Arithmetic Logic Unit (ALU)

Dr. Igor Ivkovic

iivkovic@uwaterloo.ca

[with material from “Computer Organization and Design” by Patterson and Hennessy, and “Digital Design and Computer Architecture” by Harris and Harris, both published by Morgan Kaufmann]

Objectives

- Binary Representations of Integers
- Basic Arithmetic Circuitry
- Composing Arithmetic Logic Unit (ALU)
- Multipliers and Dividers

Unsigned Binary Integers

- **Given an n-bit number:**

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

  - No sign bit present
  - Range: 0 to $+2^n - 1$

- **Example:**
  - $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$
    $= 0 + \ldots + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$
    $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$

- **Using 32 bits:**
  - Represent 0 to +4,294,967,295
  - How do we represent negative numbers?

One Idea: Signed/Magnitude Binary Integers

- **Given an n-bit number:**

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

  - Left-most bit represents the sign (1 for negative, 0 for non-negative)
  - Range: $-2^{n-1} + 1$ to $+2^{n-1} - 1$ (cannot represent -2^{n-1} and 2^{n-1})

- **Example:**
  - $\underline{1}000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$
    $= -(0 + \ldots + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0)$
    $= -(0 + \ldots + 8 + 0 + 2 + 1) = -11_{10}$

- **Using 32 bits with signed/magnitude representation:**
  - Represent –2,147,483,647 to +2,147,483,647
  - 0 is represented twice as $1000\ldots0000_2$ and $0000\ldots0000_2$
  - Not trivial to perform arithmetic operations such as addition and subtraction

Better Idea: Two's Complement Integers /1

- **Given an n-bit number:**

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

  - Left-most bit represents the sign (1 for negative, 0 for non-negative)
  - Range: -2^{n-1} to $+2^{n-1} - 1$ **(cannot represent 2^{n-1})**

- **Example:**
  - $\underline{\mathbf{1}}111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
    $\equiv \underline{\mathbf{-1\times 2^{31}}} + 1\times 2^{30} + \ldots + 1\times 2^2 + 0\times 2^1 + 0\times 2^0$
    $= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$

Multiply by – 1

- **Using 32 bits**
  - Represent –2,147,483,648 to +2,147,483,647
  - 0 is represented only once as $0000\ldots0000_2$
  - Easier to perform arithmetic operations such as addition and subtraction (see the discussion that follows)

Better Idea: Two's Complement Integers /2

- **Bit 31 or the left-most bit:**
  - The most significant bit (MSB) or the sign bit
  - **1**111 1111 1111 1111 1111 1111 1111 1100_2 (MSB)
  - MSB = 1 for negative numbers
  - MSB = 0 for non-negative numbers
- **Non-negative numbers have the same unsigned and two's complement representation**
- **Compute the negative value when MSB = 1:**
  - Start with -2^{n-1} and add $+2^k$ for each k-th bit from n – 2 to 0 that is 1; ignore the bits that are not 1
- **Example:**
  - Start with $1100_2 = -2^3 + 2^2 = -8 + 4 = -4_{10}$

Better Idea: Two's Complement Integers /3

- **Some specific numbers:**
  - Most positive: 0111 1111 … 1111
  - 2: 0000 0000 … 0010
  - 1: 0000 0000 … 0001
  - 0: 0000 0000 … 0000
  - –1: 1111 1111 … 1111
  - –2: 1111 1111 … 1110
  - –3: 1111 1111 … 1101
  - –4: 1111 1111 … 1100
  - –5: 1111 1111 … 1011
  - –6: 1111 1111 … 1010
  - –7: 1111 1111 … 1001
  - –8: 1111 1111 … 1000
  - Most negative: 1000 0000 … 0000

Notice the pattern for the negative numbers: counting the binary numbers in reverse

Signed Negation

- **How to negate a signed integer?**
  - **Easy: Compute the complement and add 1**
  - Compute the complement means convert 1 → 0 and 0 → 1

$$x + \bar{x} = 1111...111_2 = -1$$

$$\bar{x} + 1 = -x$$

- **Example: Negate +2**
  - +2 = 0000 0000 … 0010_2
  - –2 = 1111 1111 … 1101_2 + 1
    = 1111 1111 … 1110_2
  - Interesting: What if you have to negate 0?

Sign Extension

- **Representing a number using more bits**
  - The first goal is to preserve the numeric value
- **How to extend the bit representation?**
  - **Fill the extra bit slots with the MSB (the sign bit)**
  - For the unsigned values, fill the extra slots with 0
- **Example: 4-bit to 8-bit**
  - +5: 0101 => 0000 0101
  - –5: 1011 => 1111 1011
- **Example: 8-bit to 16-bit**
  - +2: 0000 0010 => 0000 0000 0000 0010
  - –2: 1111 1110 => 1111 1111 1111 1110

Logical Operations

- **Instructions for bitwise manipulation:**

Operation	C	Java	MIPS
Shift left	<<	<<	`sll`
Shift right	>>	>>>	`srl`
Bitwise AND	&	&	`and, andi`
Bitwise OR	|	|	`or, ori`
Bitwise NOT	~	~	`nor`

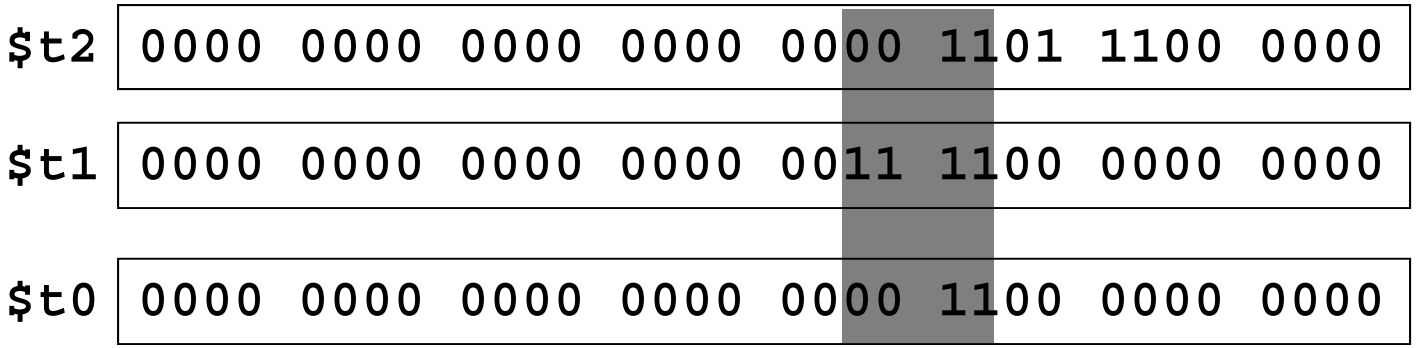
  - Useful for extracting and inserting groups of bits in a word
  - **MIPS instructions discussed later in the course**

Shift Operations

- **Shift left logical (sll)**
  - Shift left and fill with 0 bits
  - sll by i bits multiplies by 2^i
  - **Example: 11001 << 2 = 00100**
- **Shift right logical (srl)**
  - Shift right and fill with 0 bits
  - srl by i bits divides by 2^i (unsigned only)
  - **Example: 11001 >> 2 = 00110**
- **We will use shifters in creation of multipliers**
  - More on shifters in the context of MIPS later in the course

