CMSC 411
Computer Architecture

Lecture 10

Arithmetic Logic Unit

Ergun Simsek

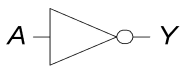
How can we design/build an adder?
How can we determine whether a<b?

Overview

- Logic gates and operations
- Constructing an Arithmetic Logic Unit
- Scaling bit operations to word sizes
- Optimization for carry generation

Single-Input Logic Gates

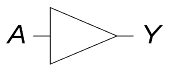
NOT



$Y = \overline{A}$

A	Y
0	1
1	0

BUF

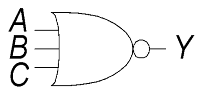


$Y = A$

A	Y
0	0
1	1

Multiple-Input Logic Gates

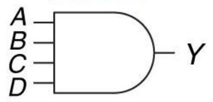
NOR3



$Y = \overline{A+B+C}$

A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

AND4

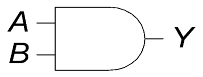


$Y = ABCD$

A	C	B	D	Y
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

Two-Input Logic Gates

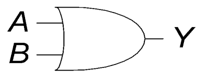
AND



$Y = AB$

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

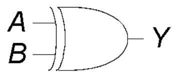
OR



$Y = A + B$

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

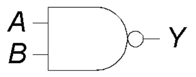
XOR



$Y = A \oplus B$

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

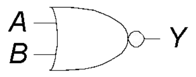
NAND



$Y = \overline{AB}$

A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

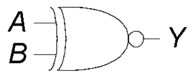
NOR



$Y = \overline{A + B}$

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

XNOR

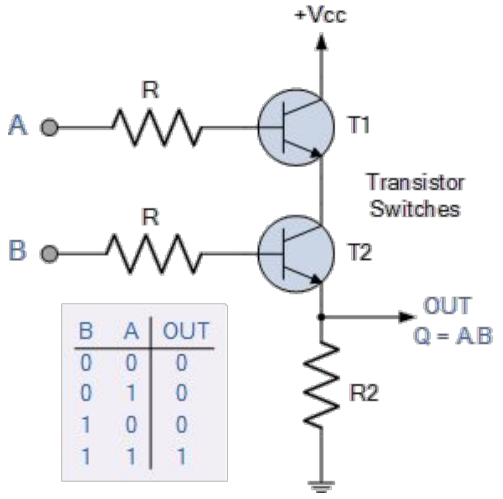


$Y = \overline{A \oplus B}$

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	1

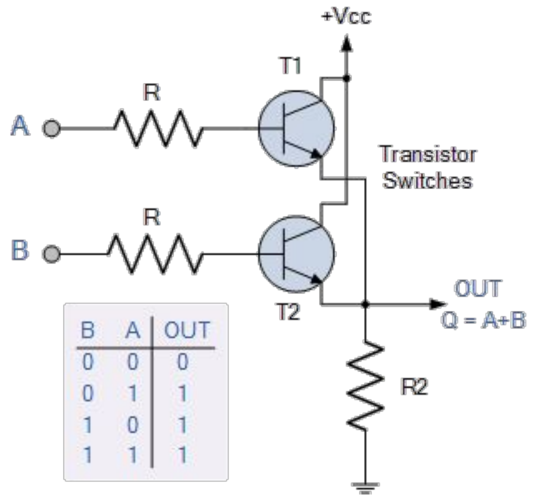
