

EEC 483 Computer Organization (Fall 2006)

Chapter 3. Arithmetic for Computers

Chansu Yu

Cleveland State University

Table of Contents

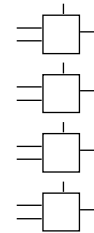
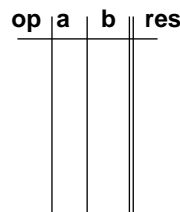
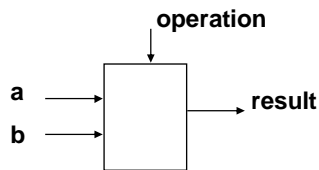
- Ch.1, 4 Introduction & Performance
 - Ch. 2 Instruction: Machine Language
 - 2.1 Introduction
 - 2.2 Operations (arithmetic, memory operations)
 - 2.3 Operands
 - 2.4 Representing instructions
 - 2.5 Logical operations
 - 2.6 Control flow operations
 - 2.7 Supporting procedures
 - 2.8 Beyond numbers
 - 2.9 MIPS addressing
 - 2.10 Starting a program
 - Ch. 3 CPU Implementation: Arithmetic
 - 3.1 Introduction
 - 3.2 Signed and unsigned numbers
 - 3.3 Addition and subtraction
 - 3.4 Multiplication
 - 3.5 Division
 - 3.6 Floating point
 - **Appendix B Constructing an Arithmetic Logic Unit (ALU)**
 - Ch. 5 CPU Implementation: All others
 - Ch. 6-9 Advanced topics
- Software interface* →
- ← *Hardware interface*
- Key parts of this course

An ALU (arithmetic logic unit)

- ❑ Let's build an ALU to support the `and/or/add/sub` instructions
 - we'll just build a 1 bit ALU, and use 32 of them
 - `and/or` is simpler because each bit operates independently
 - `add/sub` is complicated due to ??? (carry & borrow)
- ❑ 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.

An ALU (arithmetic logic unit)

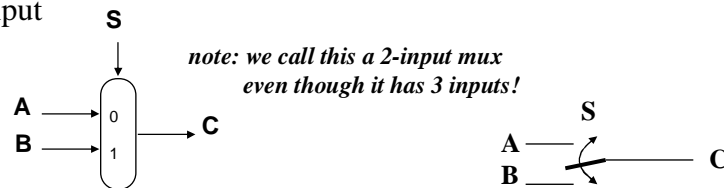
- ❑ 1-bit `andi` and `ori`



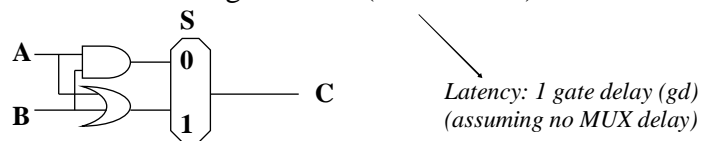
- ❑ 3-input implementation (a,b,op)

A different implementation with: The Multiplexor

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



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

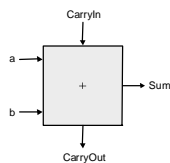


Cleveland State
University

5

c.yu91@csuohio.edu

1-bit ALU for Addition



- ❑ Boolean equation for carryout ??? $c_{out} = a b + a c_{in} + b c_{in}$
- ❑ Boolean equation for sum??? $sum = a \text{ xor } b \text{ xor } c_{in}$
- ❑ Implementation ???

Latency: 2 gds for Cout &
1 gd for sum

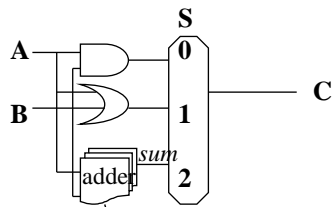
Cleveland State
University

6

c.yu91@csuohio.edu

1-bit ALU for And/Or/Addition

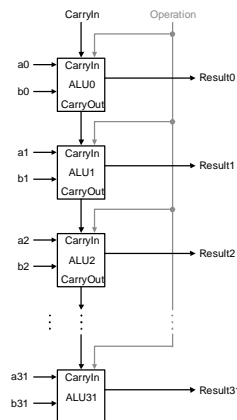
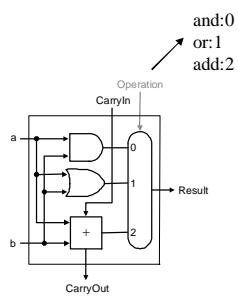
- How could we build a 1-bit ALU for add, and, and or?



