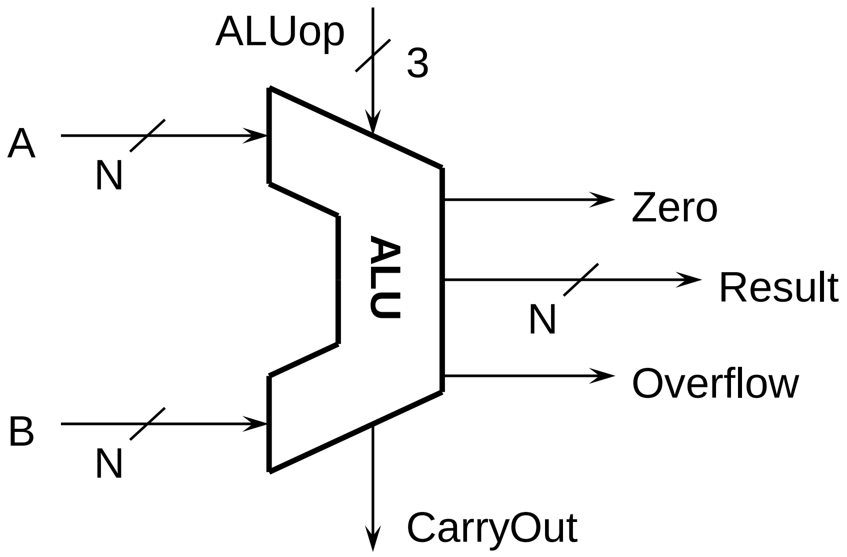
Computer Architecture
Ch. 3-2: ALU Design, Add/Sub

Spring, 2005

Sao-Jie Chen (csj@cc.ee.ntu.edu.tw)

Review: Functional Specification of the ALU



• ALU Control Lines (ALUop)	Function
000	And
001	Or
010	Add
110	Subtract
111	Set-on-less-than

MIPS arithmetic instructions

Instruction	***Example***	***Meaning***	***Comments***
add	add \$1,\$2,\$3	\$1 = \$2 + \$3	3 operands; exception possible
subtract	sub \$1,\$2,\$3	\$1 = \$2 – \$3	3 operands; exception possible
add immediate	addi \$1,\$2,80	\$1 = \$2 + 80	+ constant; exception possible
add unsigned	addu \$1,\$2,\$3	\$1 = \$2 + \$3	3 operands; no exceptions
subtract unsigned	subu \$1,\$2,\$3	\$1 = \$2 – \$3	3 operands; no exceptions
add imm. unsign.	addiu \$1,\$2,80	\$1 = \$2 + 80	+ constant; no exceptions
multiply	mult \$2,\$3	Hi, Lo = \$2 x \$3	64-bit signed product
multiply unsigned	multu\$2,\$3	Hi, Lo = \$2 x \$3	64-bit unsigned product
divide	div \$2,\$3	Lo = \$2 ÷ \$3,	Lo = quotient, Hi = remainder
		Hi = \$2 mod \$3	
divide unsigned	divu \$2,\$3	Lo = \$2 ÷ \$3,	Unsigned quotient & remainder
		Hi = \$2 mod \$3	
Move from Hi	mfhi \$1	\$1 = Hi	Used to get copy of Hi
Move from Lo	mflo \$1	\$1 = Lo	Used to get copy of Lo

MIPS logical instructions

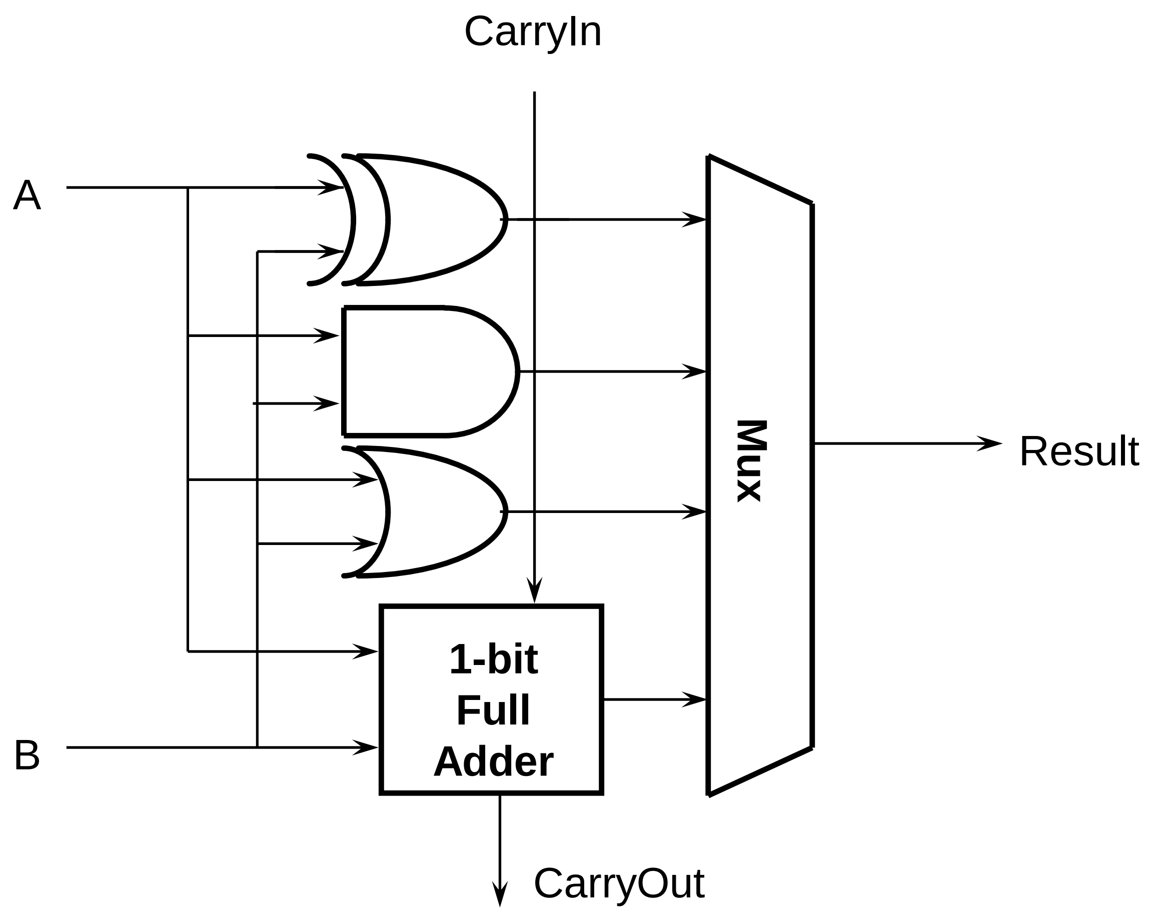
Instruction	*Example*	*Meaning*	*Comment*
and	and \$1,\$2,\$3	\$1 = \$2 & \$3	3 reg. operands; Logical AND
or	or \$1,\$2,\$3	\$1 = \$2 | \$3	3 reg. operands; Logical OR
xor	xor \$1,\$2,\$3	\$1 = \$2 ⊕ \$3	3 reg. operands; Logical XOR
nor	nor \$1,\$2,\$3	\$1 = ~(\$2 | \$3)	3 reg. operands; Logical NOR
and immediate	andi \$1,\$2,10	\$1 = \$2 & 10	Logical AND reg, constant
or immediate	ori \$1,\$2,10	\$1 = \$2 | 10	Logical OR reg, constant
xor immediate	xori \$1, \$2,10	\$1 = \$2 ⊕ 10	Logical XOR reg, constant
shift left logical	sll \$1,\$2,10	\$1 = \$2 << 10	Shift left by constant
shift right logical	srl \$1,\$2,10	\$1 = \$2 >> 10	Shift right by constant
shift right arithm.	sra \$1,\$2,10	\$1 = \$2 >> 10	Shift right (sign extend)
shift left logical	sllv \$1,\$2,\$3	\$1 = \$2 << \$3	Shift left by variable
shift right logical	srlv \$1,\$2, \$3	\$1 = \$2 >> \$3	Shift right by variable
shift right arithm.	srav \$1,\$2, \$3	\$1 = \$2 >> \$3	Shift right arith. by variable

Additional MIPS ALU requirements

- Xor, Nor, Xori
  => Logical XOR, logical NOR or use 2 steps: (A OR B) XOR 1111...1111
- Sll, Srl, Sra
  => Need left shift, right shift, right shift arithmetic by 0 to 31 bits
- Mult, MultU, Div, DivU
  => Need 32-bit multiply and divide, signed and unsigned