Building Gates with Transistors

AND



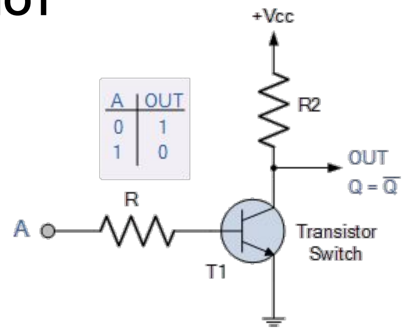
B	A	OUT
0	0	0
0	1	0
1	0	0
1	1	1

OR



B	A	OUT
0	0	0
0	1	1
1	0	1
1	1	1

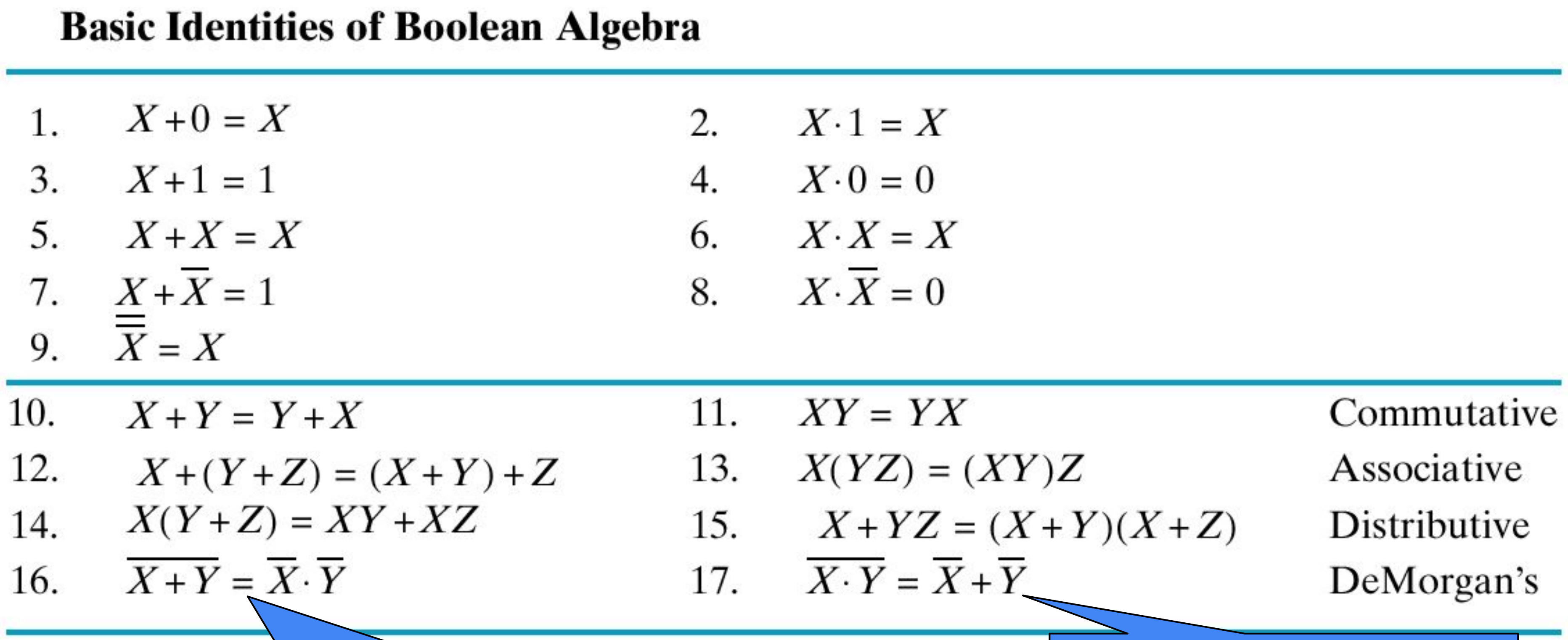
NOT



A	OUT
0	1
1	0

If you put two NOTs back to back, what do you get?

Boolean Algebra



Basic Identities of Boolean Algebra

1.	$X + 0 = X$	2.	$X \cdot 1 = X$	
3.	$X + 1 = 1$	4.	$X \cdot 0 = 0$	
5.	$X + X = X$	6.	$X \cdot X = X$	
7.	$X + \overline{X} = 1$	8.	$X \cdot \overline{X} = 0$	
9.	$\overline{\overline{X}} = X$			
10.	$X + Y = Y + X$	11.	$XY = YX$	Commutative
12.	$X + (Y + Z) = (X + Y) + Z$	13.	$X(YZ) = (XY)Z$	Associative
14.	$X(Y + Z) = XY + XZ$	15.	$X + YZ = (X + Y)(X + Z)$	Distributive
16.	$\overline{X + Y} = \overline{X} \cdot \overline{Y}$	17.	$\overline{X \cdot Y} = \overline{X} + \overline{Y}$	DeMorgan's

NOR = invert, then AND

NAND = invert, then OR

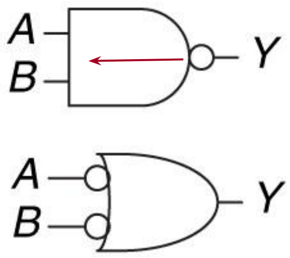
Left and right columns are *duals*
Replace ANDs and ORs, 0s and 1s

DeMorgan’s Theorem: Bubble Pushing

Bubble pushing: imagine the bubble at the output is being pushed towards the inputs

1. it becomes a bubble at every input, and
2. the shape of the gate changes from AND to OR, and vice versa

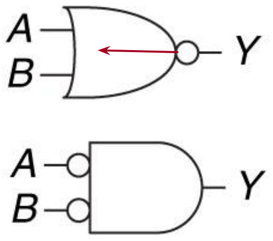
NAND



$Y = \overline{AB} = \overline{A} + \overline{B}$

A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

NOR



$Y = \overline{A+B} = \overline{A}\,\overline{B}$

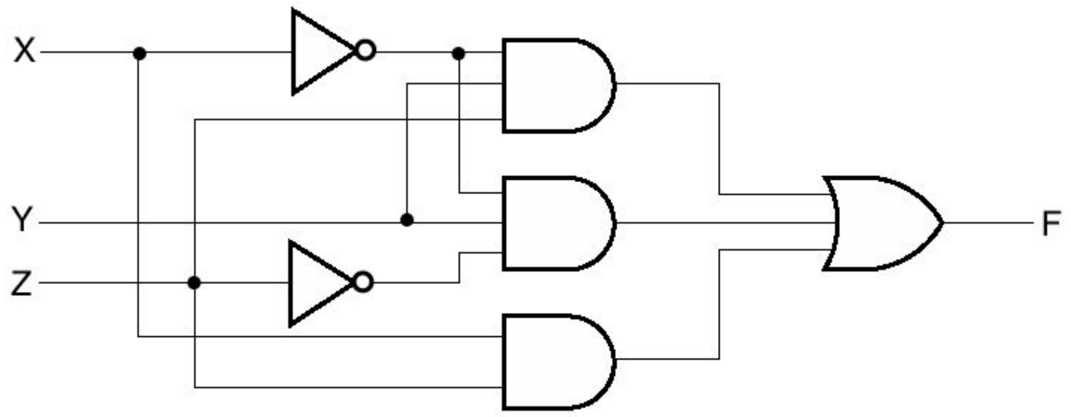
A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

Algebraic/Boolean Manipulation

Apply algebraic and Boolean identities to simplify expression

Example:

$$F = \overline{X}YZ + \overline{X}Y\overline{Z} + XZ$$



(a) $F = \overline{X}YZ + \overline{X}Y\overline{Z} + XZ$

Simplification Example

Apply

$$F = \overline{X}YZ + \overline{X}Y\overline{Z} + XZ$$

14. $X(Y+Z) = XY + XZ$

$$F = \overline{X}Y(Z + \overline{Z}) + XZ$$

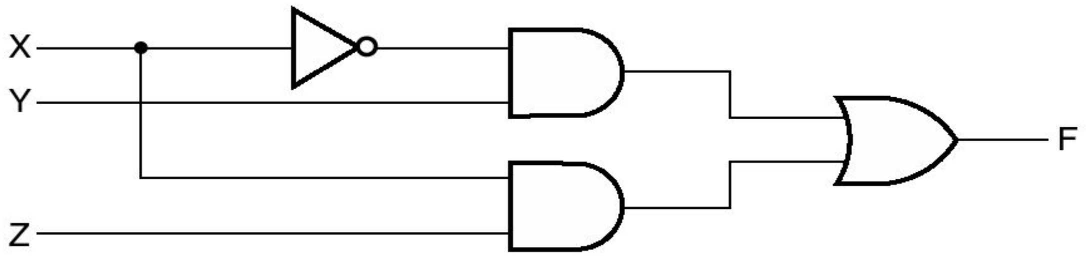
7. $X + \overline{X} = 1$

$$F = \overline{X}Y \cdot 1 + XZ$$

2. $X \cdot 1 = X$

$$F = \overline{X}Y + XZ$$

$$F = \overline{X}Y + XZ$$



(b) $F = \overline{X}Y + XZ$

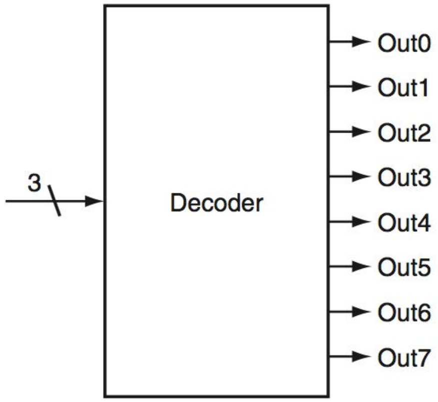
Fig. 2-4 Implementation of Boolean Function with Gates

Fewer Gates

Combinational logic

A logic system whose blocks do not contain memory and hence compute the same output given the same input.

Example: Decoders

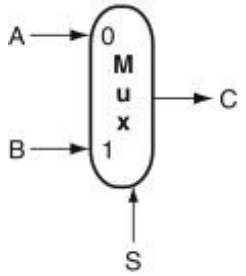


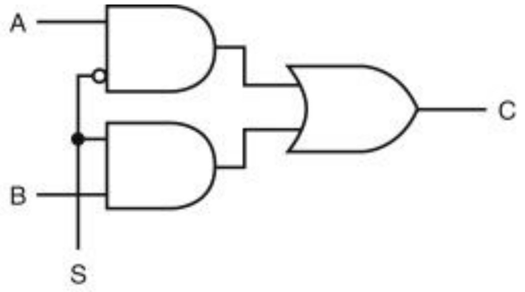
a. A 3-bit decoder

Inputs			Outputs							
12	11	10	Out7	Out6	Out5	Out4	Out3	Out2	Out1	Out0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

b. The truth table for a 3-bit decoder

Example: Multiplexor





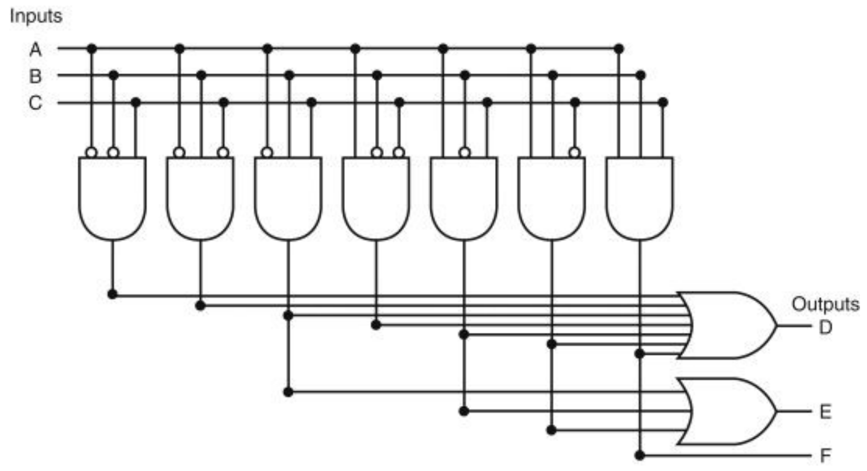
Combinational logic

Example: Programmable logic array (PLA)
Two stage logic elements functioning as a sum of products

Inputs			Outputs		
A	*B*	*C*	*D*	*E*	*F*
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	0	1

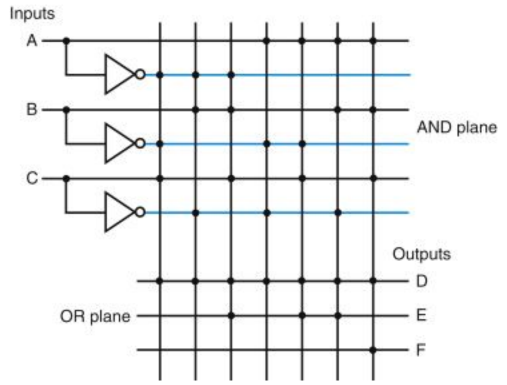
e.g.,
D = “A not” and “B not” and “C” or
“A not” and “B” and “C not” or
….

Products



Sums

=

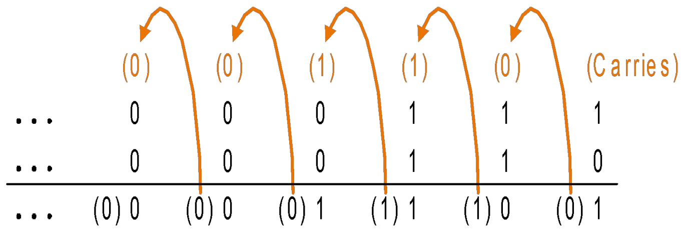


Addition and Subtraction

Digits are added bit by bit from right to left, with carries passed to the next digit to the left

Example:

```
   0000 0000 0000 0000 0000 0111 =  7
+  0000 0000 0000 0000 0000 0110 =  6
--------------------------------------------
   0000 0000 0000 0000 0000 1101 = 13
```



Subtraction uses addition: the appropriate operand is simply negated

Example:

```
     0000 0000 0000 0000 0000 0000 0000 0111 =  7
 -   0000 0000 0000 0000 0000 0000 0000 0110 =  6
----------------------------------------------------------------
     0000 0000 0000 0000 0000 0000 0000 0001 =  1
```

Or using two’s complement arithmetic

```
     0000 0000 0000 0000 0000 0000 0000 0111 =   7
 +   1111  1111  1111  1111  1111  1111 1111 1010 = - 6
----------------------------------------------------------------
     0000 0000 0000 0000 0000 0000 0000 0001 =   1
```

Arithmetic Overflow

Overflow occurs when the result of an operation cannot be represented with the available hardware

Most hardware detects and signals overflow via an exception

Some high level languages ignore overflow (e.g. C and Java) and some check for and handle it (e.g. Ada and Fortran)

addu/addiu doesn’t check for overflow

Try this in MARS or in SPIM (then replace addi with addiu!)

```
.text
li $t0, 0x7fffffff
addi $t0, $t0, 1
```

```
Runtime exception at 0x00400008: arithmetic overflow
```

Arithmetic Overflow

How can MIPS detect an overflow?