carryout
 adder has two outputs
 What to do with the carryout??? => connect to the next bit ALU

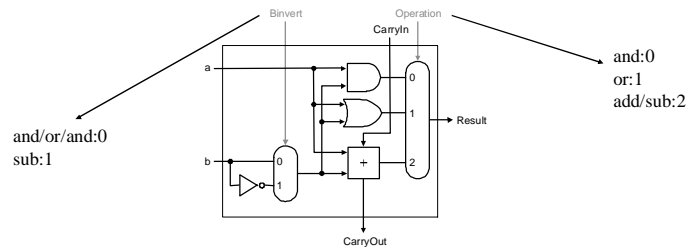
- How could we build a 32-bit ALU?

Building a 32 bit ALU



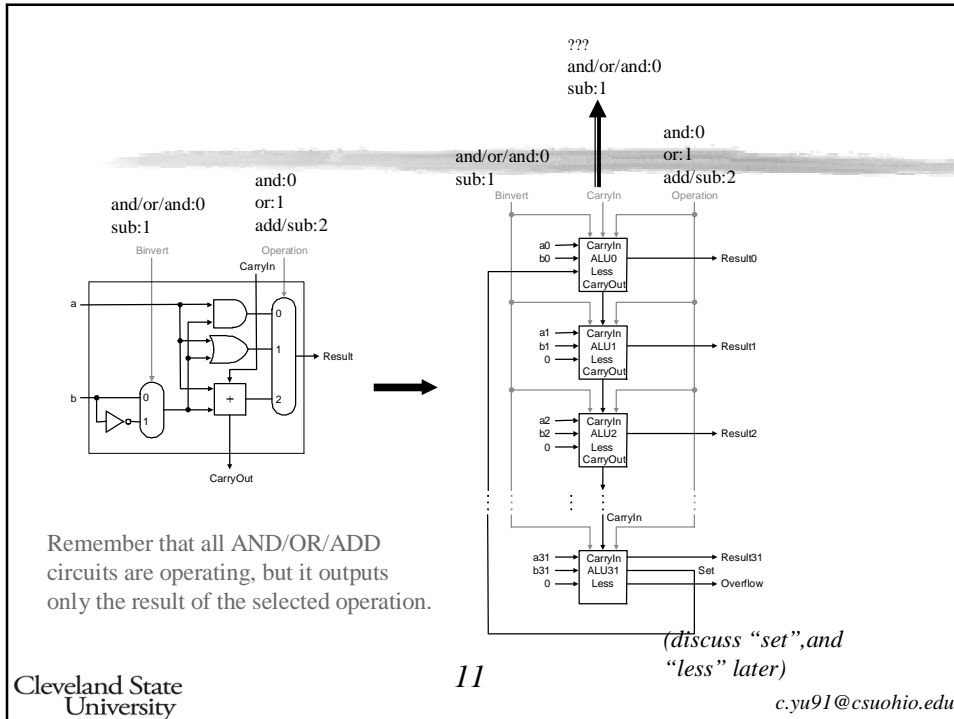
What about subtraction (a - b) ?

- ❑ Two's complement approach: just negate b and add.
- ❑ How do we negate (-b = b^{bar}+1)?
- ❑ A very clever solution:
 - b^{bar}: invert each bit
 - +1: carry in to each bit (NO!!!) => 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<b: answer = -(b-a), where (b-a)>0
 - $a-b = a+(-b) = a+(2^n-b) = 2^n-(b-a)$:
 - This is exactly the representation of -(b-a): OK
 - If a>b: answer = (a-b), where (a-b)>0
 - $a-b = a+(-b) = a+(2^n-b) = (a-b)+2^n$: This is not the representation of (a-b), but 2ⁿ is just ignored: OK
 - One's Complement
 - If a<b: answer = -(b-a), where (b-a)>0
 - $a-b = a+(-b) = a+(2^n-b)-1 = 2^n-(b-a)-1$:
 - This is exactly the representation of -(b-a): OK
 - If a>b: answer = (a-b), where (a-b)>0
 - $a-b = a+(-b) = a+(2^n-b)-1 = (a-b)+2^n-1$: This is not the representation of (a-b)???
 - 2ⁿ 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 !!!