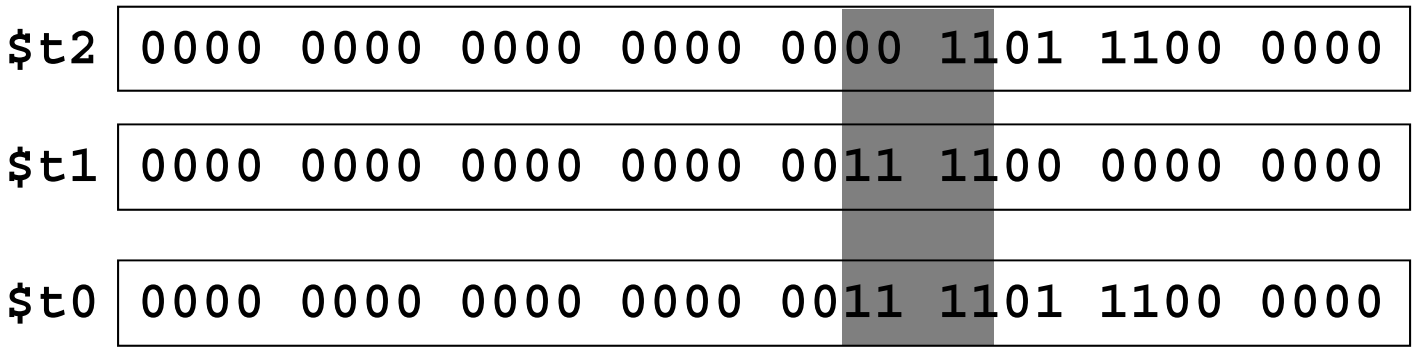
AND Operation

- **AND Operation**
  - **Useful to mask bits in a word**
  - Select some bits, clear others to 0
  - MIPS: **`and $t0, $t1, $t2`**



OR Operation

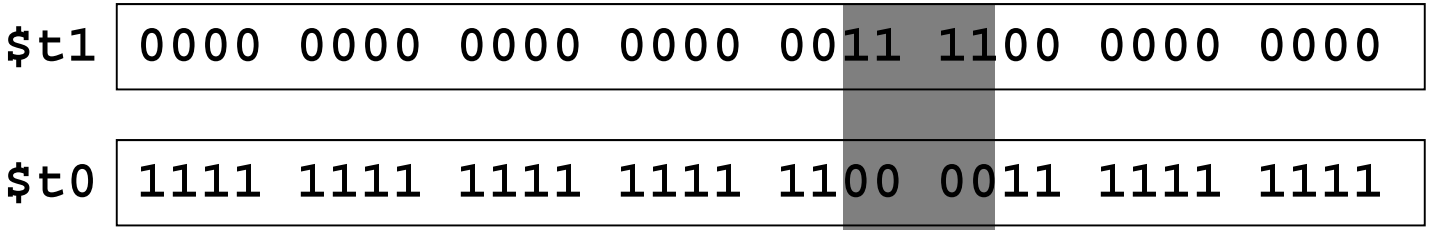
- **OR Operation**
  - **Useful to include bits in a word**
  - Set some bits to 1, leave others unchanged
  - MIPS: **`or $t0, $t1, $t2`**



NOT Operation

- **NOT Operation**
  - **Useful to invert bits in a word**
  - Change 0 to 1, and 1 to 0
  - MIPS has a NOR 3-operand instruction
  - a NOR b == NOT (a OR b)
  - MIPS : **`nor $t0, $t1, $0`**

Register $0 always reads zero



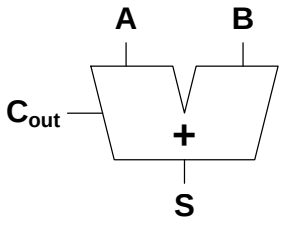
Arithmetic Circuitry

- **Digital building blocks:**
  - So far, we have covered gates, multiplexers, decoders, and registers
  - We will use these to build arithmetic circuits, counters, memory arrays, and logic arrays
- **Digital building blocks demonstrate hierarchy, modularity, and regularity:**
  - Hierarchy of simpler components
  - Well-defined interfaces and functions
  - Regular structure easily extends to different sizes
- **We will use all of these building blocks to create a microprocessor**

Adders /1

- **Adders:**
  - Add two input bits, A and B, and output the result, S
  - If there is carry-over, it is outputted as C_{out}
  - Carry-over is added to the next adder in the chain, or it indicates overflow if it is the last adder in the chain

Half Adder



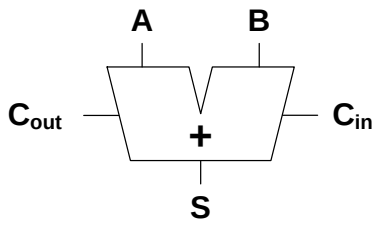
A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

S =

C_{out} =

Carry-over when A and B are both 1

Full Adder



C_{in}	A	B	C_{out}	S
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

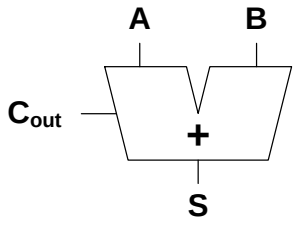
S =

C_{out} =

Adders /2

- **Adders:**
  - Half adder has no carry-over input from the previous adder
  - Full adder represents a chained adder and input from the previous adder is given as C_{in}
- **How to create an adder from gates?**
  - Use the logical formulas shown below each truth table

Half Adder

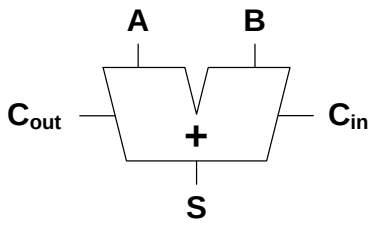


A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$S = A \oplus B$

$C_{out} = AB$

Full Adder



C_{in}	A	B	C_{out}	S
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

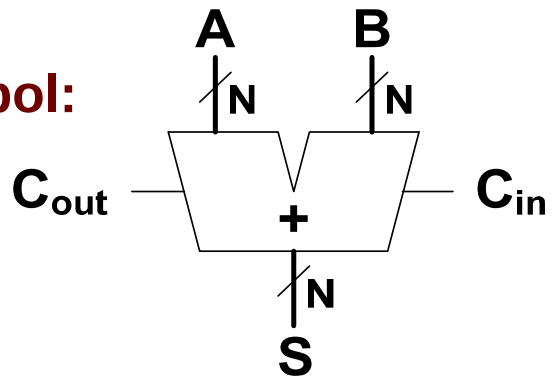
$S = A \oplus B \oplus C_{in}$

$C_{out} = AB + AC_{in} + BC_{in}$

Adders /3

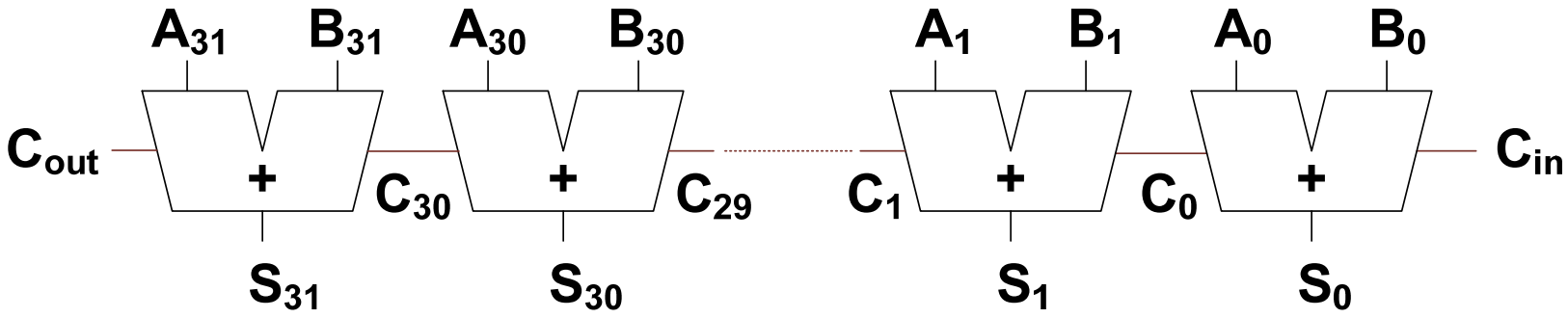
- **Multibit Adders (CPAs)**
  - When composing multibit adders, different strategies for propagating carry-over bits apply
- **Types of carry propagate adders (CPAs):**
  - Ripple-carry (slow)
  - Carry-lookahead (fast)
  - Prefix (faster)
  - Carry-lookahead and prefix adders can perform faster for large adders but require more hardware