Add XOR to ALU

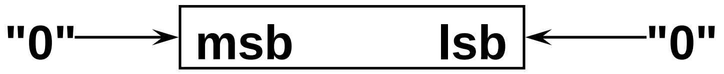
- **Expand Multiplexor**



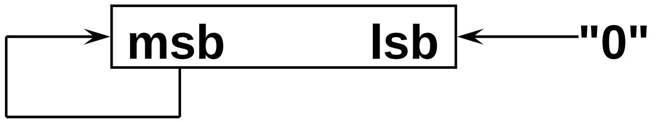
Shifters

- **Three different kinds:**

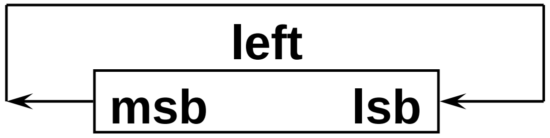
  ***logical*-- value shifted in is always "0"**

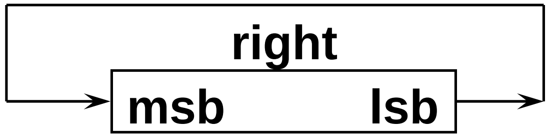


  ***arithmetic*-- on right shifts, sign extend**



  ***rotating*-- shifted out bits are wrapped around (not in MIPS)**

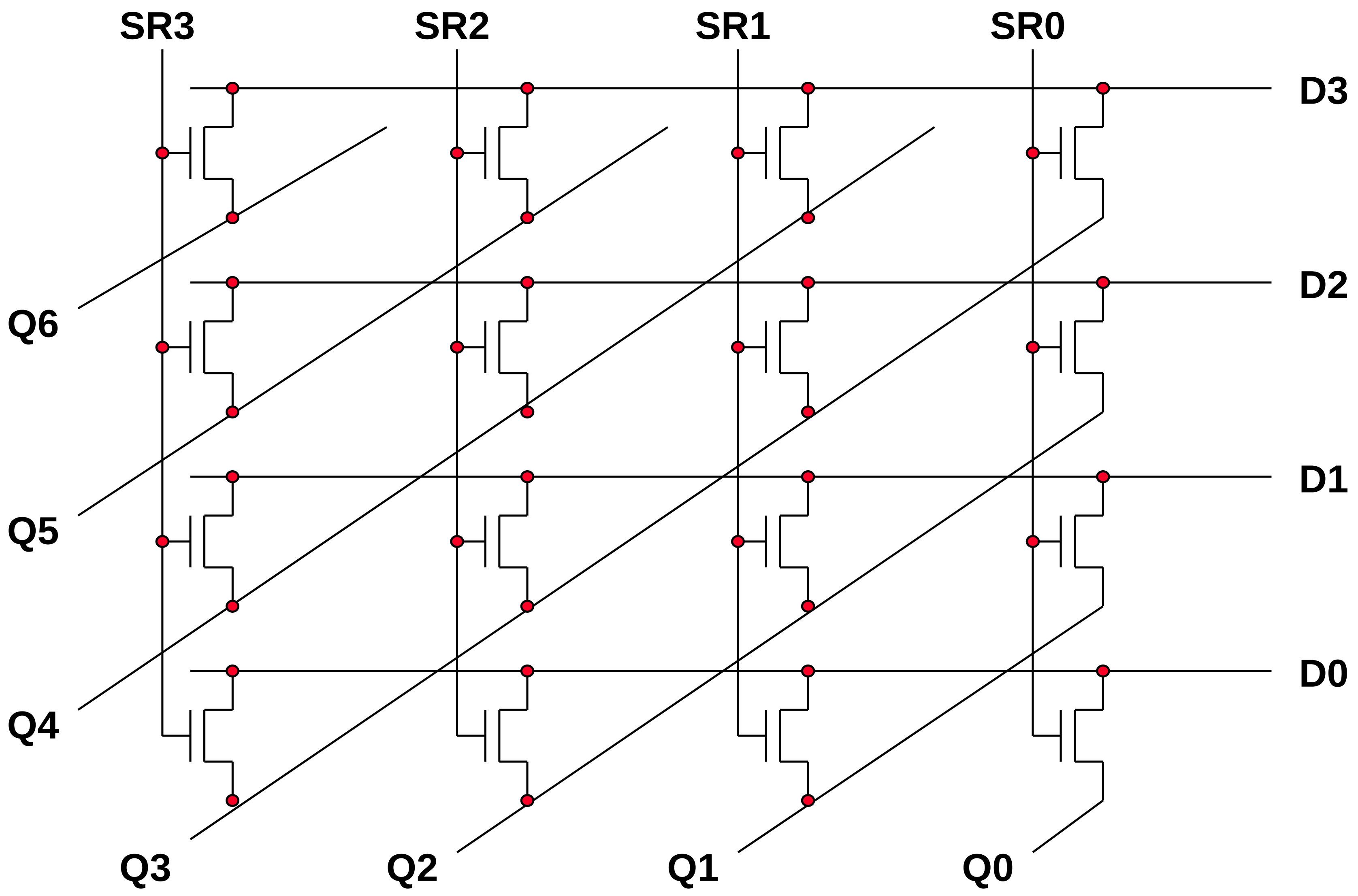




Note: these are single bit shifts. A given instruction might request 0 to 32 bits to be shifted!

Barrel Shifter

- Technology-dependent solutions:



Compare and Branch

- Compare and Branch
  - BEQ rs, rt, offset   if R[rs] == R[rt] then PC-relative branch
  - BNE rs, rt, offset   ≠
- Compare to zero and branch
  - BLEZ rs, offset   if R[rs] <= 0 then PC-relative branch
  - BGTZ rs, offset   >
  - BLT   <
  - BGEZ   ≥
  - BLTZAL rs, offset   if R[rs] < 0 then branch and link (into R 31)
  - BGEZAL   ≥

MIPS ALU requirements

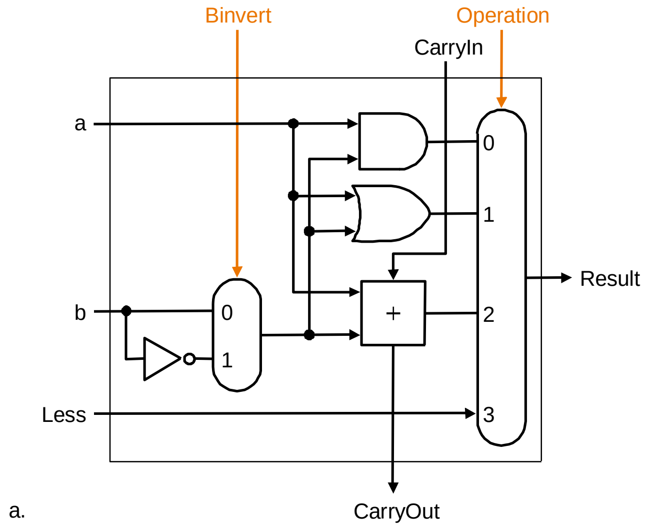
- Add, AddU, Sub,   SubU, AddI, AddIU
  => 2's complement adder with overflow detection & inverter
- SLTI, SLTIU (set less than)
  => 2's complement adder with inverter, check sign bit of result
- BEQ, BNE (branch on equal or not equal)
  => 2's complement adder with inverter, check if result = 0
- And, Or, Andi, Ori
  => Logical AND, logical OR
- ALU from last lecture supports these ops