What's more

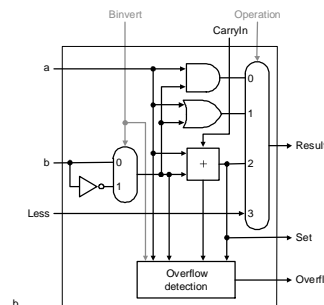
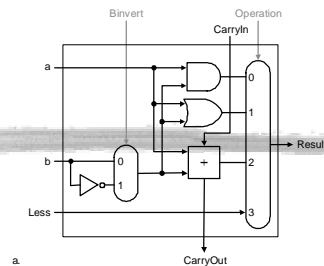
- ❑ MIPS instructions
 - add/ sub/ and/ or/ lw/ sw/ beq/ bne/ slt/ j/ jr/ ...
- ❑ We're covering ALU instructions in Ch.4
 - add/ sub/ and/ or
- ❑ What other instructions require ALU operations?
 - All inst: PC calculation: $PC = PC + 4$, $PC = PC + \text{jump offset}$
 - Lw/sw: Address calculation: $lw \$s0, 100(\$t1)$
 - SlT: $slt \$s0, \$s1, \$s2$: if $\$s1 < \$s2$, set $\$s0 = 1$
 \Rightarrow if $(\$s1 - \$s2) < 0$: "subtraction" + some other activities
 - Beq/bne: $beq \$s0, \$s1, label$: if $\$s0 = \$s1$, jump to label
 \Rightarrow 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)



and/or/and:0
sub:1
slt: 1

same

and/or/add:0
sub:1
slt: 1

and:0
or:1
add/sub:2
slt: 3

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
...
c32= 64gd

{ r31=63gd

Thus, r0' = 63gd

Cleveland State University 15 c.yu91@csuohio.edu

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-r31 must carry the result of subtraction (in slt, r0-r31 Do not carry the result of the subtraction)

Cleveland State University 16 c.yu91@csuohio.edu

Supporting MIPS instructions

- and/or/add/sub
- slt: subtract and output r0 based on “set” output from the 1-bit adder of the last-bit ALU. R1-r31 is “0” (“less” input)
- beq: subtract and output “zero” flag based on all “r” bits

Operation input

- 0: and
- 1: or
- 2: add/sub/beq
- 3: slt

Binvert input

- 0: and/or/add
- 1: sub/beq/slt

Cleveland State University 17 c.yu91@csuohio.edu

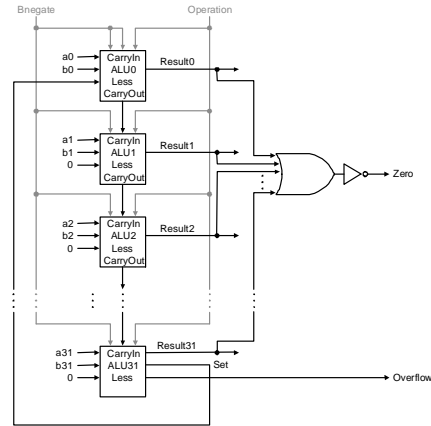
Supporting MIPS ISA

- ❑ 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
 - 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

Cleveland State University 18 c.yu91@csuohio.edu

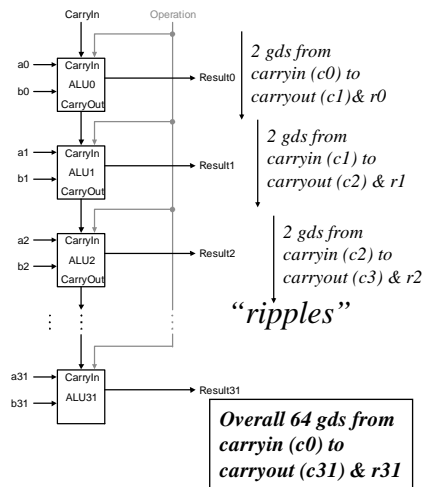
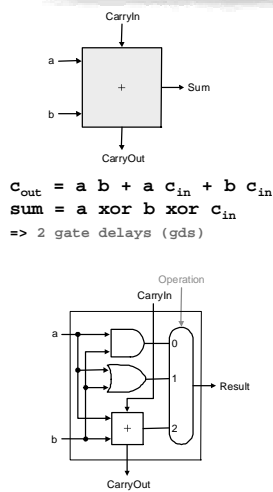
Problem: ripple carry adder is slow

- ❑ 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 gds 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