Symbol:



Ripple-Carry Adder

- **Ripple-Carry Adder**
  - Simply chain 1-bit adders together
  - Carry ripples through entire chain
  - Advantage: simpler operational semantics
  - Disadvantage: slower performance

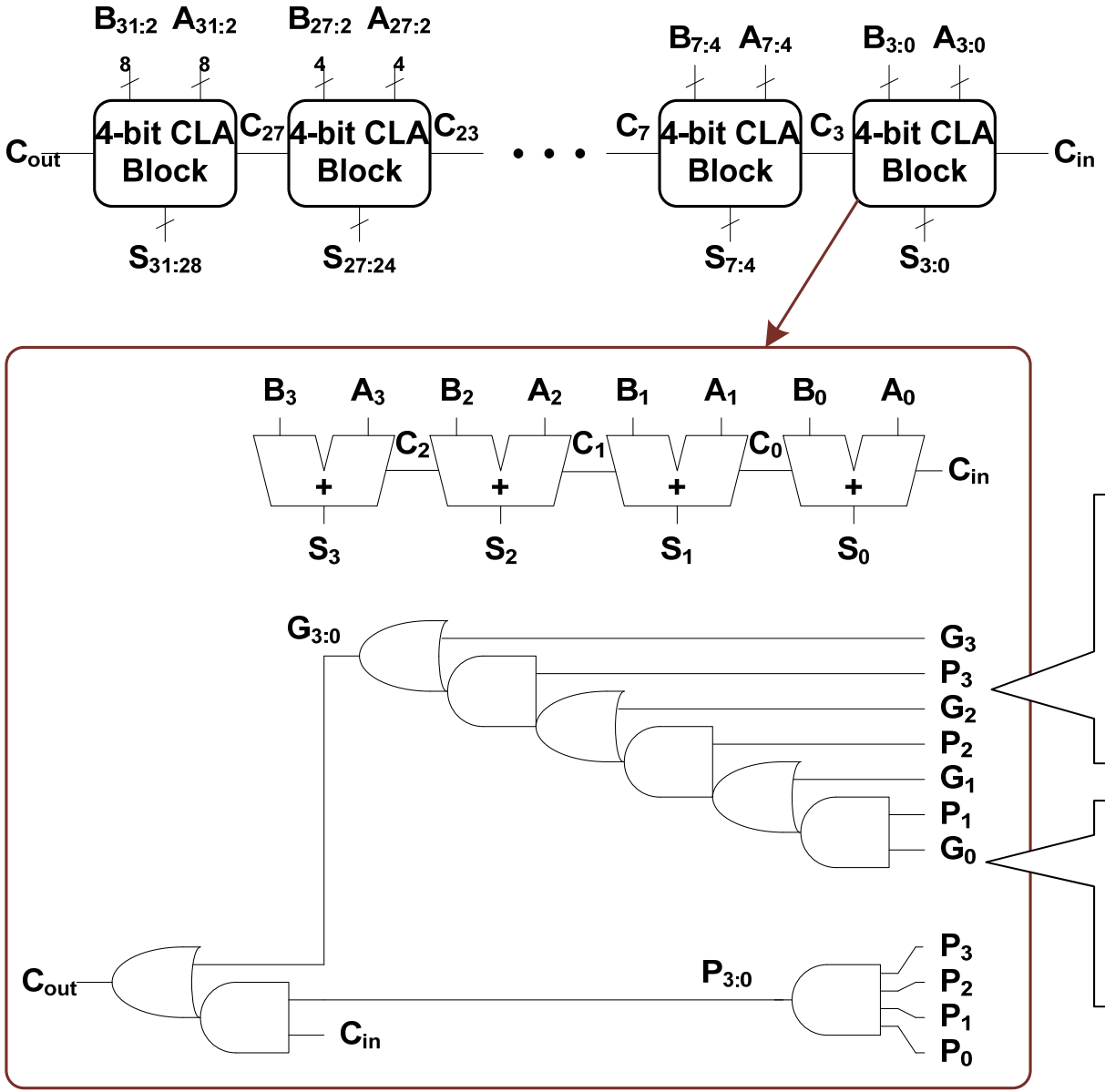


- **Ripple-Carry Adder Delay**
  - $\mathbf{t_{ripple} = N \times t_{FA}}$, where t_{FA} is the delay of a full adder

Carry-Lookahead Adder /1

- **Carry-Lookahead Adder**
  - Instead of waiting for each adder to compute the carry-out bit, divide the addition into k-bit blocks and precompute carry-outs
  - Compute carry-out (C_{out}) for k-bit blocks using generate, G_i, and propagate, P_i, signals
- **Operational Semantics:**
  - Column i adder produces a carry-out by either generating a carry-out or propagating a carry-in to the carry-out
  - Generate (G_i) and propagate (P_i) signals for each column:
    - Column i will generate a carry-out if A_i AND B_i are both 1
      $\mathbf{G_i = A_i\ B_i}$
    - Column i will propagate a carry-in to the carry-out if A_i OR B_i is 1
      $\mathbf{P_i = A_i + B_i}$
    - The carry-out of column i (C_i) is:
      $\mathbf{C_i = A_i\ B_i + (A_i + B_i)C_{i-1} = G_i + P_i\ C_{i-1}}$

Carry-Lookahead Adder /2



$G_i = A_i B_i$

$P_i = A_i + B_i$

$C_i = A_i B_i + (A_i + B_i)C_{i-1}$
$= G_i + P_i C_{i-1}$

Example (in class):
A = 0001
B = 1111
C_{out} = 1

Carry-Lookahead Adder /3

- **Operational Semantics Continued:**
  - Step 1. Compute G_i and P_i for all columns
  - Step 2. Compute G and P for k-bit blocks
  - Step 3. C_{in} propagates through each k-bit P/G block
- **Example: 4-bit blocks ($G_{3:0}$ and $P_{3:0}$)**
  - $G_{3:0} = G_3 + P_3 (G_2 + P_2 (G_1 + P_1 G_0)$
  - $P_{3:0} = P_3 P_2 P_1 P_0$
- **Generally, as shown on the previous slide:**
  - $G_{i:j} = G_i + P_i (G_{i-1} + P_{i-1} (G_{i-2} + P_{i-2} G_j)$
  - $P_{i:j} = P_i P_{i-1} P_{i-2} P_j$
  - $C_i = G_{i:j} + P_{i:j} C_{i-1}$

Carry-Lookahead Adder /4

- **Carry-Lookahead Adder Delay**
  - For N-bit CLA with k-bit blocks:

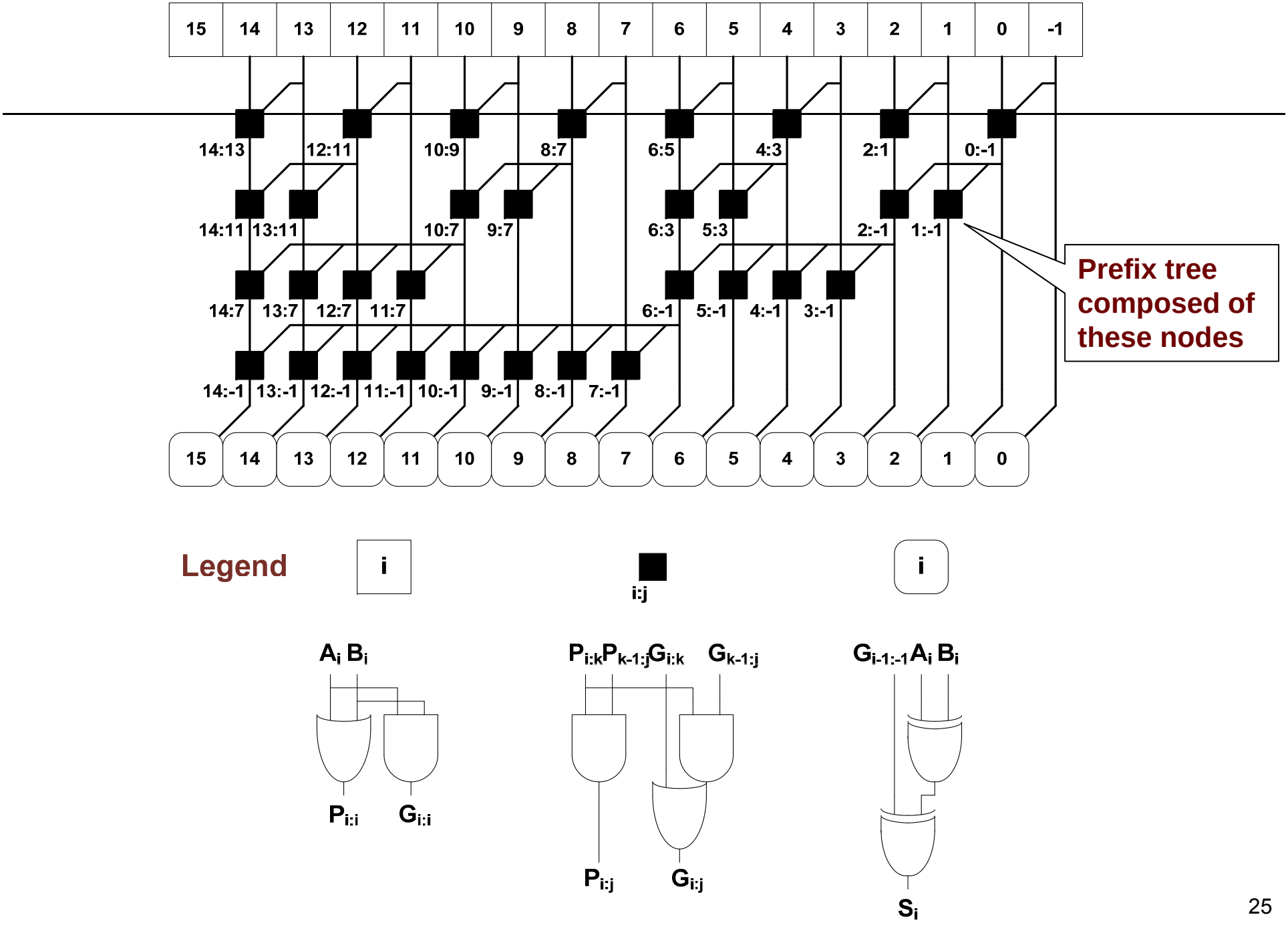
    $$\mathbf{t_{CLA} = t_{pg} + t_{pg\_block} + (N/k - 1) \times t_{AND\_OR} + k \times t_{FA}}$$

    - where t_{pg} is the delay needed to generate all P_i, G_i
    - t_{pg_block} is the delay needed to generate all $P_{i:j}$, $G_{i:j}$
    - t_{AND_OR} is the delay from C_{in} to C_{out} of final AND/OR gate in k-bit CLA block

  - **An N-bit CLA is generally much faster than a ripple-carry adder for N > 16**

Prefix Adder /1

- **Prefix Adder**
  - Continues on the idea of carry-lookahead adder
  - Computes carry-in (Ci-1) for each column then computes the sum $S_i = (A_i \oplus B_i) \oplus C_i$
  - Computes G and P for 1-, 2-, 4-, 8-, etc bit blocks until all G_i (carry-in) are known **(computed in $\log_2 N$ stages)**
- **Operational Semantics Overview:**
  - Carry-in either generated in a column or propagated from a previous column
  - Column -1 holds C_{in}, so **$G_{-1} = C_{in}$ and $P_{-1} = 0$**
  - Carry-in to column i equals carry-out of column i-1: **$C_{i-1} = G_{i-1:-1}$**
  - $G_{i-1:-1}$ is the generate signal spanning columns i-1 to -1
  - Sum equation: $S_i = (A_i \oplus B_i) \oplus G_{i-1:-1}$
  - Goal: Quickly compute $G_{0:-1}$, $G_{1:-1}$, $G_{2:-1}$, $G_{3:-1}$, $G_{4:-1}$, $G_{5:-1}$, …
    - **These are called prefixes**


15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 -1
14:13 12:11 10:9 8:7 6:5 4:3 2:1 0:-1
14:11 13:11 10:7 9:7 6:3 5:3 2:-1 1:-1
14:7 13:7 12:7 11:7 6:-1 5:-1 4:-1 3:-1
14:-1 13:-1 12:-1 11:-1 10:-1 9:-1 8:-1 7:-1
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Prefix tree composed of these nodes
Legend
i
i:j
i
A_i B_i
$P_{i:i}$ $G_{i:i}$
$P_{i:k}P_{k-1:j}G_{i:k}$ $G_{k-1:j}$
$P_{i:j}$ $G_{i:j}$
$G_{i-1:-1}A_i$ B_i
S_i

Prefix Adder /3

- **Operational Semantics Continued:**
  - Generate and propagate signals for a block spanning bits i:j:

    $G_{i:j} = G_{i:k} + P_{i:k} G_{k-1:j}$

    $P_{i:j} = P_{i:k} P_{k-1:j}$
- **Generate Stage: block i:j will generate a carry if**
  - Upper part (i:k) generates a carry, or
  - Upper part propagates a carry generated in lower part (k-1:j)
- **Propagate Stage: block i:j will propagate a carry if**
  - Both the upper and lower parts propagate the carry
- See Section 5.2.1 of the Harris textbook for details

Prefix Adder /4

- **Prefix Adder Delay**
  - For N-bit prefix adder (PA):

    $\mathbf{t_{PA} = t_{pg} + \log_2 N \times (t_{pg\_prefix}) + t_{XOR}}$

    - t_{pg} is the delay to produce P_i G_i (OR and AND gate)
    - t_{pg_prefix} is the delay of black prefix cell (AND-OR gate)

  - The delay grows logarithmically instead of linearly
  - **An N-bit PA is generally faster than a CLA for N ≥ 32**

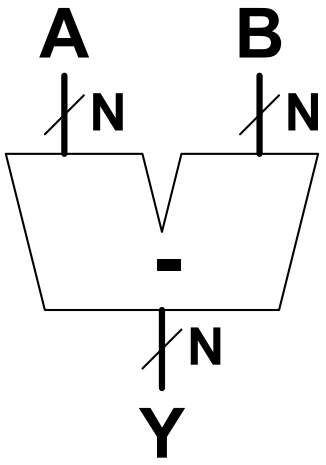
Adder Delay Comparisons

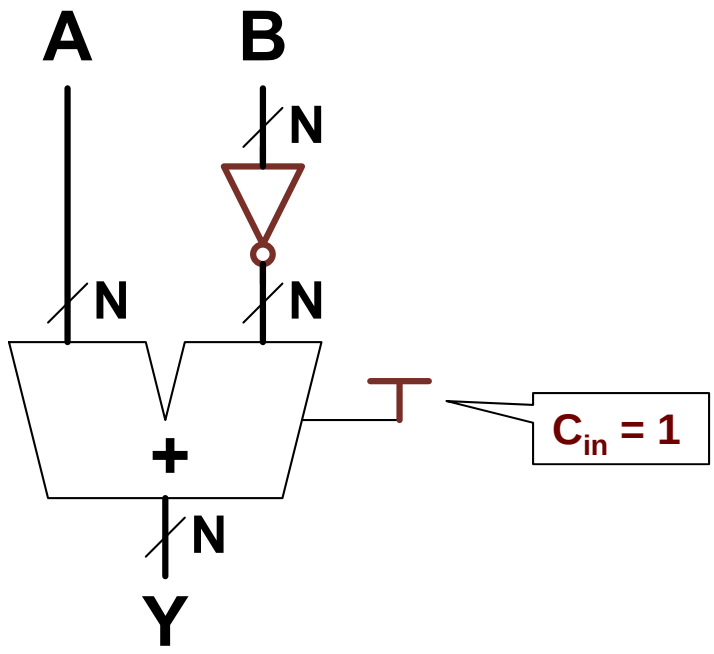
- **Let us compare the delay of 32-bit ripple-carry, carry-lookahead, and prefix adders**
  - CLA has 4-bit blocks
  - 2-input gate delay = 100 ps; full adder delay = 300 ps
  - t_{ripple} = N x t_{FA} = 32(300 ps) **= 9.6 ns**
  - t_{CLA} = t_{pg} + t_{pg_block} + (N/k – 1) x t_{AND_OR} + k x t_{FA}
    = [100 + 600 + (7)200 + 4(300)] ps **= 3.3 ns**
  - t_{PA} = t_{pg} + $\log_2$N x (t_{pg_prefix}) + t_{XOR}
    = [100 + $\log_2$32(200) + 100] ps **= 1.2 ns**