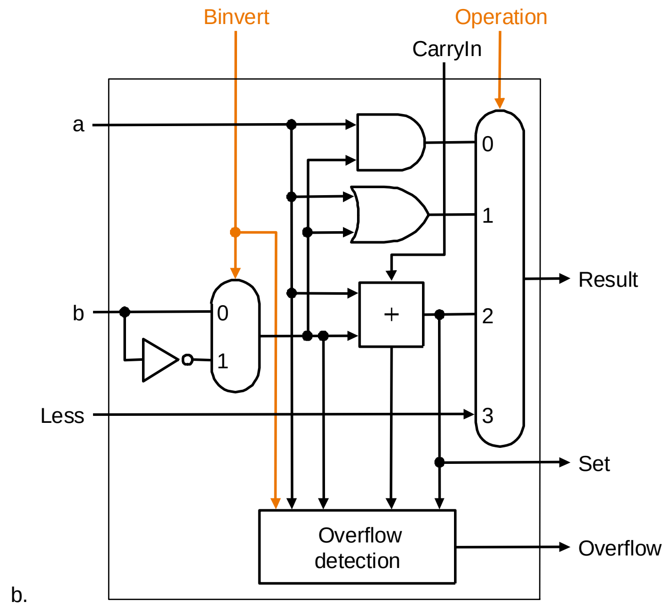
Tailoring the ALU to the MIPS

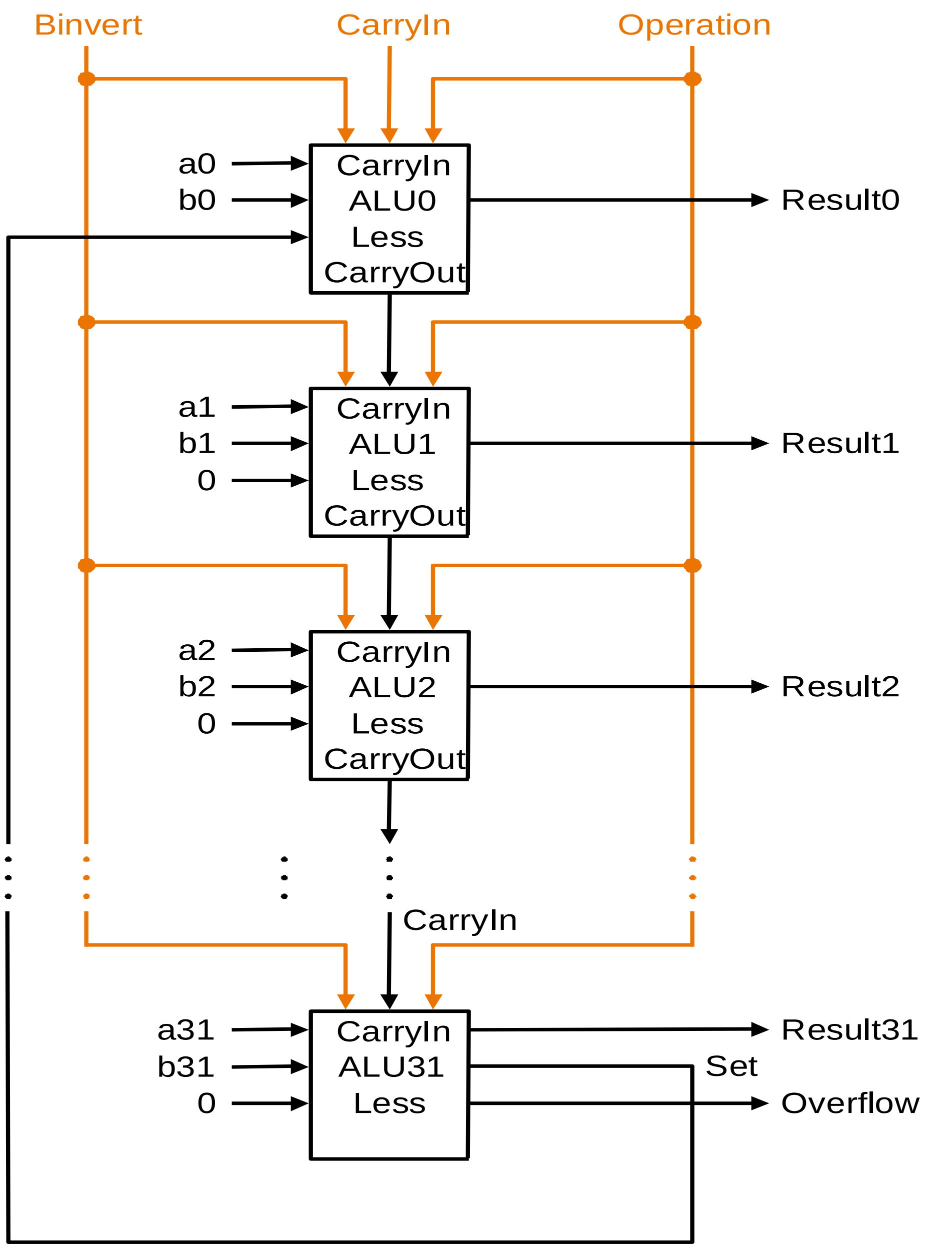
- Need to support the set-on-less-than instruction (slt)
  - remember: `slt` is an arithmetic instruction
  - produces a 1 if rs < rt and 0 otherwise
  - use subtraction: (a-b) < 0 implies a < b
- Need to support test for equality (beq $t5, $t6, $t7)
  - use subtraction: (a-b) = 0 implies a = b

Supporting slt

- **Can we figure out the idea?**






Binvert
CarryIn
Operation
a0
b0
CarryIn
ALU0
Less
CarryOut
Result0
a1
b1
0
CarryIn
ALU1
Less
CarryOut
Result1
a2
b2
0
CarryIn
ALU2
Less
CarryOut
Result2
CarryIn
a31
b31
0
CarryIn
ALU31
Less
Result31
Set
Overflow

Addition & Subtraction

- **Just like in grade school (carry/borrow 1s)**

```
     0111          0111          0110
  +  0110       -  0110       -  0101
  ---------------------------------------
```

