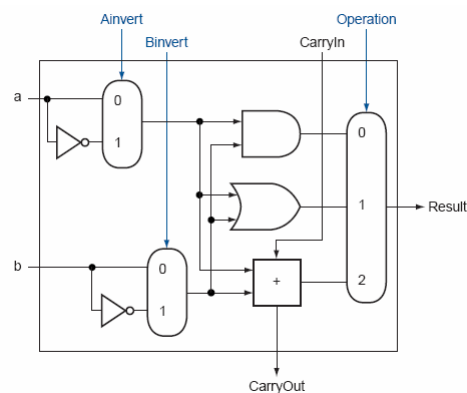
- Time is estimated by simply counting the number of gates along the path through a piece of logic. Compare the number of *gate delays* for paths of two 16-bit adders, one using ripple carry and one using two-level carry lookahead.
- the carry out signal takes two gate delays per bit. Then the number of gate delays between a carry in to the least significant bit and the carry out of the most significant is $16 \times 2 = 32$.



Example

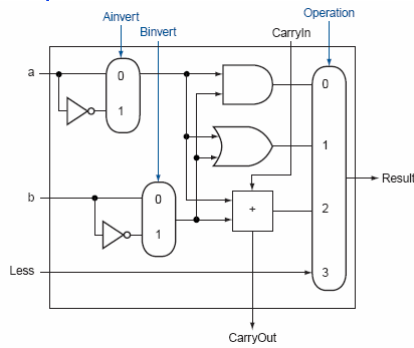
- For carry lookahead, the carry out of the most significant bit is just C_4 , defined in the example. It takes two levels of logic to specify C_4 in terms of P_i and G_i (the OR of several AND terms). P_i is specified in one level of logic (AND) using p_i , and G_i is specified in two levels using p_i and g_i , so the worst case for this next level of abstraction is two levels of logic. p_i and g_i are each one level of logic, defined in terms of a_i and b_i . If we assume one gate delay for each level of logic in these equations, the worst case is $2 + 2 + 1 = 5$ gate delays.

NOR

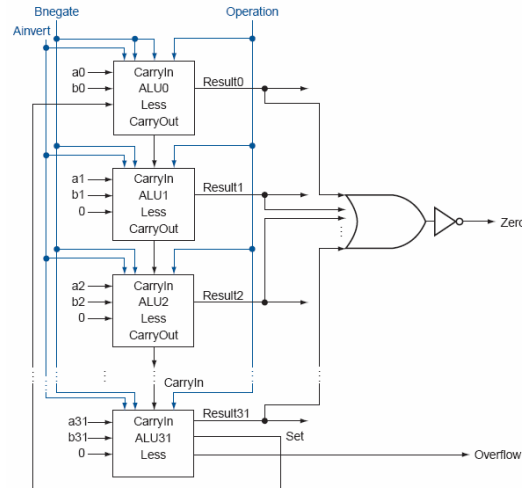


$$\overline{(a + b)} = \bar{a} \cdot \bar{b}$$

32-bit ALU with 6 Functions



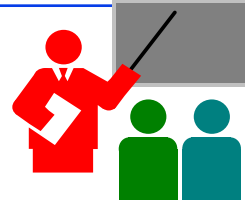
Function	Ainvert	Binvert	Operation
and	0	0	00
or	0	0	01
add	0	0	10
subtract	0	1	10
sll	0	1	11
nor	1	1	00



32-bit ALU Elaboration

- We have now accounted for all but one of the arithmetic and logic functions for the core MIPS instruction set. 32-bit ALU with 6 functions omits support for shift instructions.
- It would be possible to widen 1-bit ALU multiplexer to include 1-bit shift left and/or 1-bit shift right.
- Hardware designers created the circuit called a barrel shifter, which can shift from 1 to 31 bits in no more time than it takes to add two 32-bit numbers. Thus, shifting is normally done outside the ALU.

Summary



- ❑ 1-Bit ALU
- ❑ Full Adder
- ❑ 32-Bit ALU