Subtractor

- **Subtraction:**
  - Subtract A from B by first changing the sign of B using an inverter (i.e., complement B's bits and then add 1)
  - Then add A and the inverted B

Symbol **Implementation**

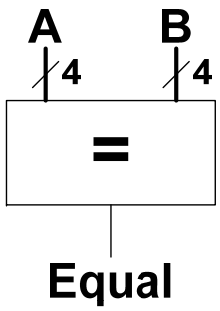




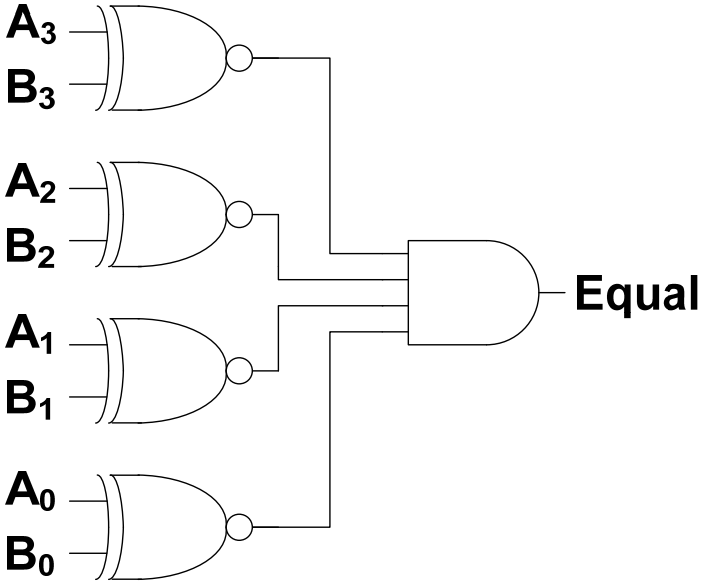
Equality Comparator

- **Comparator:**
  - Determines if two N-bit binary numbers, A and B, are equal, or if one number is greater than or less than the other number
- **Equality Comparator:**
  - Produces a single output that indicates if A is equal to B

Symbol

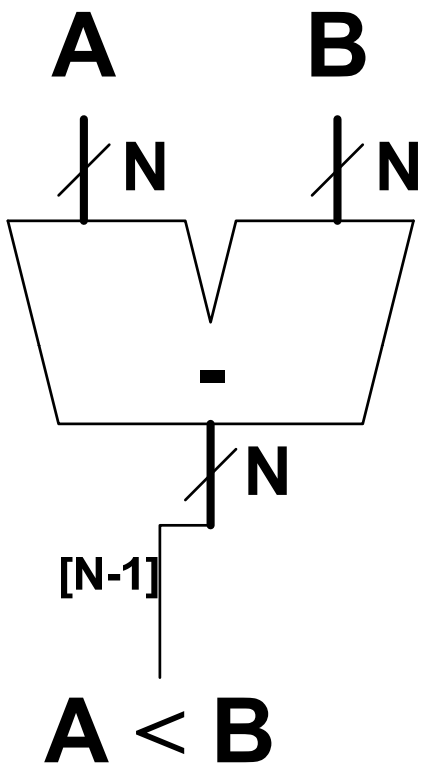


Implementation



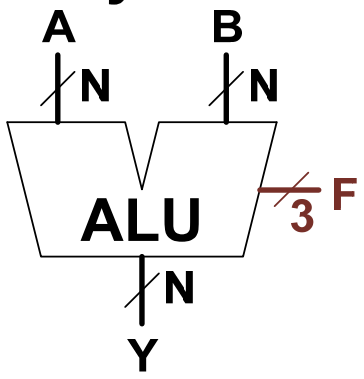
Magnitude Comparator

- **Magnitude Comparator:**
  - Produces one or more outputs indicating the relative values of A and B
  - First compute A – B and then look at the MSB of the result
  - If the MSB equals 1 (i.e., the result is negative), A is less than B; otherwise, A is greater than or equal to B



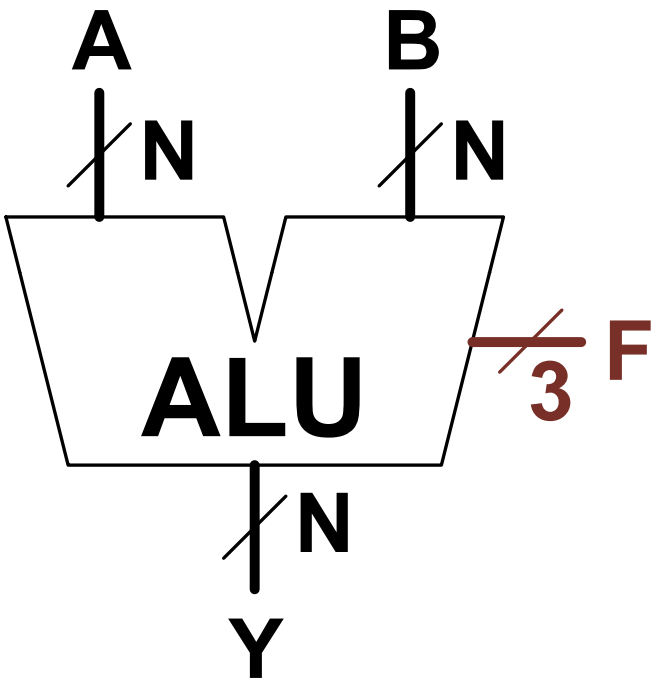
Introducing Arithmetic Logic Unit (ALU) /1

- **Arithmetic/Logical Unit (ALU):**
  - Combines mathematical and logical operations
  - A typical ALU performs AND, OR, addition, subtraction, and magnitude comparison operations
  - **The ALU is at the core of most computer systems**
  - Certain ALUs produce extra outputs, called flags, that indicate information about the ALU output
  - For example, overflow flag indicates that the result of the adder overflowed; zero flag indicates that the result is zero
- **The symbol for ALU:**