Look at these examples (using two's complement)

0110 + 0101 = 1 011, pos. no + pos no = neg. no!

1010 + 1001 = 10011 4 bit + 4 bit = 5 bit!

This is exactly how MIPS detect overflows

carry-out of the last full adder is equal to 1!

or

sign(a) = sign(b) ≠ sign(sum)

Operations	**Operand *A***	**Operand *B***	**Result**
A + B	≥ 0	≥ 0	< 0
A + B	< 0	< 0	≥ 0
A - B	≥ 0	< 0	< 0
A - B	< 0	≥ 0	≥ 0

Overflow or No Overflow?

```
  0 1 1 0
+ 0 1 1 1
----------
  1 1 0 1
```

OVERFLOW

```
  1 1 0 0
+ 1 1 0 1
----------
  1 0 0 1
```

NO OVERFLOW

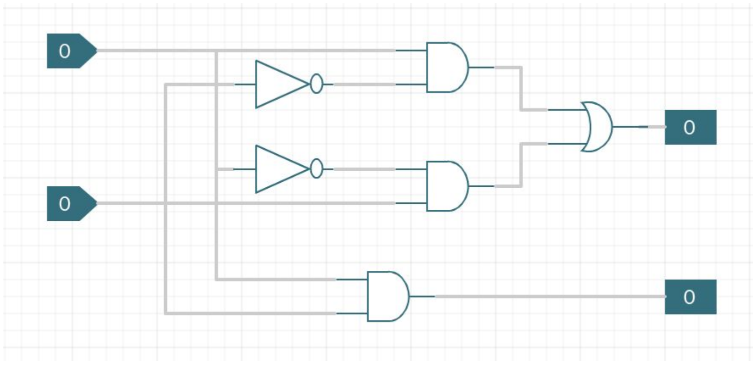
```
  0 0 0 1
+ 1 0 0 1
----------
  1 0 1 0
```

NO OVERFLOW

If an overflow is detected,

- PC will be changed to a predefined address in memory for exception
- Interrupted address is saved for resumption

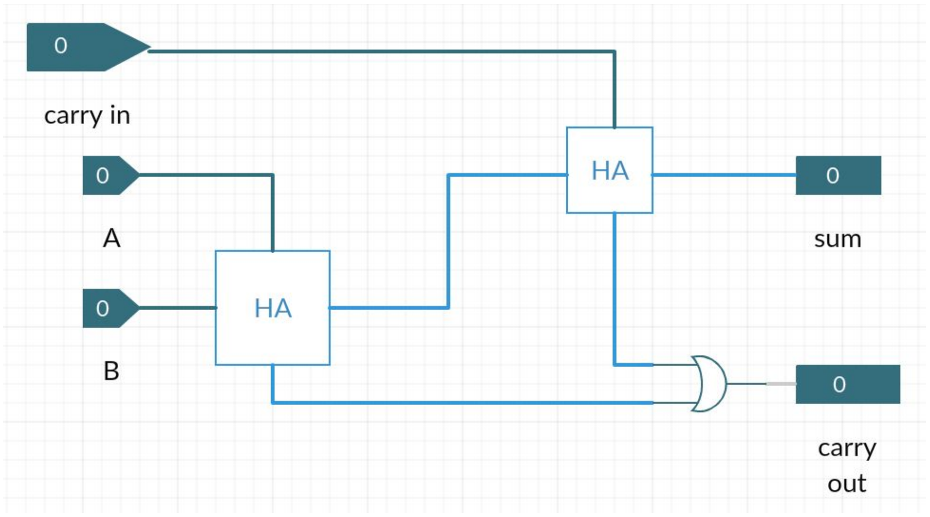
One-Bit Addition (a.k.a Half Adder)



sum = A XOR B

carry = A AND B

One-Bit Addition with Carry-in (a.k.a Full Adder)



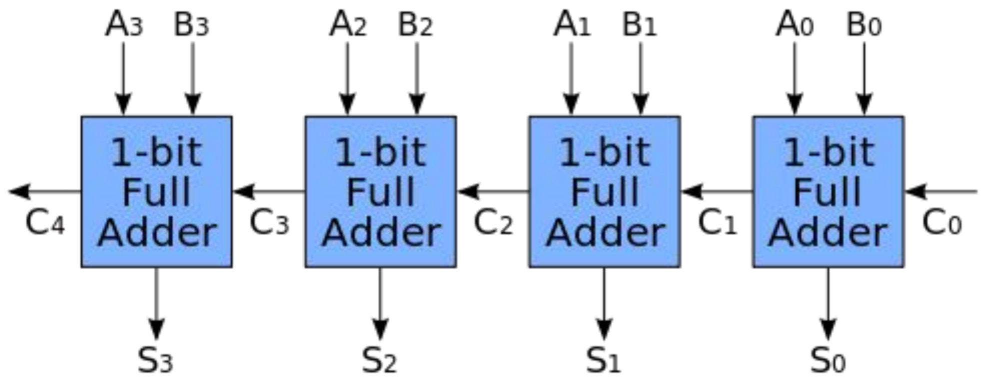
$\text{sum}_{\text{full}} = \text{carry}_{\text{in}}$ XOR sum_{h1}