- **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
```

note that overflow term is somewhat misleading, it does not mean a carry "overflowed"

Overflow Detection

Overflow: the result is too large (or too small) to represent properly

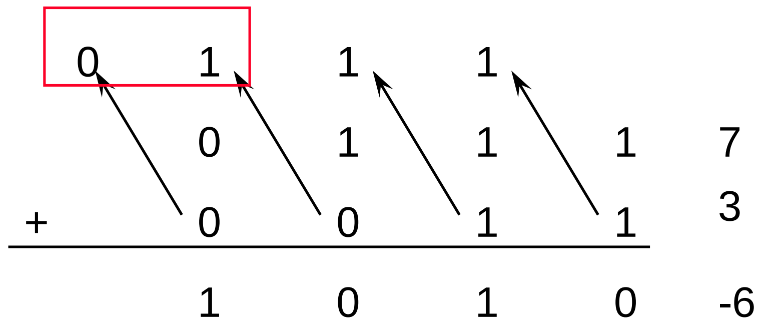
- Example: - 8 <= 4-bit binary number <= 7

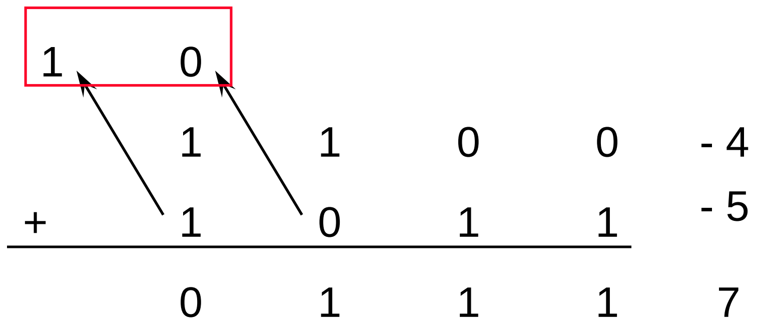
- When adding operands with different signs, overflow cannot occur!

- Overflow occurs when adding:
  - 2 positive numbers and the sum is negative
  - 2 negative numbers and the sum is positive

Homework exercise: Prove you can detect overflow by:

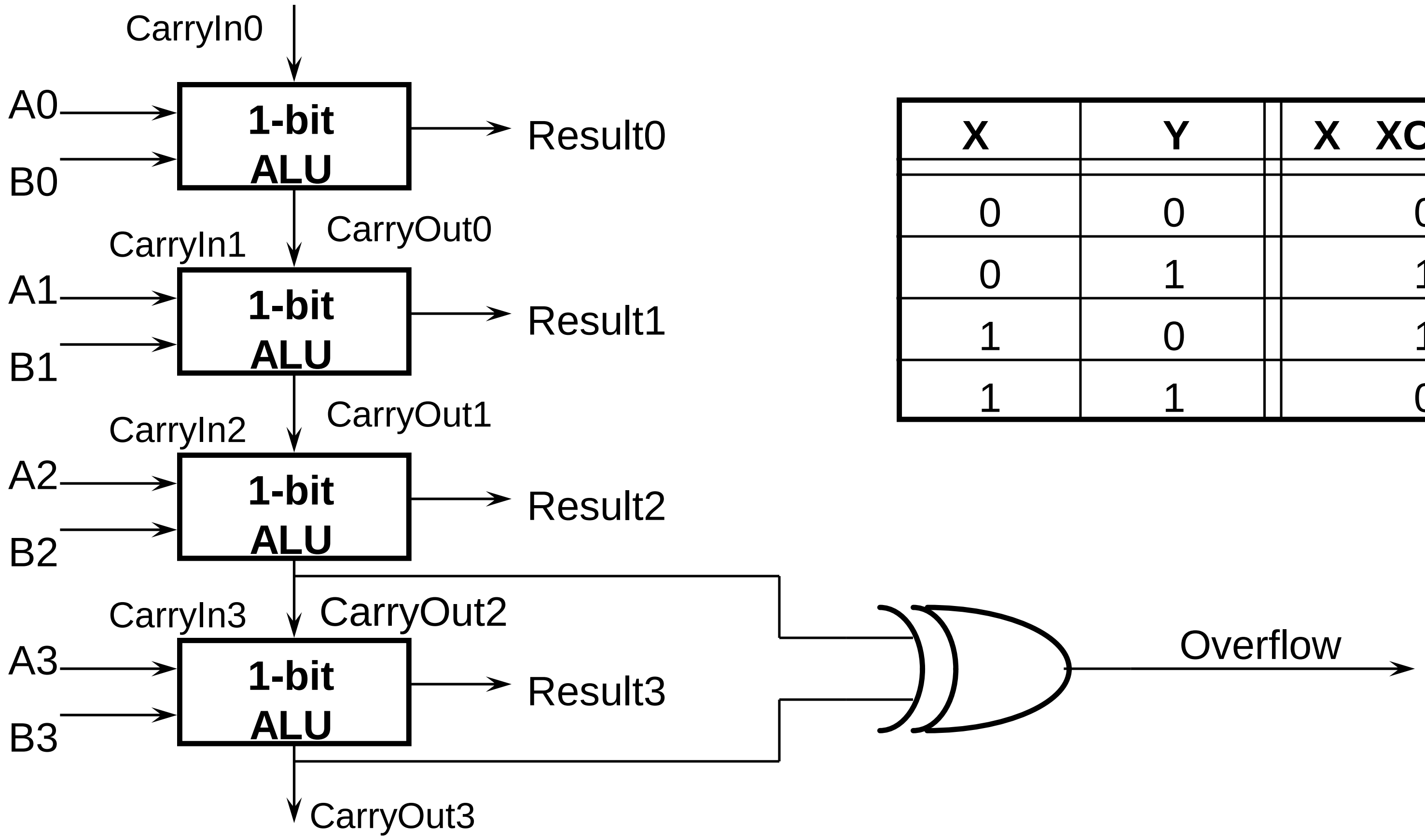
- Carry into MSB ! = Carry out of MSB





Overflow Detection Logic

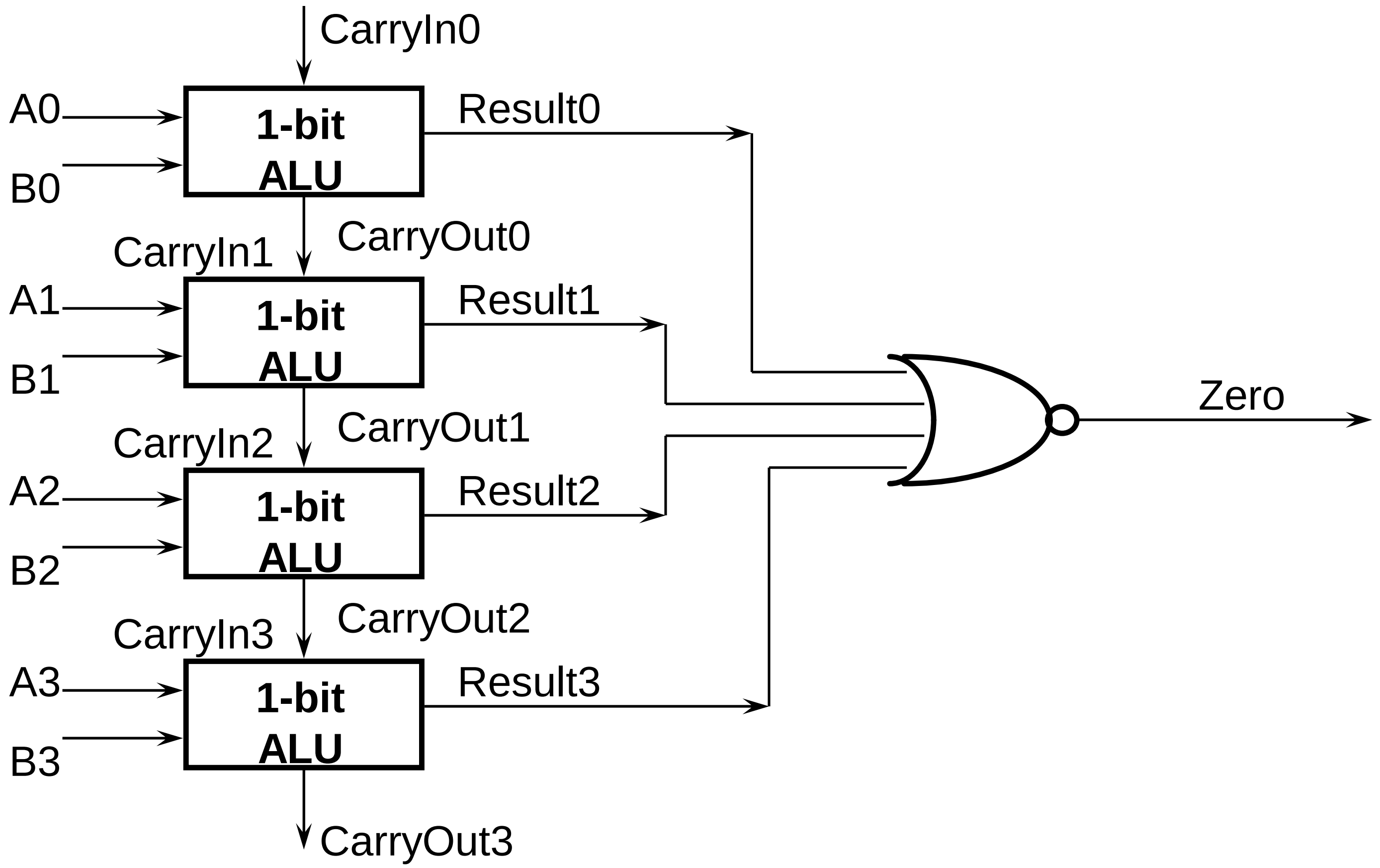
- **Carry into MSB ! = Carry out of MSB**
  - **For a N-bit ALU: Overflow = CarryIn[N - 1] XOR CarryOut[N - 1]**



X	Y	X XOR Y
0	0	0
0	1	1
1	0	1
1	1	0

Zero Detection Logic

- **Zero Detection Logic is just a one BIG NOR gate**
  - **Any non-zero input to the NOR gate will cause its output to be zero**

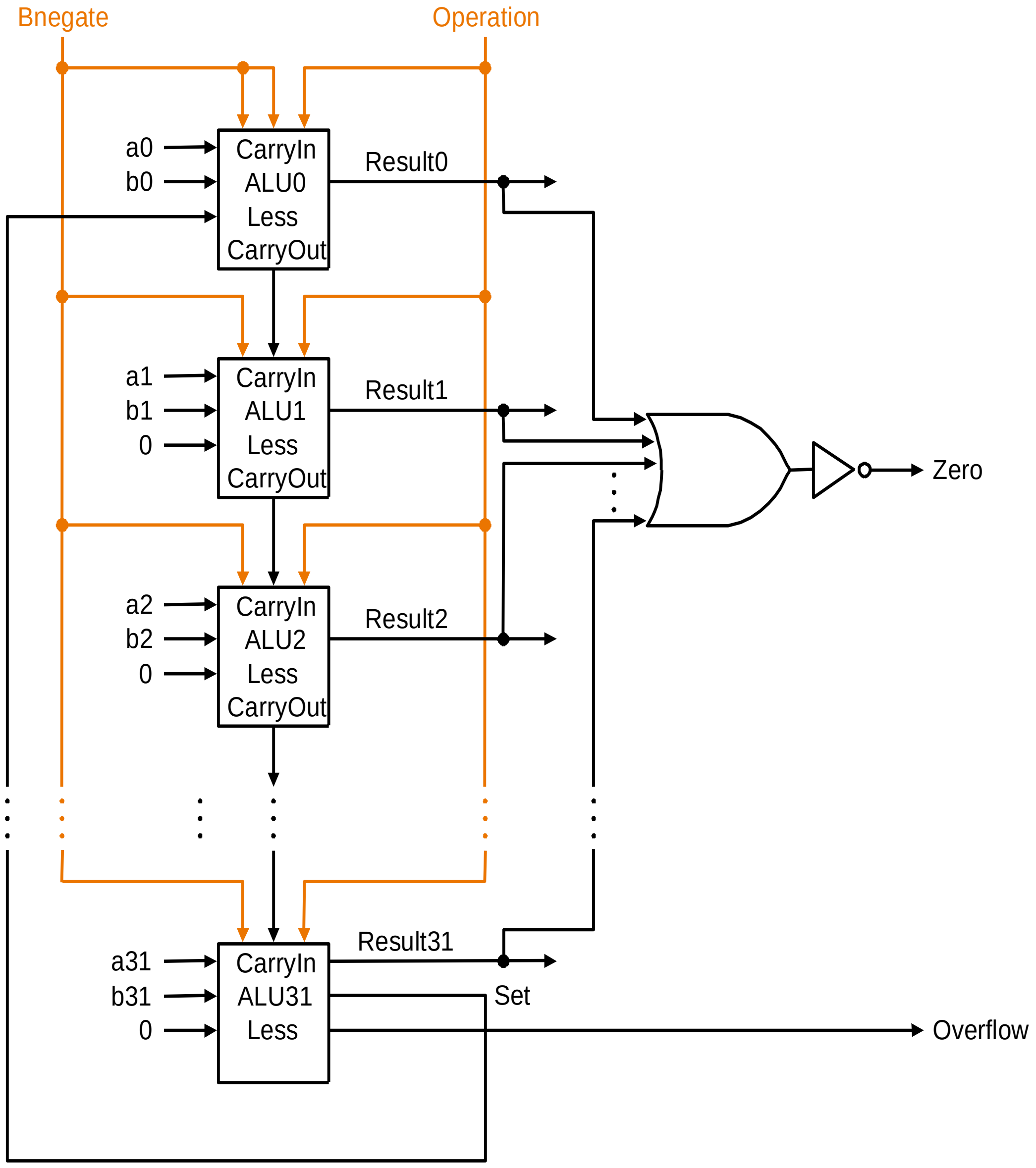


Test for equality

- Notice control lines:

```
000 = and
001 = or
010 = add
110 = subtract
111 = slt
```

- ***Note:*** *zero is a 1 when the result is zero!*

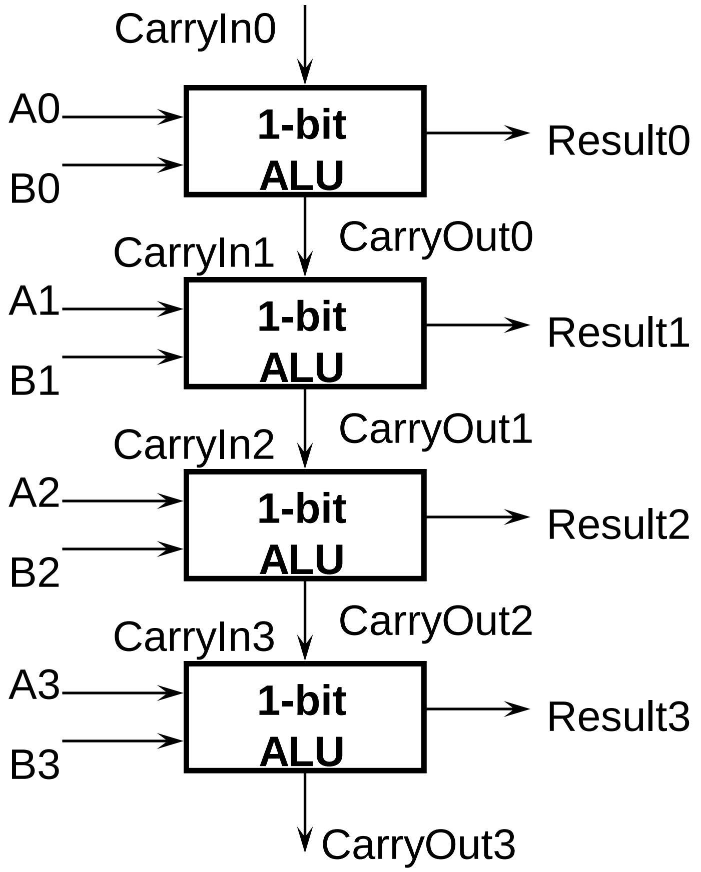


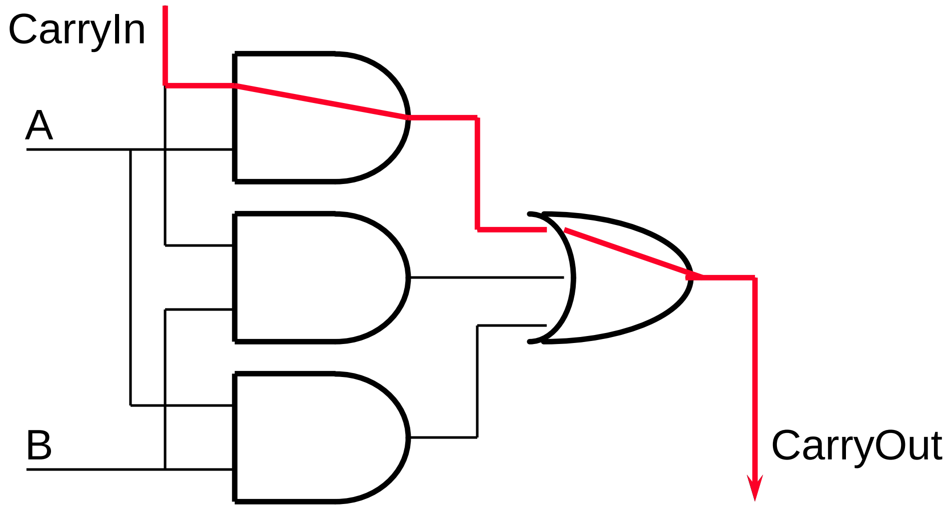
Conclusion

- 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
  - 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

The Disadvantage of Ripple Carry

- The adder we just built is called a Ripple Carry Adder
  - The carry bit may have to propagate from LSB to MSB
  - Worst case delay for a N-bit adder: 2N-gate delay





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

Can you see the ripple? How could you get rid of it?
Use sum-of-products

$c_1 = b_0c_0 + a_0c_0 + a_0b_0$

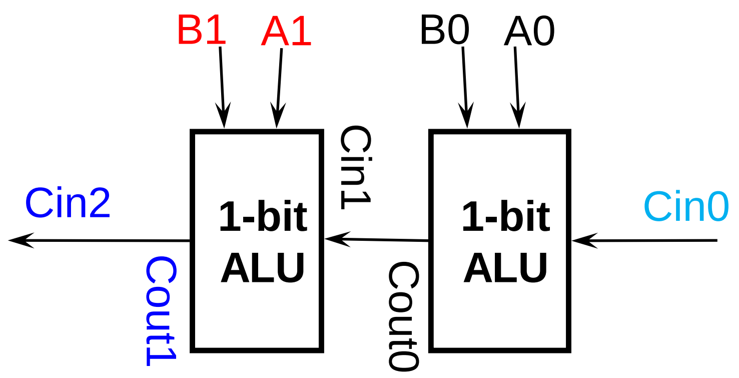
$c_2 = b_1c_1 + a_1c_1 + a_1b_1$ $\qquad c_2 = (b_1+a_1)(b_0c_0+a_0c_0+a_0b_0)+a_1b_1$

$c_3 = b_2c_2 + a_2c_2 + a_2b_2$ $\qquad c_3 = (b_2+a_2)(b_1c_1+a_1c_1+a_1b_1)+a_2b_2$

$c_4 = b_3c_3 + a_3c_3 + a_3b_3$ $\qquad c_4 = \ldots$

Not feasible! Why?

The Theory Behind Carry Lookahead



- Recalled: CarryOut = (B & CarryIn) | (A & CarryIn) | (A & B)
  - Cin1 = Cout0 = (B0 & Cin0) | (A0 & Cin0) | (A0 & B0)
  - Cin2 = Cout1 = (B1 & Cin1) | (A1 & Cin1) | (A1 & B1)
- Substituting Cin1 into Cin2:
  - Cin2 = (A1 & B0 & Cin0) | (A1 & A0 & Cin0) | (A1 & A0 & B0) |
    (B1 & B0 & Cin0) | (B1 & A0 & Cin0) | (B1 & A0 & B0) | (A1 & B1)
- Now define two new terms:

Generate Carry at Bit i	gi = Ai & Bi
Propagate Carry via Bit i	pi = Ai or Bi

Carry-lookahead adder

- An approach in-between our two extremes
- Motivation:
  - If we didn't know the value of carry-in, what could we do?
  - When would we always generate a carry? $g_i = a_i b_i$
  - When would we propagate the carry? $p_i = a_i + b_i$
- Did we get rid of the ripple?

$c_1 = b_0 c_0 + a_0 c_0 + a_0 b_0 = g_0 + p_0 c_0$

$c_2 = g_1 + p_1 c_1$ $\qquad c_2 = g_1 + p_1 g_0 + p_1 p_0 c_0$

$c_3 = g_2 + p_2 c_2$ $\qquad c_3 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0$

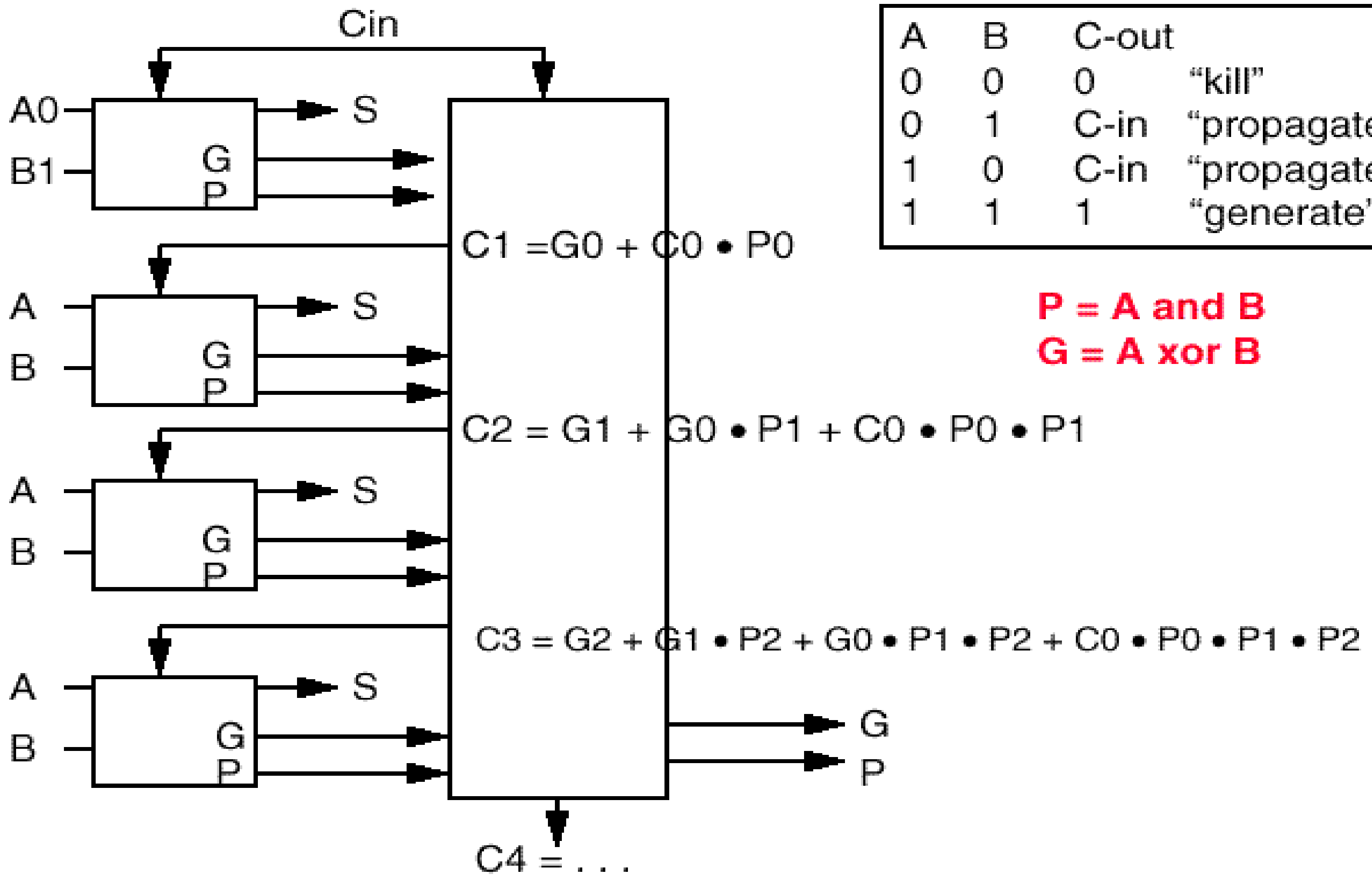
$c_4 = g_3 + p_3 c_3$ $\qquad c_4 =$

Feasible! Why?

The Theory Behind Carry Lookahead (Continued)

- Using the two new terms we just defined:
  - Generate Carry at Bit i gi = Ai & Bi
  - Propagate Carry via Bit i pi = Ai or Bi
- We can rewrite:
  - Cin1 = g0 | (p0 & Cin0)
  - Cin2 = g1 | (p1 & g0) | (p1 & p0 & Cin0)
  - Cin3 = g2 | (p2 & g1) | (p2 & p1 & g0) | (p2 & p1 & p0 & Cin0)
- Carry going into bit 3 is 1 if
  - We generate a carry at bit 2 (g2)
  - Or we generate a carry at bit 1 (g1) and