Problem: ripple carry adder is slow (RCA)

c1: 2-input "and" (3) + 3-input "or" (1)
c2: 3-input "and" (7) + 7-input "or" (1)
 ...
c32: 33-input "and" + (2³²-1)-input "or"

- Can you see the ripple? How could you get rid of the dependency?
 - Remove c_i 's by substitution !!!

$$\begin{aligned}
 c_1 &= b_0c_0 + a_0c_0 + a_0b_0 & c_2 &= f(a_i, b_i, c_0) = 3 \cdot 2 + 1 = 7 \text{ terms} \\
 c_2 &= b_1c_1 + a_1c_1 + a_1b_1 & c_3 &= f(a_i, b_i, c_0) = 7 \cdot 2 + 1 = 15 \text{ terms} \\
 c_3 &= b_2c_2 + a_2c_2 + a_2b_2 & c_4 &= f(a_i, b_i, c_0) = 15 \cdot 2 + 1 = 31 \text{ terms} \\
 c_4 &= b_3c_3 + a_3c_3 + a_3b_3 & & \\
 &\dots & & \\
 c_{31} &= b_{31}c_{31} + a_{31}c_{31} + a_{31}b_{31} & c_{31} &= f(a_i, b_i, c_0) = \text{"TOO MANY TERMS !!!"}
 \end{aligned}$$

Not feasible! Why?
 => Gate delays ??? (=2) : OK
 => But, many-input gate is required
 (e.g. 100-input and gate)

$$\begin{aligned}
 C_k &= C_{k-1} \cdot 2 + 1 \\
 C_{31} &= 2^{32} - 1 \\
 &= 4 \text{ billions}
 \end{aligned}$$

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" + (2³²-1)-input "or"

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

5-input "and" +
5-input "or"

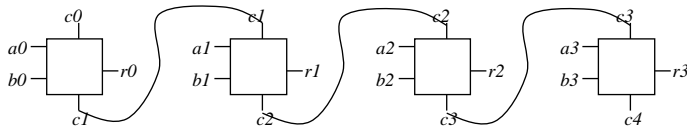
- Did we get rid of the ripple?

$$\begin{aligned}
 c_1 &= g_0 + p_0 c_0 & c_2 &= g_1 + p_1 (g_0 + p_0 c_0) = g_1 + p_1 g_0 + p_1 p_0 c_0 \\
 c_2 &= g_1 + p_1 c_1 & c_3 &= \dots \\
 c_3 &= g_2 + p_2 c_2 & c_4 &= g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 c_0 \text{ (5 terms)} \\
 c_4 &= g_3 + p_3 c_3 & &
 \end{aligned}$$

Feasible! Why?
22

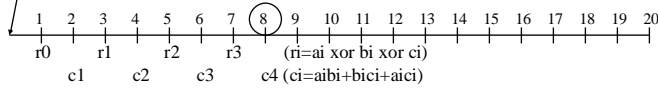
4-bit adder: RCA vs CLA

This is important because the next stage can start whenever c_4 is available.



ri's : 7gds
c4: 8gds

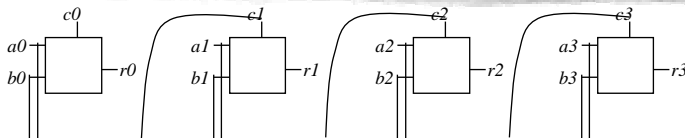
a0-a3, b0-b3, c0



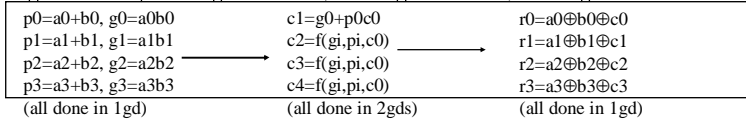
RCA

4-bit adder: RCA vs CLA

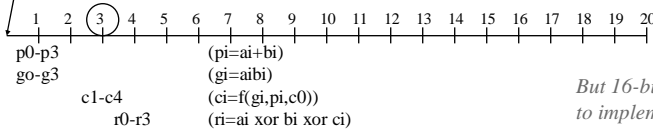
This is important because the next stage can start whenever c_4 is available.