$\text{carry}_{\text{out}} = \text{carry}_{\text{h1}}$ OR carry_{h2}

4-Bit Adder

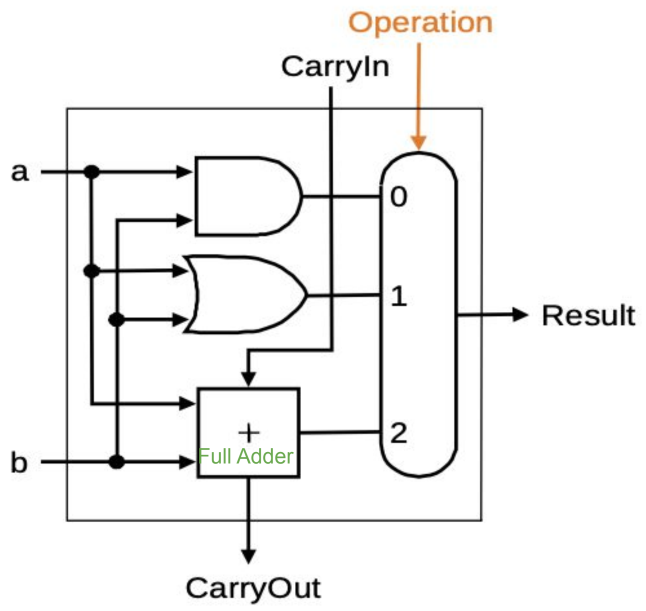
Full Adder has 3 inputs (two operands and a carry-in) and it generates a sum bit and a carry-out to be passed to the next 1-bit adder.

By using *N*-full adders in series, we can obtain *N*-bit adders.



Simple Arithmetic Logic Unit: One-Bit ALU

Handles AND, OR, and binary addition. The two **Operation** control bits determine which result is passed through the MUX (AND, OR, or sum).



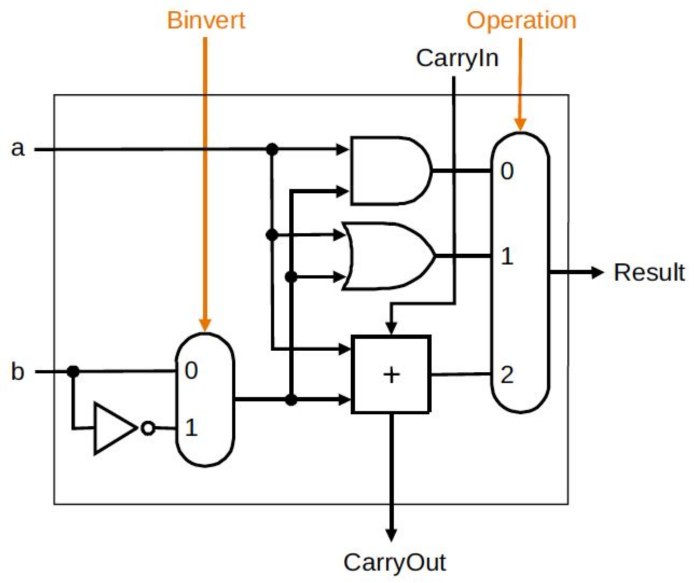
What about subtraction?

Subtraction can be performed by inverting the operand and setting the “CarryIn” input for the adder to 1 (i.e. using two’s complement)

By adding a multiplexor to the second operand, we can select either b or b

The Binvert line indicates a subtraction operation and causes the two’s complement of b to be used as an input

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



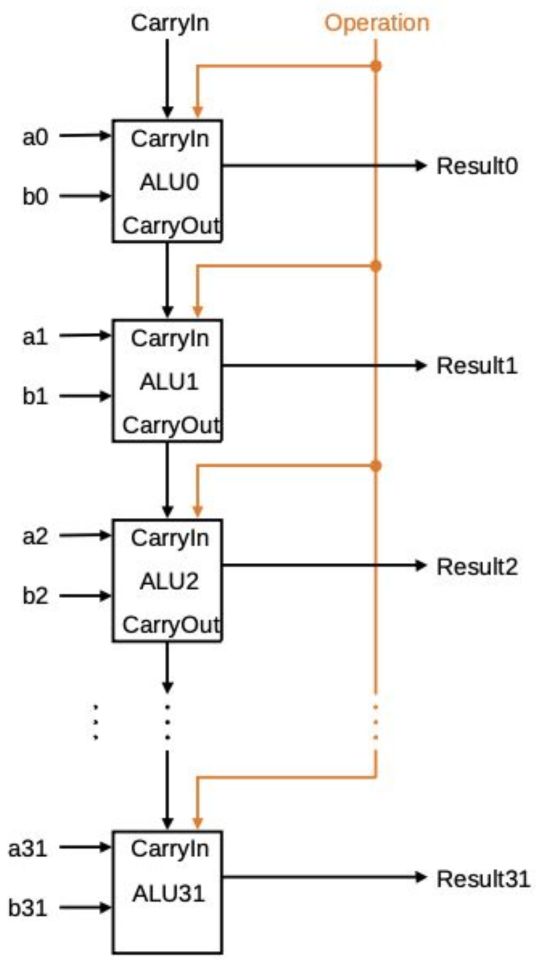
A 32-Bit ALU

A full 32-bit ALU can be created by connecting adjac
Carry-in and carry-out lines

The carry out of the least significant bit can ripple all
(ripple carry adder)

Ripple carry adders are slow since the carry propaga
sequentially

Subtraction can be performed by inverting the operar
input for the whole adder to 1 (i.e. using two's comple