where F is the selector bit that selects the corresponding operation

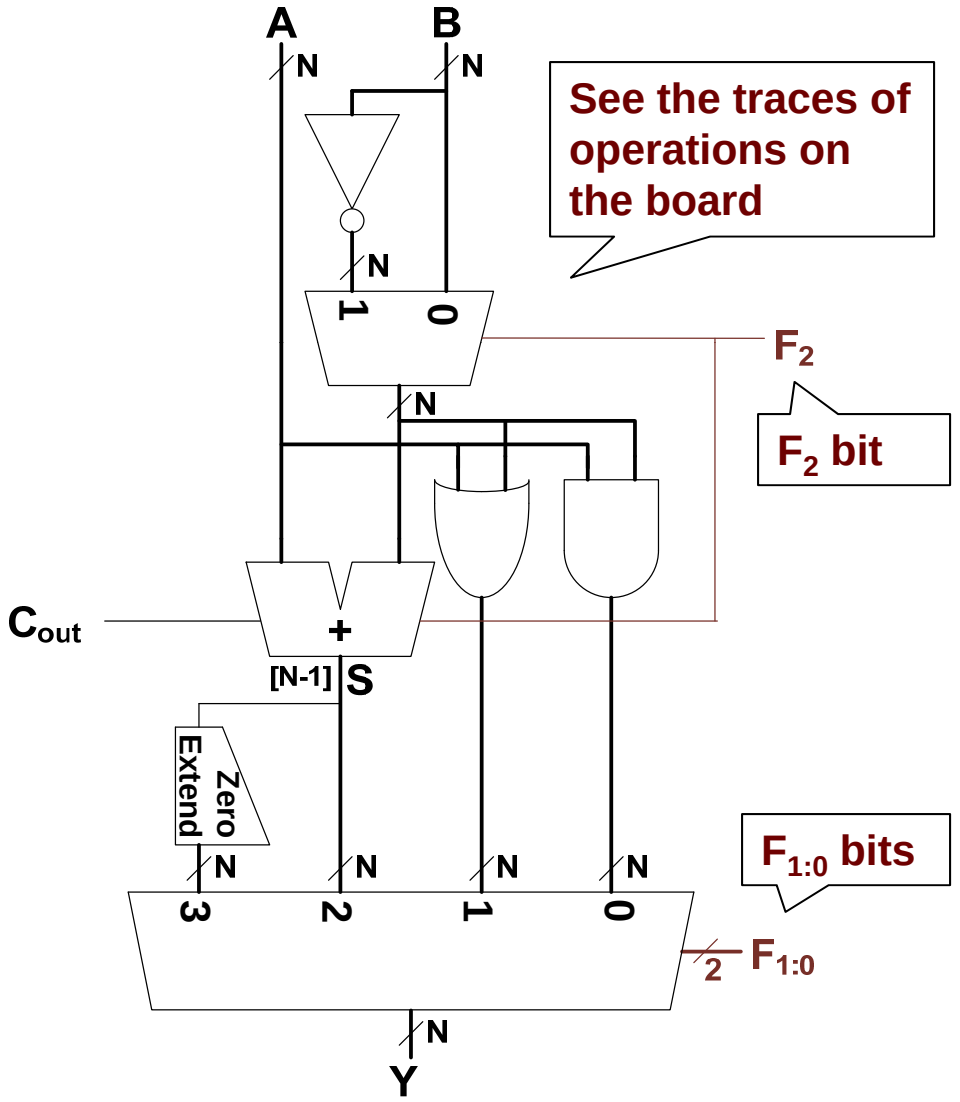
Introducing Arithmetic Logic Unit (ALU) /2



$F_{2:0}$	Function
000	A & B
001	A | B
010	A + B
011	not used
100	A & ~B
101	A | ~B
110	A - B
111	SLT

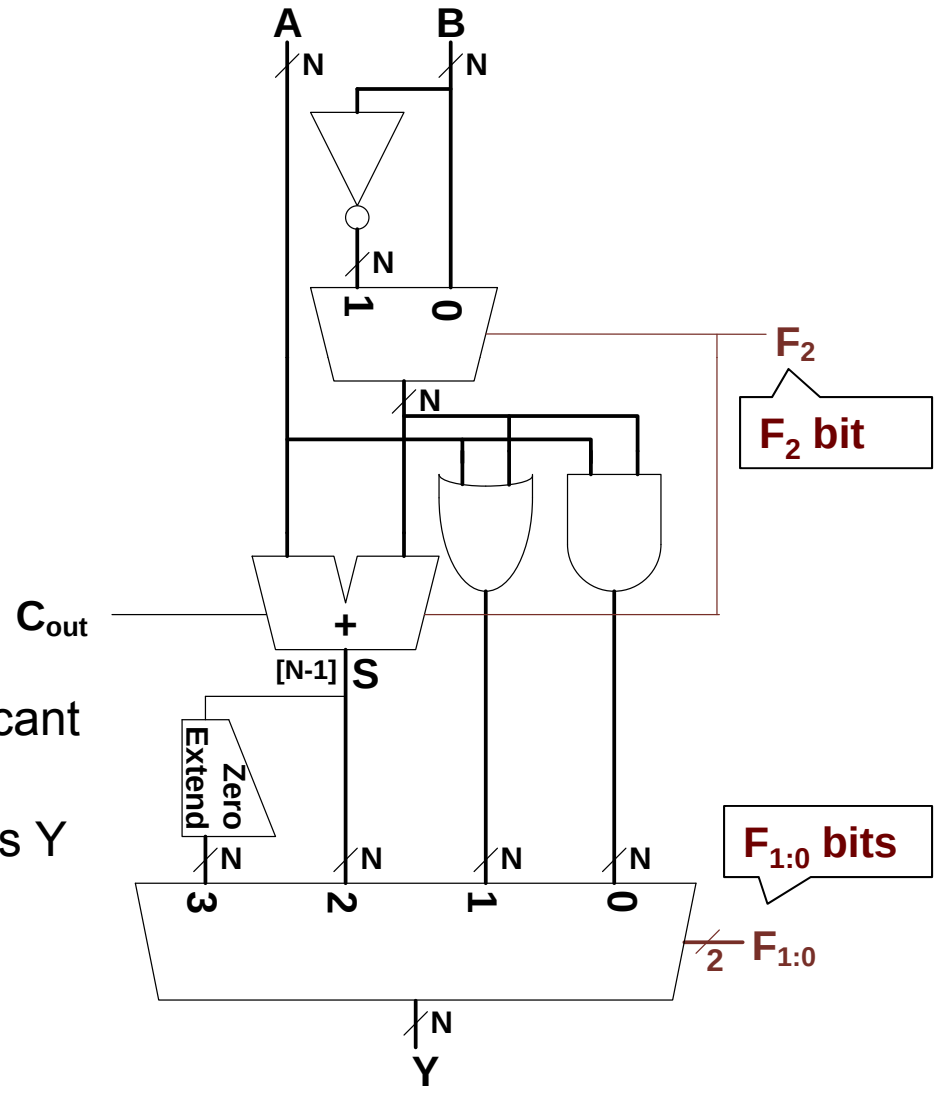
Introducing Arithmetic Logic Unit (ALU) /3

$F_{2:0}$	Function
000	A & B
001	A | B
010	A + B
011	not used
100	A & ~B
101	A | ~B
110	A - B
111	SLT



Introducing Arithmetic Logic Unit (ALU) /4

- **Configure 32-bit ALU for SLT operation:**
  - A = 25 and B = 32
  - A < B, so Y should be 32-bit representation of 1 (0x00000001)
  - $F_{2:0}$ = 111
    - F_2 = 1 (adder acts as subtracter), so 25 - 32 = -7
    - -7 has 1 in the most significant bit (S_{31} = 1)
    - $F_{1:0}$ = 11 multiplexer selects Y = S_{31} (zero extender) = 0x00000001



Shifters and Rotators /1

- **Shifters and Rotators:**
  - Move bits and multiply or divide by powers of 2.
  - **Shifter:** Circuit that shifts a binary number left or right by a specified number of bit positions
  - **Rotator:** Rotates a binary number in a circle pattern, so that empty spots are filled with bits shifted from the other end
  - **Logical Shifter:** Shifts a binary number to the left (`sll`) or right (`srl`), and fills the empty bits with 0s
  - **Arithmetic Shifter:** Performs the same as the logical shifter on left (`sla` and `sll` are the same); on right, it fill the bits with a copy of the MSB (`sra` is different than `srl`)

Shifters and Rotators /2

- **Shifters and Rotators Examples:**
  - Logical shifter:

    11001 >> 2 = 00110

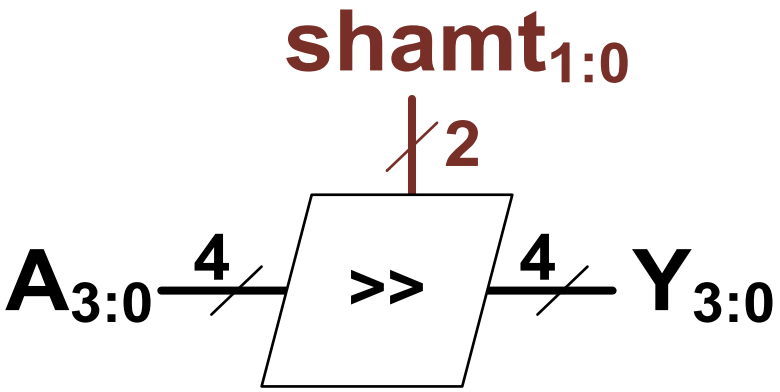
    11001 << 2 = 00100
  - Arithmetic shifter:

    11001 >>> 2 = 11110

    11001 <<< 2 = 00100
  - Rotator:

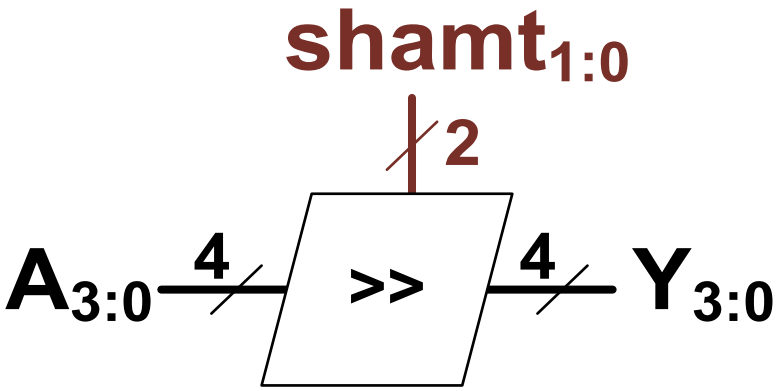
    11001 ROR 2 = 01110

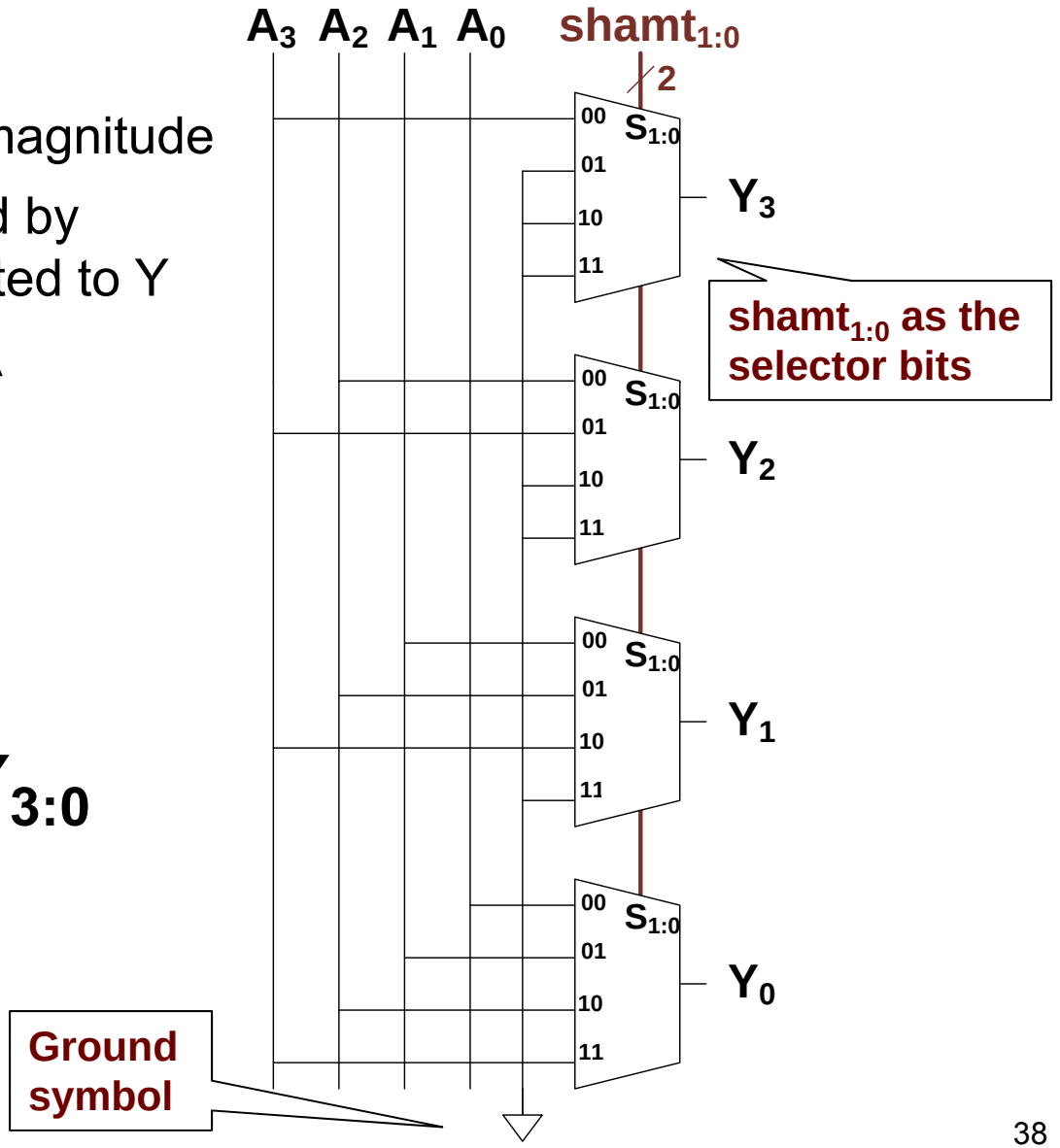
    11001 ROL 2 = 00111



Shifters and Rotators /3

- **$shamt_{1:0}$:**
  - Represents the shift magnitude
  - The input, A, is shifted by $shamt_{1:0}$ bits, and routed to Y
  - If $shamt_{1:0}$ = 00, Y = A





Shifters and Rotators /4

- **Shifters and Rotators Applied:**
  - $A << N = A \times 2^N$

    $00001 << 2 = 00100 \quad (1 \times 2^2 = 4)$

    $11101 << 2 = 10100 \quad (-3 \times 2^2 = -12)$
  - $A >>> N = A / 2^N$

    $01000 >>> 2 = 00010 \quad (8 / 2^2 = 2)$

    $10000 >>> 2 = 11100 \quad (-16 / 2^2 = -4)$
  - Trace these through the matching shifters on the next slide

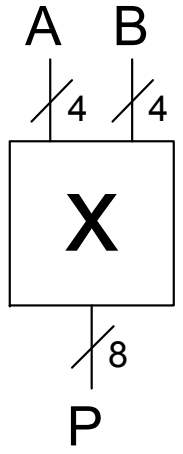
Multipliers /1

- **Multipliers:**
  - N-by-N multipliers multiply two N-bit numbers, and produce 2N-bit results
  - **The partial products in binary multiplication are either the multiplicand or all 0s**
  - Multiplication of 1-bit binary numbers is equivalent to the AND operation, so AND gates are used to form the partial products

Decimal		Binary
230	multiplicand	0101
x 42	multiplier	x 0111
460	partial products	0101
+ 920		0101
9660		0101
		+ 0000
	result	0100011