ri's : 4gds
c4: 3gds



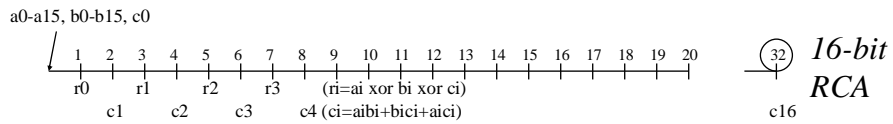
a0-a3, b0-b3, c0



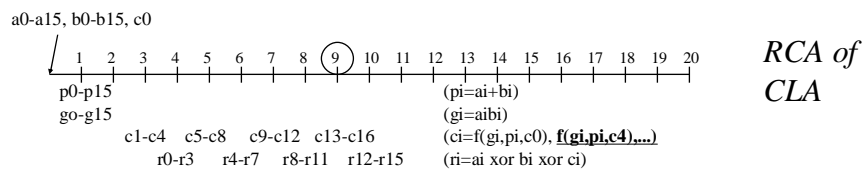
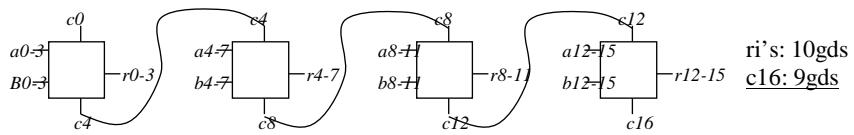
CLA

But 16-bit CLA is too BIG to implement !!!
c16: 17-input "and" +
17-input "or"

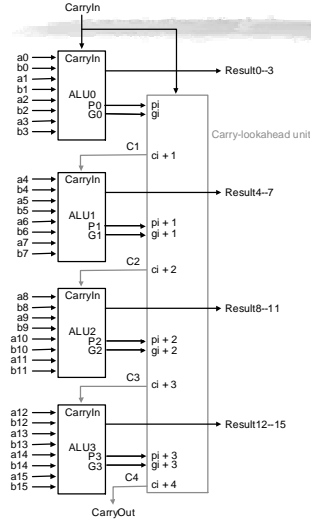
16-bit adder: 16-bit RCA vs (RCA of CLAs) vs (CLA of CLAs)



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

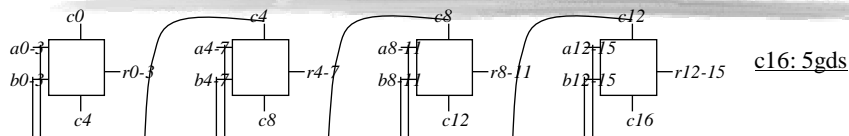


16-bit adder: CLA of CLAs



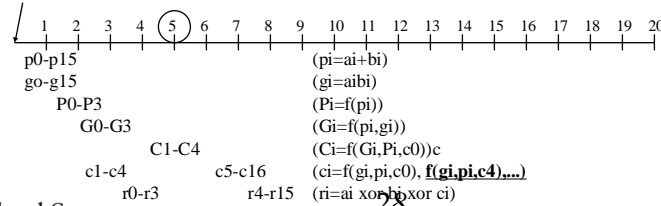
- Could use ripple carry of 4-bit CLA adders
- Better: use the CLA principle again!

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



$p_0 = a_0 + b_0, g_0 = a_0 b_0$ $p_1 = a_1 + b_1, g_1 = a_1 b_1$ \dots $p_{15} = a_{15} + b_{15}, g_{15} = a_{15} b_{15}$ (all done in 1gds)	$P_0 = p_3 p_2 p_1 p_0$ \dots $P_3 = p_{15} p_{14} p_{13} p_{12}$ $G_0 = g_3 + p_3 g_2 + \dots$ \dots $G_3 = g_{15} + p_{15} g_{14} + \dots$ (all done in 2gds)	$C_1 = c_4 = G_0 + P_0 c_0$ $C_2 = c_8 = f(G_i, P_i, c_0)$ $C_3 = c_{12} = f(G_i, P_i, c_0)$ $C_4 = c_{16} = f(G_i, P_i, c_0)$ (all done in 2gds)
-------------------------------------------------------------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------

a0-a15, b0-b15, c0



CLA of CLA

Carry Look-ahead Adder (CLA)

□ Basic quantities and equations

- $a_0 \sim a_{15}, b_0 \sim b_{15}, c_0$
- $c_{i+1} = a_i b_i + a_i c_i + b_i c_i$ and $r_i = a_i \oplus b_i \oplus c_i$