MIPS “slt” instruction

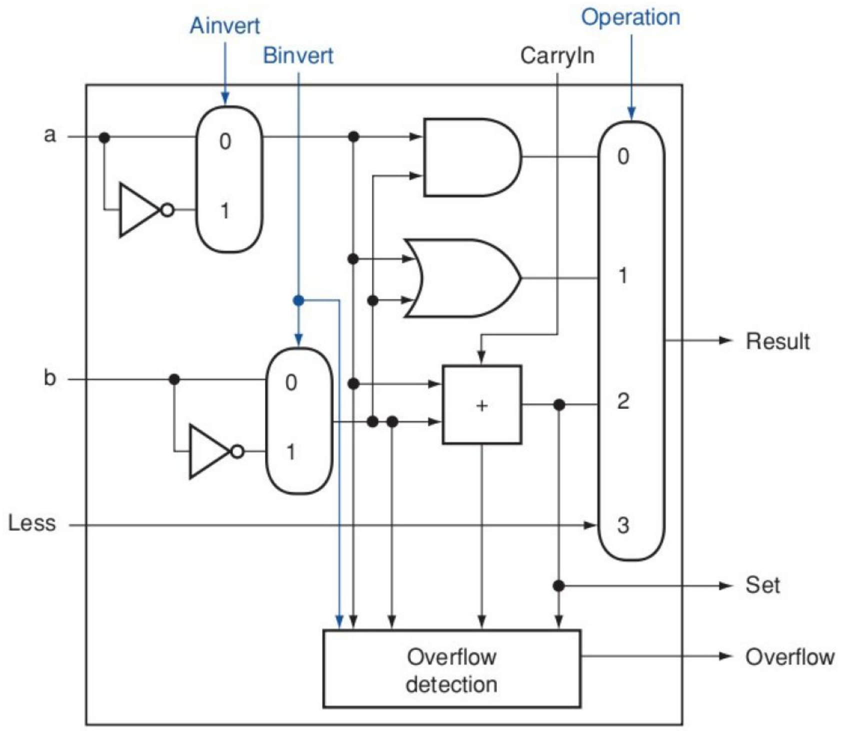
Each 1-bit ALU has an additional input called “Less”, which provides results for slt function. This input has value 0 for all but 1-bit ALU for the least significant bit.

Note that a < b if and only if (a-b) < 0, which requires “bit 31” to be 1.

Sets the least significant bit to the value of sign bit

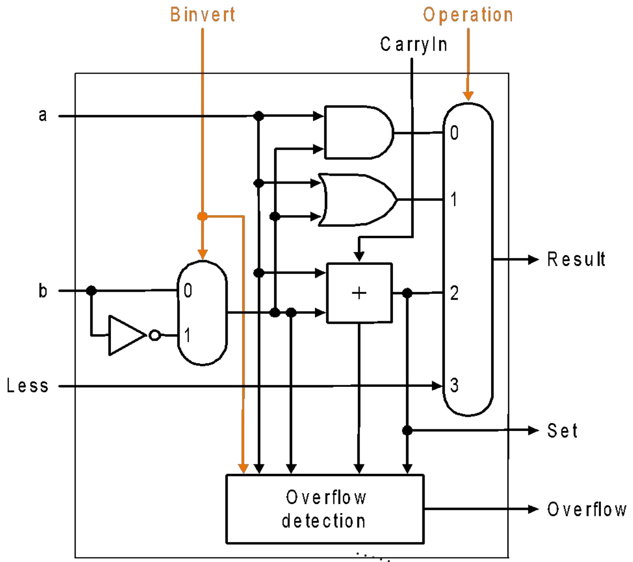
So, it returns

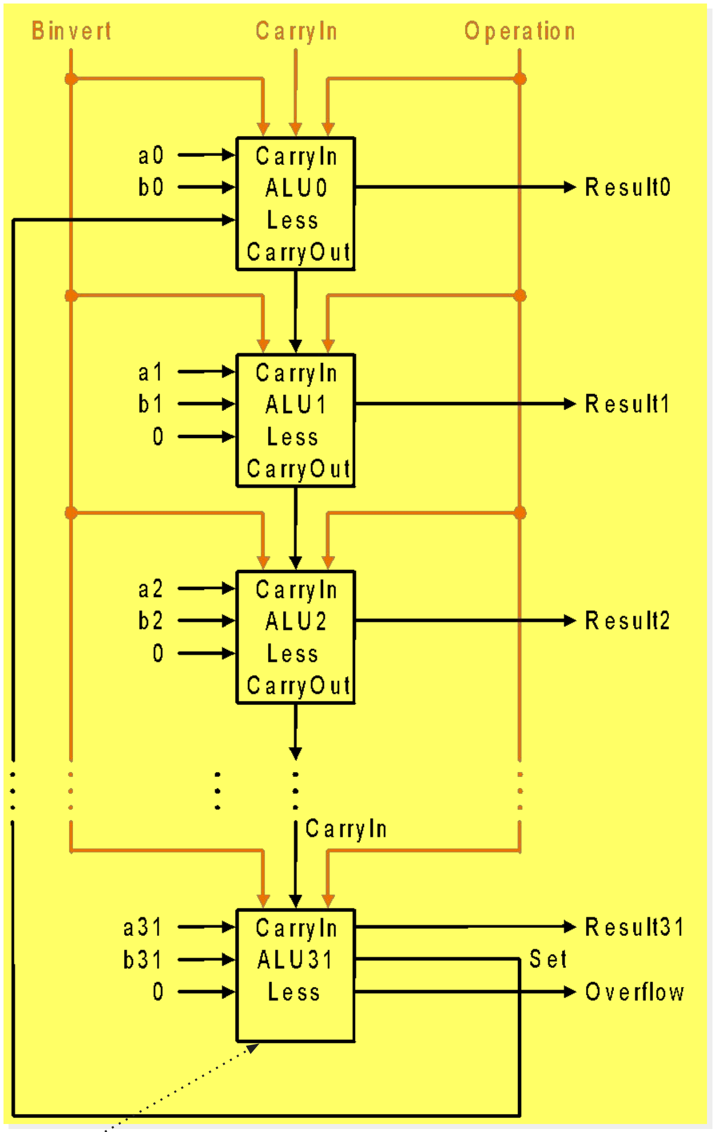
- 1 when a<b, because bit-31 is 1
- 0 otherwise, because bit-31 is 0



Pay attention to the inputs to the overflow detection unit

32-Bit ALU





Most Significant Bit

32-Bit Basic MIPS ALU

How can we do conditional branching?