    bit 2 allows it to propagate (p2) ⇨ (p2 & g1)
  - Or we generate a carry at bit 0 (g0) and
    bit 1 as well as bit 2 allows it to propagate (p2 & p1 & g0)
  - Or we have a carry input at bit 0 (Cin0) and
    bit 0, 1, and 2 all allow it to propagate (p2 & p1 & p0 & Cin0)

Carry Look Ahead (Design trick: peek)

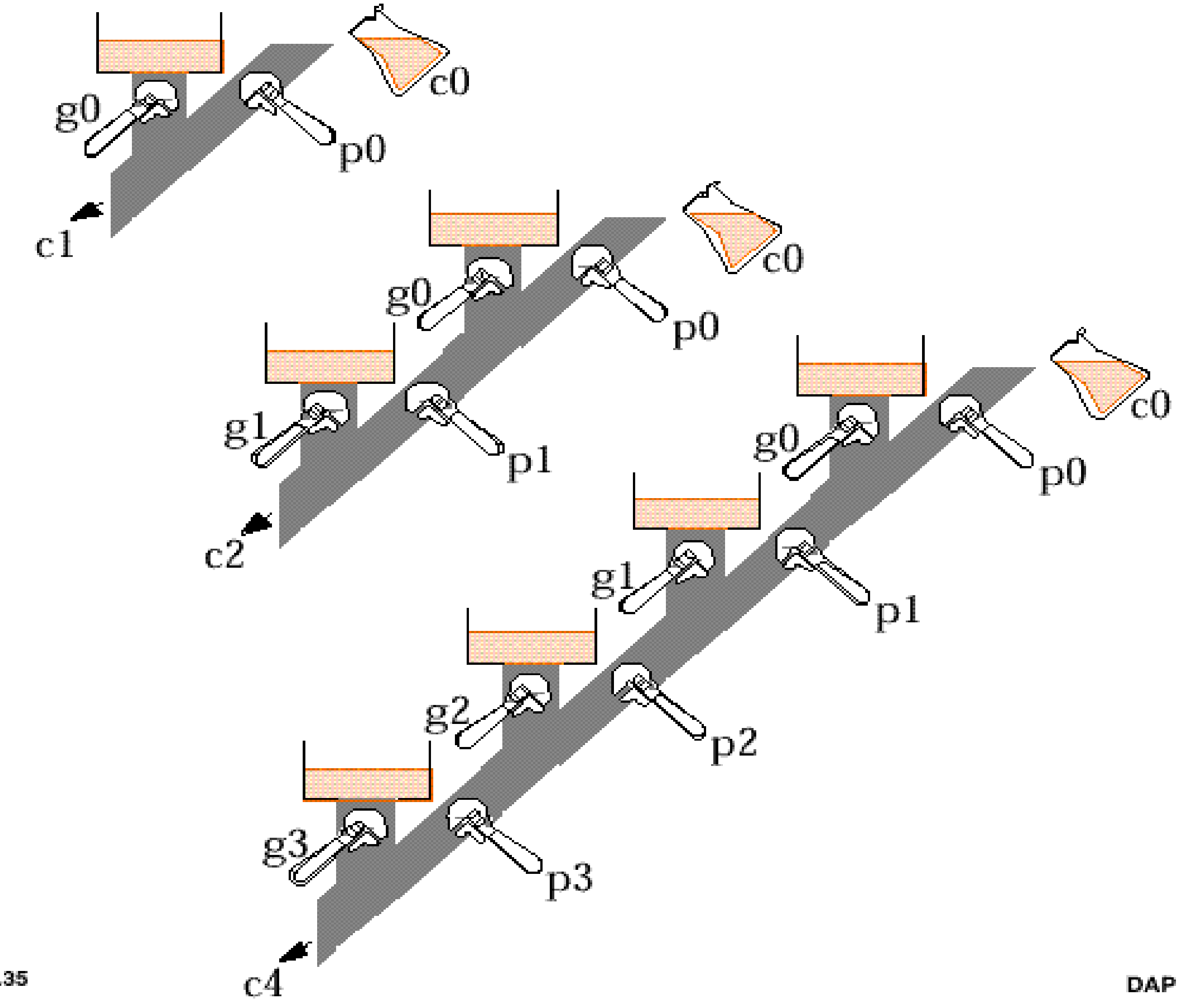


Cin
A0
B1
S
G
P
C1 =G0 + C0 • P0
A
B
S
G
P
C2 = G1 + G0 • P1 + C0 • P0 • P1
A
B
S
G
P
C3 = G2 + G1 • P2 + G0 • P1 • P2 + C0 • P0 • P1 • P2
A
B
S
G
P
G
P
C4 = . . .

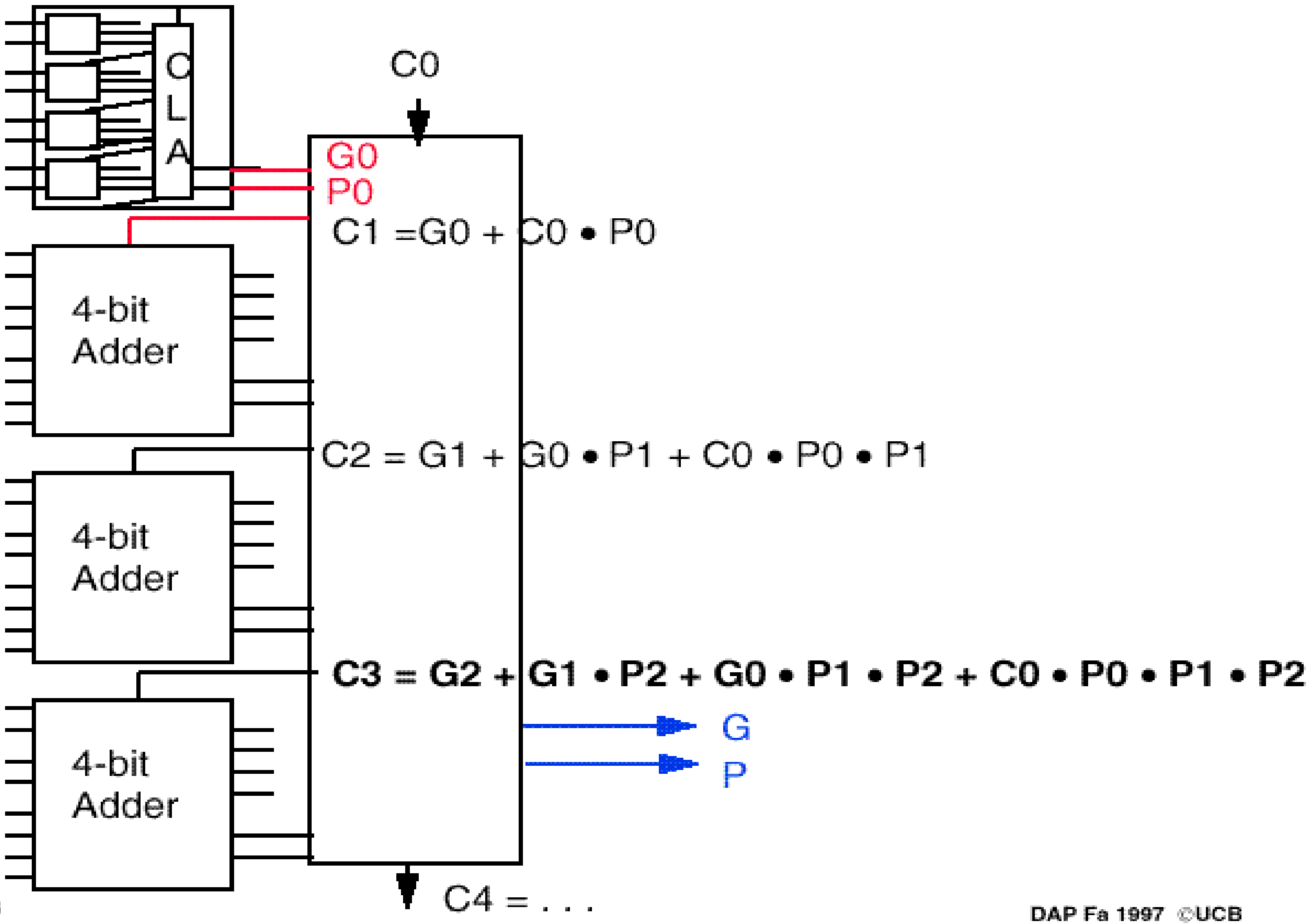
A	B	C-out	
0	0	0	“kill”
0	1	C-in	“propagate”
1	0	C-in	“propagate”
1	1	1	“generate”

P = A and B
G = A xor B

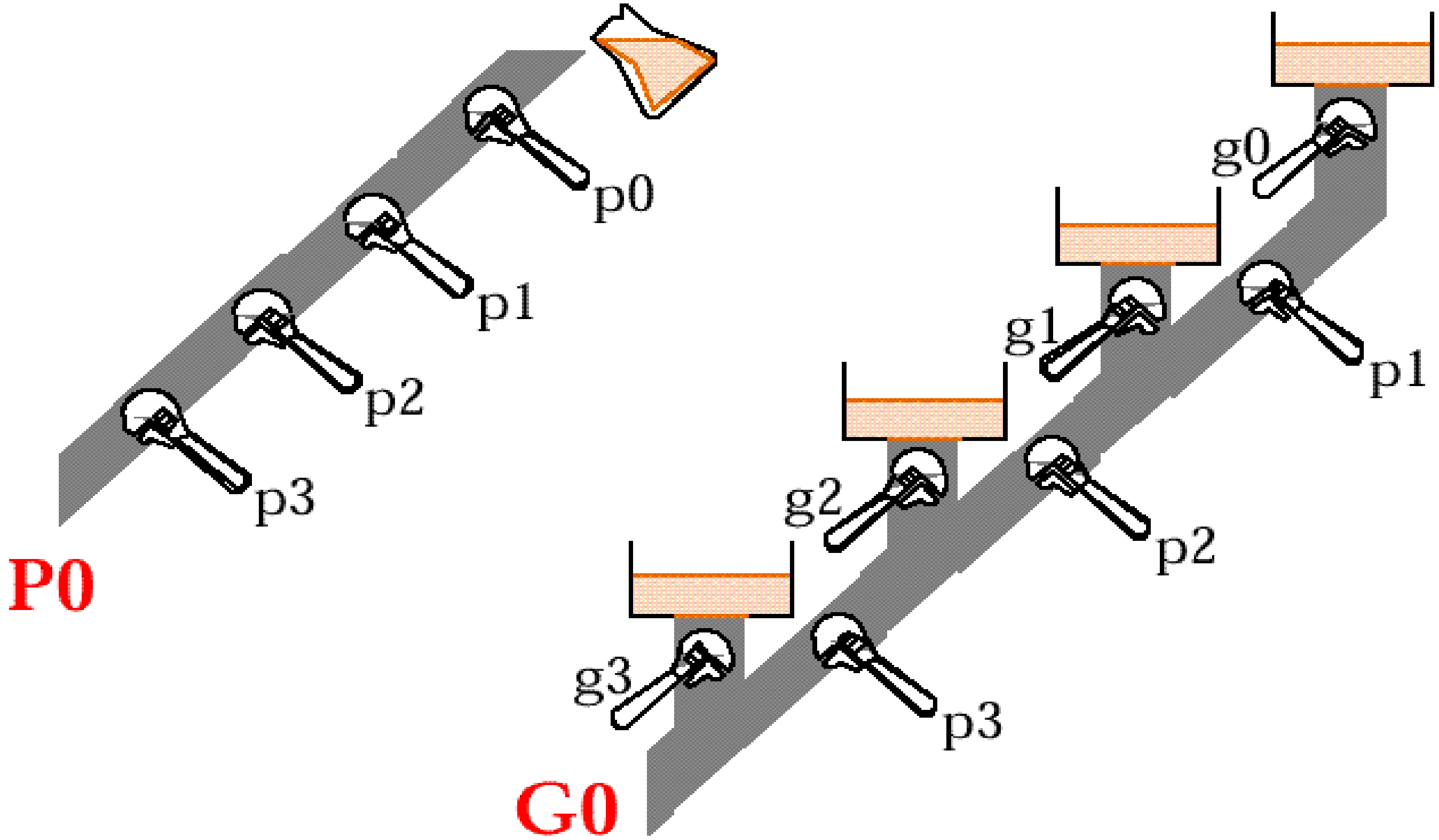
Plumbing as Carry Lookahead Analogy



Cascaded Carry Look-ahead (16-bit): Abstraction

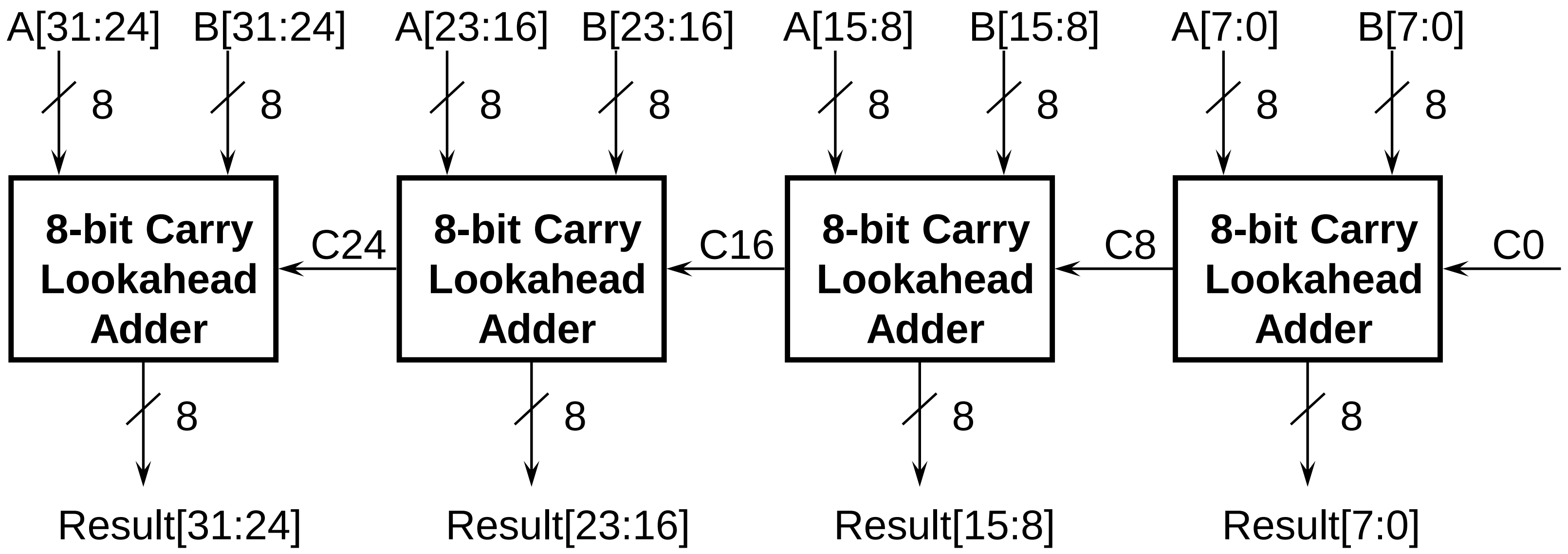

C
L
A
C0
G0
P0
C1 =G0 + C0 • P0
4-bit Adder
C2 = G1 + G0 • P1 + C0 • P0 • P1
4-bit Adder
C3 = G2 + G1 • P2 + G0 • P1 • P2 + C0 • P0 • P1 • P2
G
P
4-bit Adder
C4 = . . .

2nd level Carry, Propagate as Plumbing



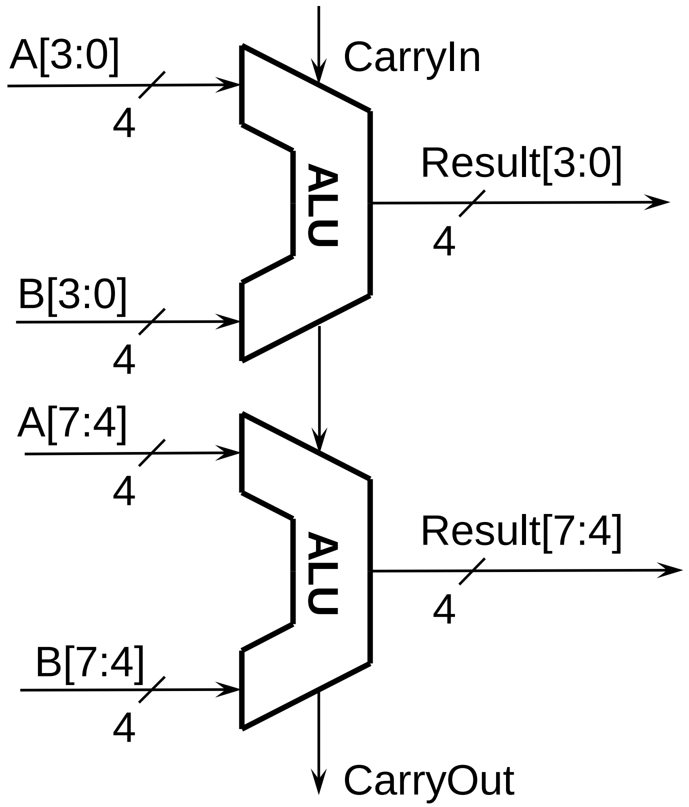
A Partial Carry Lookahead Adder

- **It is very expensive to build a "full" carry lookahead adder**
  - **Just imagine the length of the equation for Cin31**