□ 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

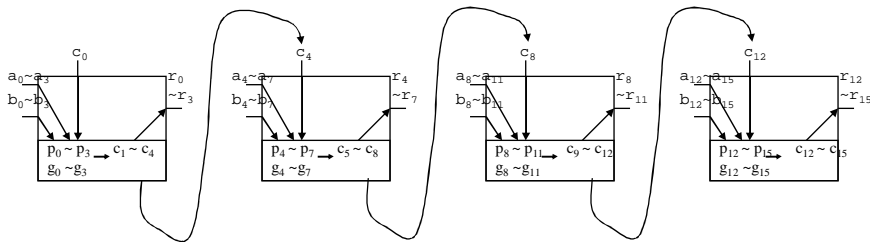
$c_1 = g_0 + p_0 c_0$	(2 terms)	}	<i>c₁~c₄ depends on p_i, g_i and c₀. So does r₀~r₃.</i>		
$c_2 = g_1 + p_1 c_1$	$c_2 = g_1 + p_1 (g_0 + p_0 c_0) = g_1 + p_1 g_0 + p_1 p_0 c_0$			(3 terms)	
$c_3 = g_2 + p_2 c_2$	$c_3 = g_2 + p_2 (g_1 + p_1 g_0 + p_1 p_0 c_0)$			$= g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0$	(4 terms)
$c_4 = g_3 + p_3 c_3$	$c_4 = g_3 + p_3 (g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0)$			$= g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 c_0$	(5 terms)

Similarly, $c_5 \sim c_8$ depends on p_i, g_i and c_4 . So does $r_4 \sim r_7$.

Similarly, $c_9 \sim c_{12}$ depends on p_i, g_i and c_8 . So does $r_8 \sim r_{11}$.

Similarly, $c_{13} \sim c_{16}$ depends on p_i, g_i and c_{12} . So does $r_{12} \sim r_{15}$.

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

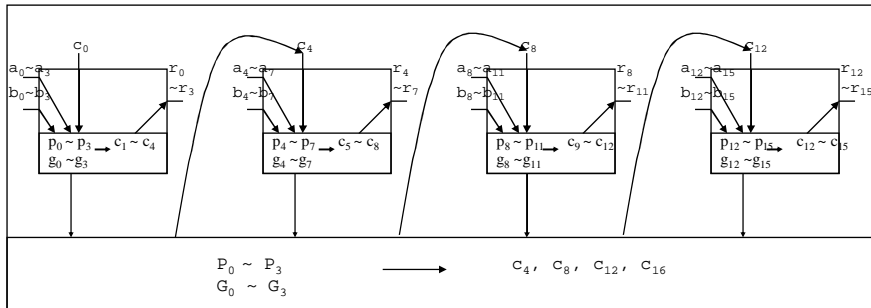
From above, $c_4 = g_3 + P_3g_2 + P_3P_2g_1 + P_3P_2P_1g_0 + P_3P_2P_1P_0c_0 = G_0 + P_0c_0$ (2 terms)

where, $G_0 = g_3 + P_3g_2 + P_3P_2g_1 + P_3P_2P_1g_0$, $P_0 = P_3P_2P_1P_0$

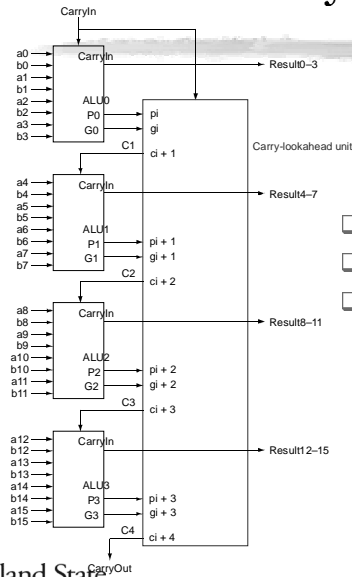
Similarly, $c_8 = G_1 + P_1c_4 = G_1 + P_1(G_0 + P_0c_0) = G_1 + P_1G_0 + P_1P_0c_0$ (3 terms)

$c_{12} = G_2 + P_2c_8 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0c_0$ (4 terms)

$c_{16} = G_3 + P_3c_{12} = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0c_0$ (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!

```
module MIPSALU (ALUctl, A, B, ALUOut, Zero);
  input [3:0] ALUctl;
  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 @(ALUctl, A, B) //reevaluate if these change
  case (ALUctl)
    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
```

FIGURE B.4.3 A Verilog behavioral definition of a MIPS ALU. This could be synthesized using a module library containing basic arithmetic and logical operations.