Conditional Branching

“bne” and “beq” instruction compares two operands for equality

a = b if and only if (a-b) = 0

Then why don’t we

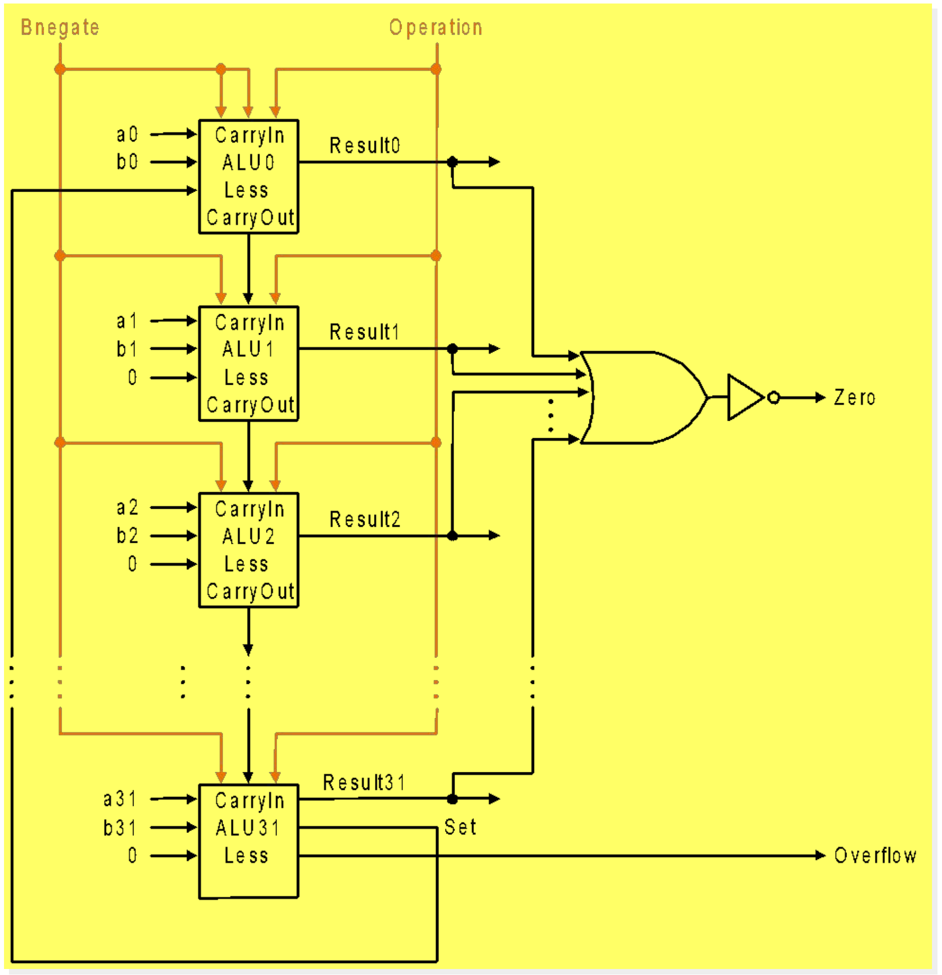
- subtract b from a for all the bits
- “or” all of them

if they are all 0, then the output will be 0 (iow, a=b)

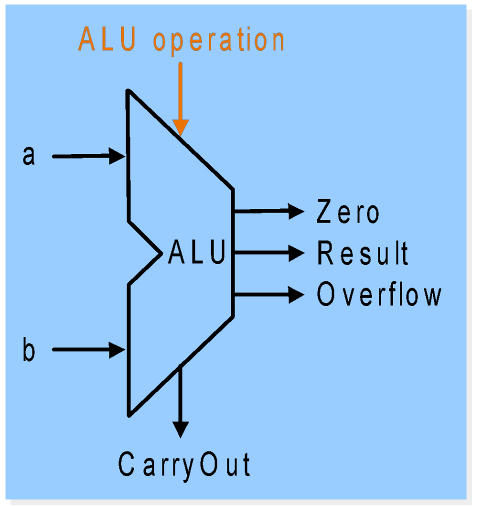
otherwise, output will be 1

This is exact opposite of what we want!

How can we do conditional branching?



MIPS' ALU Circuits



ALU Symbol

Optimizing Adder's Design

Problem with Ripple Carry Adders

- The CarryIn input depends on the operation in the adjacent 1-bit adder
- The result of adding most significant bits is only available after all other bits, i.e. after n-1 single-bit additions
- The sequential chain reaction is too slow to be used in time-critical hardware

Approach-1: Assume we have infinite hardware!

We know that $c1 = (b0 \,.\, c0) + (a0 \,.\, c0) + (a0 \,.\, b0)$

Similarly, $c2 = (b1 \,.\, c1) + (a1 \,.\, c1) + (a1 \,.\, b1)$

Then $c2 = (a1 \,.\, a0 \,.\, b0) + (a1 \,.\, a0 \,.\, c0) + (a1 \,.\, b0 \,.\, c0)$

$+ (b1 \,.\, a0 \,.\, b0) + (b1 \,.\, a0 \,.\, c0) + (b1 \,.\, b0 \,.\, c0) + (a1.\, b1)$

We can build a fast adder in the expense of exponentially growing gate number

Optimizing Adder's Design

Approach-2: Divide N-bits into M groups (e.g. 1011 1001 0110 1001), work on these small groups, propagate group's "CarryOut"s