Multiplying a single digit of the multiplier with multiplicand in stages

Shifted partial products are then summed to form the result



230 x 42 = 9660

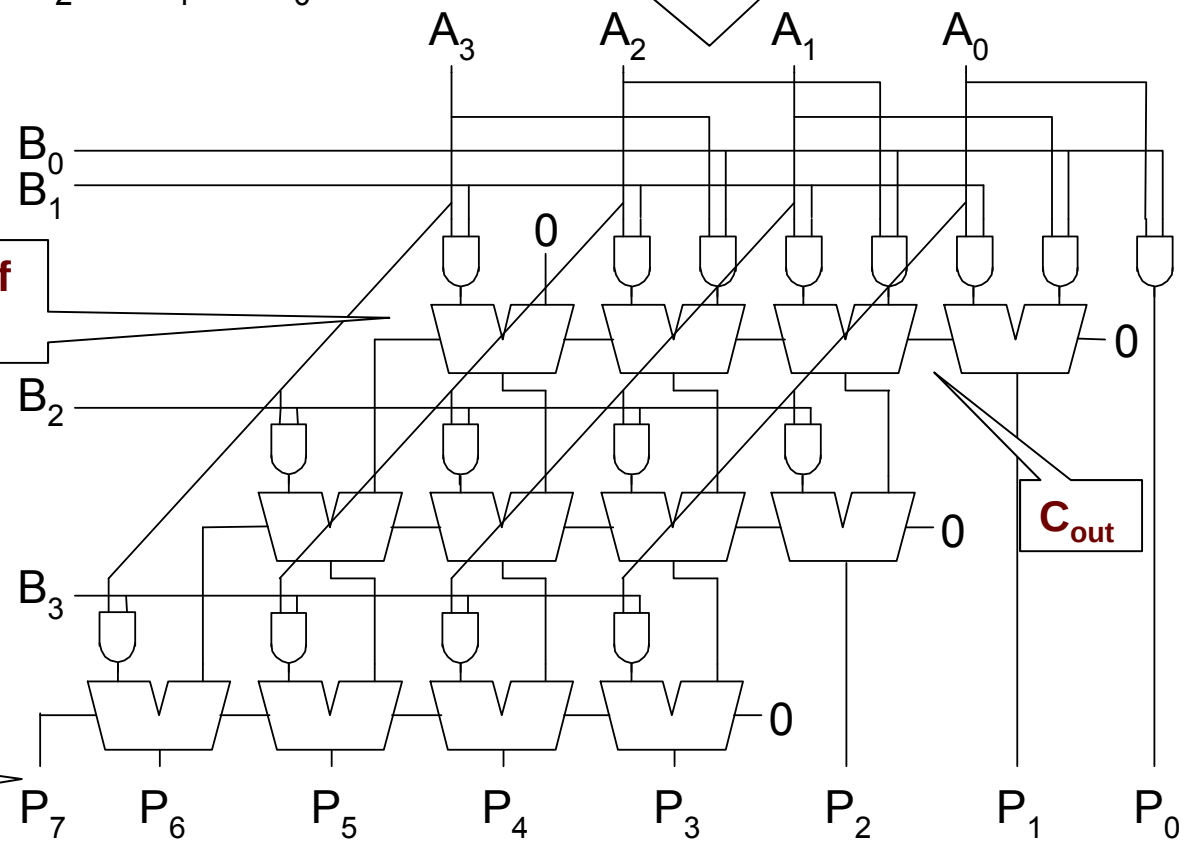
5 x 7 = 35

$$
\begin{array}{rcccccccc}
 & & & A_3 & A_2 & A_1 & A_0 & & \\
\times & & & B_3 & B_2 & B_1 & B_0 & & \\
\hline
 & & & A_3B_0 & A_2B_0 & A_1B_0 & A_0B_0 & & \\
 & & A_3B_1 & A_2B_1 & A_1B_1 & A_0B_1 & & & \\
 & A_3B_2 & A_2B_2 & A_1B_2 & A_0B_2 & & & & \\
+ & A_3B_3 & A_2B_3 & A_1B_3 & A_0B_3 & & & & \\
\hline
P_7 & P_6 & P_5 & P_4 & P_3 & P_2 & P_1 & P_0
\end{array}
$$

Trace execution of 0101 x 0101

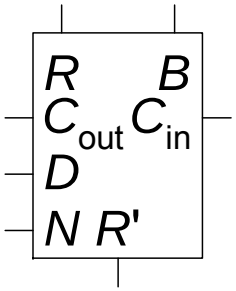
Adders add components of the partial products

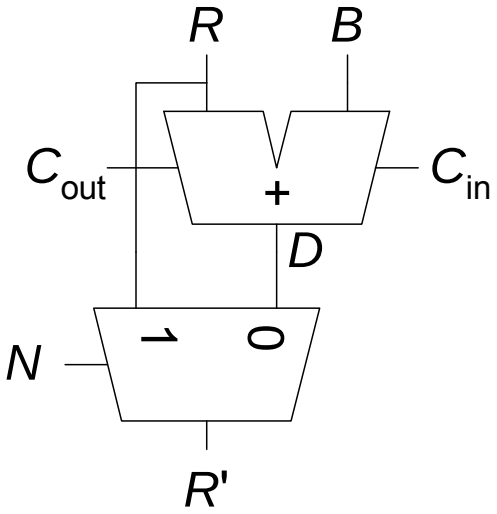
Shifting performed via the output bits

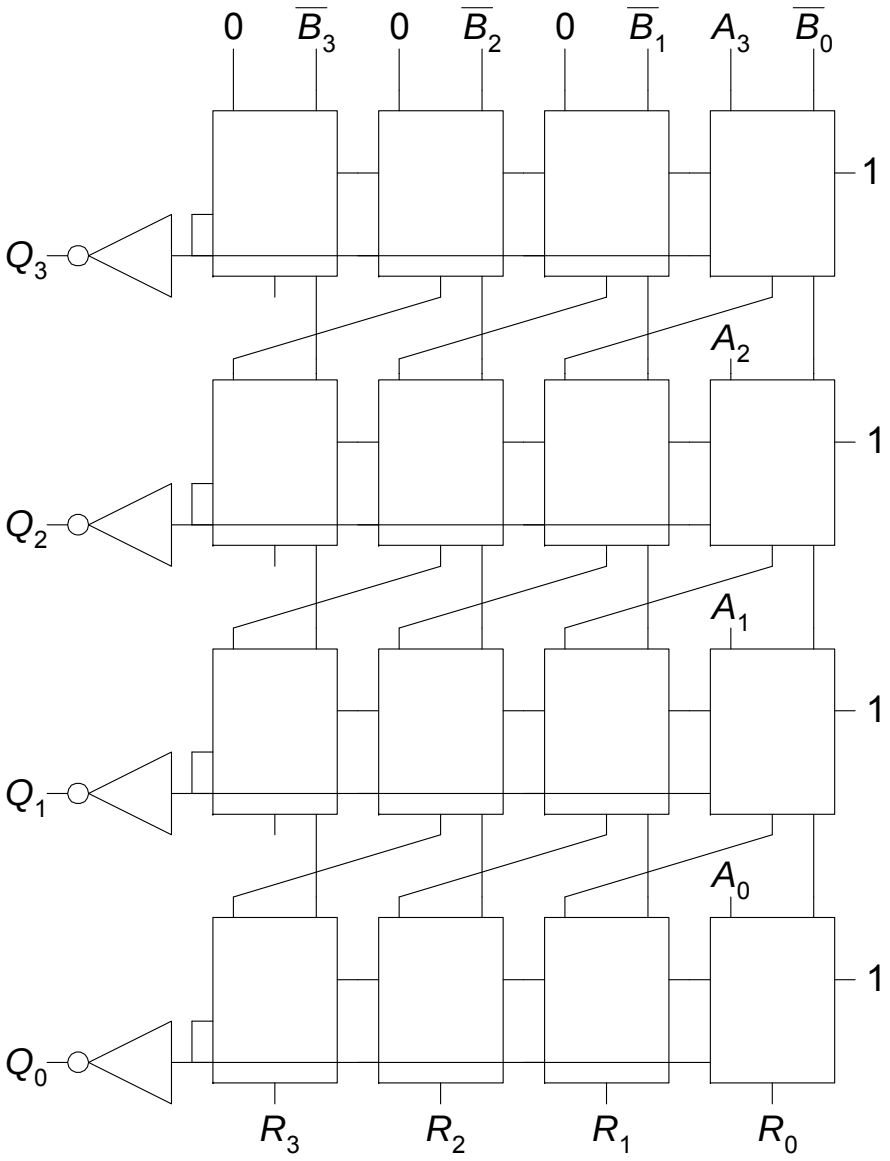


Divider Overview

- **Divider Example:**
  - The divider computes A/B and produces a quotient, Q, and a remainder, R
  - N indicates if R – B is negative, and it is obtained from the C_{out} bit of the left-most block







Food for Thought

- **Download and Read Assignment #2 Specifications**
- **Read:**
  - Chapter 3 of the course textbook
    - Review the material discussed in the lecture notes in more detail
  - (Optional) Chapter 5 of the Harris and Harris textbook