- **Common practices:**
  - **Connects several N-bit Lookahead Adders to form a big adder**
  - **Example: connects four 8-bit carry lookahead adders to form a 32-bit partial carry lookahead adder**



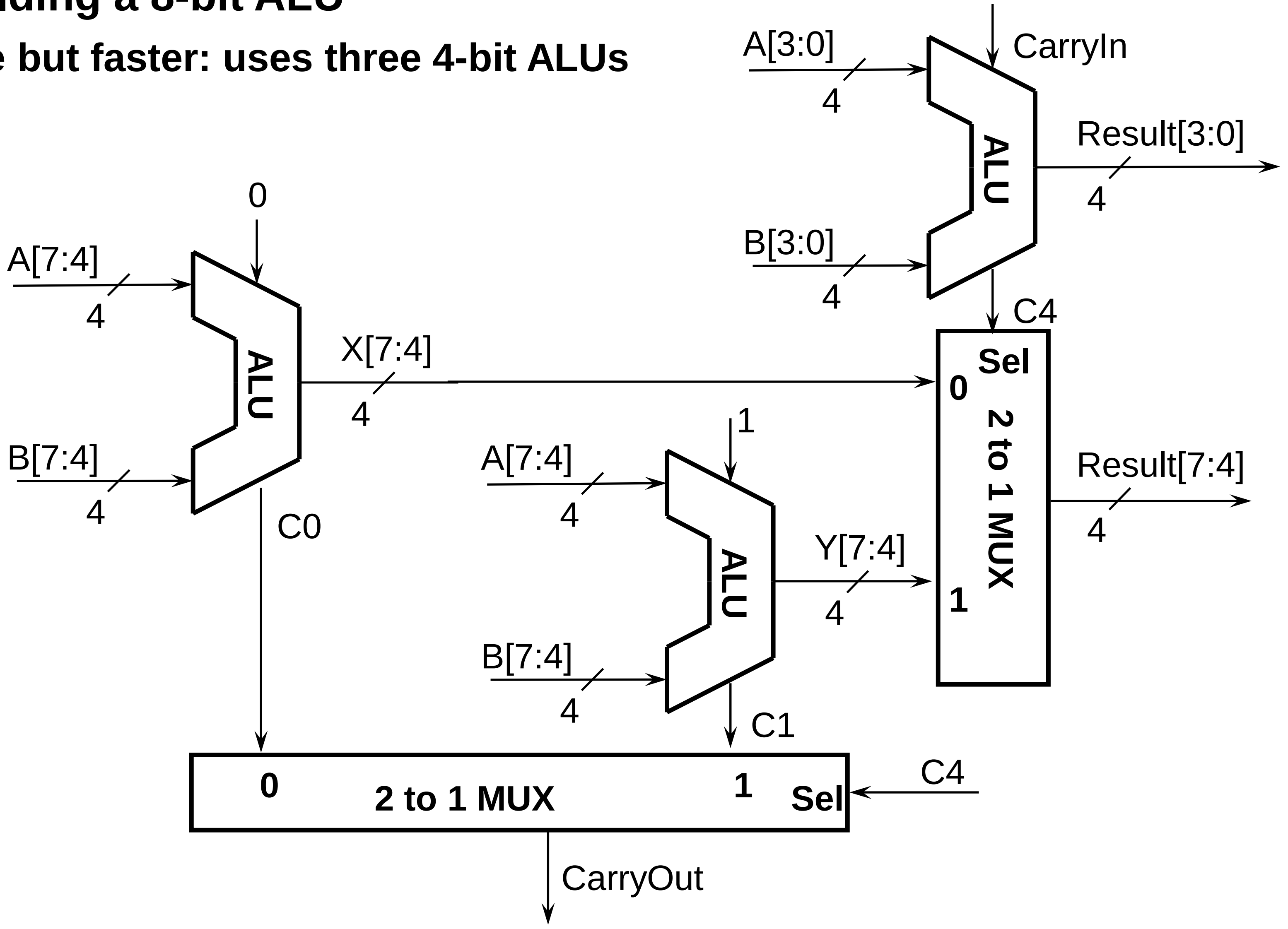
Carry Select Adder

- **Consider building a 8-bit ALU**
  - **Simple: connects two 4-bit ALUs in series**



Carry Select Adder (Continued)

- Consider building a 8-bit ALU
  - Expensive but faster: uses three 4-bit ALUs



Summary

- An Overview of the Design Process
  - Design is an iterative process-- successive refinement
  - Do NOT wait until you know everything before you start
- An Introduction to Binary Arithmetic
  - If you use 2’s complement representation, subtract is easy.
- ALU Design
  - Designing a Simple 4-bit ALU
  - Other ALU Construction Techniques
- On-line Design Notebook
  - Open a window and keep an editor running while you work
  - Refer to the handout as an example

To Get More Information

- Chapter 3 of your text book:
    - David Patterson & John Hennessy, *Computer Organization & Design*, 3rd Ed., @2004, Morgan Kaufmann Publishers.
- A good book :
    - David Winkel & Franklin Prosser, *The Art of Digital Design: An Introduction to Top-Down Design*, 2nd Ed., @1987, Prentice-Hall, Inc.