We know that

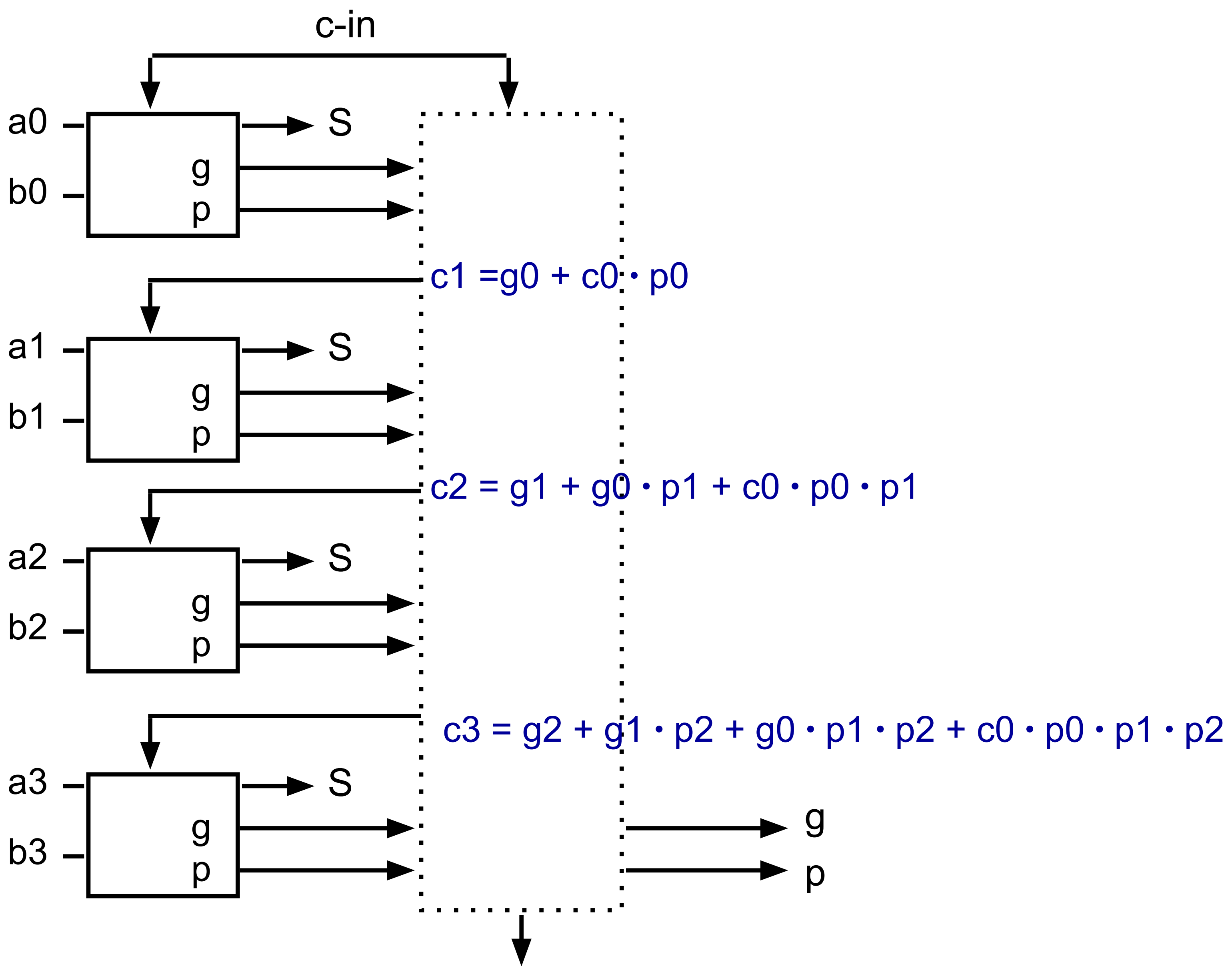
- if a = 0, b = 0, we won't have a CarryOut
- if a = 1, b = 1, we will always have a CarryOut
- Otherwise, CarryOut = CarryIn

p = a or b
g = a and b

$$\begin{aligned} c_{i+1} &= (b_i \,.\, c_i) + (a_i \,.\, c_i) + (a_i \,.\, b_i) \\ &= (a_i \,.\, b_i) + c_i \,.\, (a_i + b_i) \\ &= g_i + c_i \,.\, p_i \end{aligned}$$

a	b	c-out	
0	0	0	"kill"
0	1	c-in	"propagate"
1	0	c-in	"propagate"
1	1	1	"generate"

Carry Lookahead (propagate & generate)



c1 =g0 + c0 • p0

c2 = g1 + g0 • p1 + c0 • p0 • p1

c3 = g2 + g1 • p2 + g0 • p1 • p2 + c0 • p0 • p1 • p2

An Example

Determine g_i, p_i, P_i, G_i and carry out (C_4) values for these two 16-bit numbers:

```
a:  0001 1010 0011 0011
b:  1110 0101 1110 1011
```

Answer: Using the formula $g_i = (a_i \,.\, b_i)$ *and* $p_i = (a_i + b_i)$

g_i : 0000 0000 0010 0011

p_i : 1111 1111 1111 1011

The "super" propagates (P_0 , P_1 , P_2 , P_3) are calculated as follows:

$P_0 = p_3 . p_2 . p_1 . p_0 = 0$ $\quad$ $P_1 = p_7 . p_6 . p_5 . p_4 = 1$

$P_2 = p_{11} . p_{10} . p_9 . p_8 = 1$ $\quad$ $P_3 = p_{15} . p_{14} . p_{13} . p_{12} = 1$

The "super" generates (G_0 , G_1 , G_2 , G_3) are calculated as follows:

$G_0 = g_3 + (p_3 . g_2) + (p_3 . p_2 . g_1) + (p_3 . p_2 . p_1 . g_0) \quad = 0$

$G_1 = g_7 + (p_7 . g_6) + (p_7 . p_6 . g_5) + (p_7 . p_6 . p_5 . g_4) \quad = 1$

$G_2 = g_{11} + (p_{11} . g_{10}) + (p_{11} . p_{10} . g_9) + (p_{11} . p_{10} . p_9 . g_8) \quad = 0$

$G_3 = g_{15} + (p_{15} . g_{14}) + (p_{15} . p_{14} . g_{13}) + (p_{15} . p_{14} . g_{13} . g_{12}) \quad = 0$

Finally carry-out *(C_4) is*:

$C_4 = G_3 + (P_3 . G_2) + (P_3 . P_2 . G_1) + (P_3 . P_2 . P_1 . G_0) + (P_3 . P_2 . P_1 . P_0 . C_0) = 1$