CSE 140 Lecture 13
Standard Combinational Modules

CK Cheng

CSE Dept.

UC San Diego

Some slides from Harris and Harris

Part III. Standard Modules

Interconnect

Operators.

- Representation of numbers
- Adders
  1. Full Adder
  2. Half Adder
  3. Ripple-Carry Adder
  4. Carry Look Ahead Adder
  5. Prefix Adder
- ALU
- Comparator
- Shifter
- Multiplier
- Division

Design Flow

- Specification: Data Representations
- Arithmetic: Algorithms
- Logic: Synthesis
- Layout: Placement and Routing

1. Representation of numbers

Negative Numbers

- 2’s complement of n-bit vector

  -x: 2^n-x

- 1’s complement of n-bit vector

  -x: 2^n-x-1

1. Representation

- 2’s Complement

  -x: 2^n-x

  e.g. 16-x

- 1’s Complement

  -x: 2^n-x-1

  e.g. 16-x-1

Id	2’s comp.	1’s comp.
0	0	15
-1	15	14
-2	14	13
-3	13	12
-4	12	11
-5	11	10
-6	10	9
-7	9	8
-8	8	

1. Representation

Id	-Binary	sign mag	2’s comp	1’s comp
0	0000	1000	0000	1111
-1	0001	1001	1111	1110
-2	0010	1010	1110	1101
-3	0011	1011	1101	1100
-4	0100	1100	1100	1011
-5	0101	1101	1011	1010
-6	0110	1110	1010	1001
-7	0111	1111	1001	1000
-8			1000	

Representation

1’s Complement

For a negative number, we take the positive number and complement every bit.

2’s Complement

For a negative number, we do 1’s complement and plus one.

$(b_{n-1}, b_{n-2}, \ldots, b_0)$: $-b_{n-1}2^{n-1} + \text{sum}_{i<n-1}\ b_i 2^i$

b_{n-1}=1 iff the number is negative

Representation

2's Complement

- $x+y$
- $x-y$: $x+2^n-y= 2^n+x-y$
- $-x+y$: 2^n-x+y
- $-x-y$: 2^n-x+2^n-y

  $= 2^n+2^n-x-y$
- $-(-x)=2^n-(2^n-x)=x$

1's Complement

- $x+y$
- $x-y$: $x+2^n-y-1= 2^n-1+x-y$
- $-x+y$: $2^n-x-1+y=2^n-1-x+y$
- $-x-y$: $2^n-x-1+2^n-y-1$

  $= 2^n-1+2^n-x-y-1$
- $-(-x)=2^n-(2^n-x-1)\ -1=x$

Examples

2 + 3 = 5

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

2 - 3 = -1 (2's)

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

2 - 3 = -1 (1's)

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

-2 - 3 = -5 (2's)

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

-2 - 3 = -5 (1's)

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

Check for overflow (2's)

3 + 5 = 8

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

$C_4 C_3$

-3 + -5 = -8

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

$C_4 C_3$

Addition: 2’s Complement Overflow

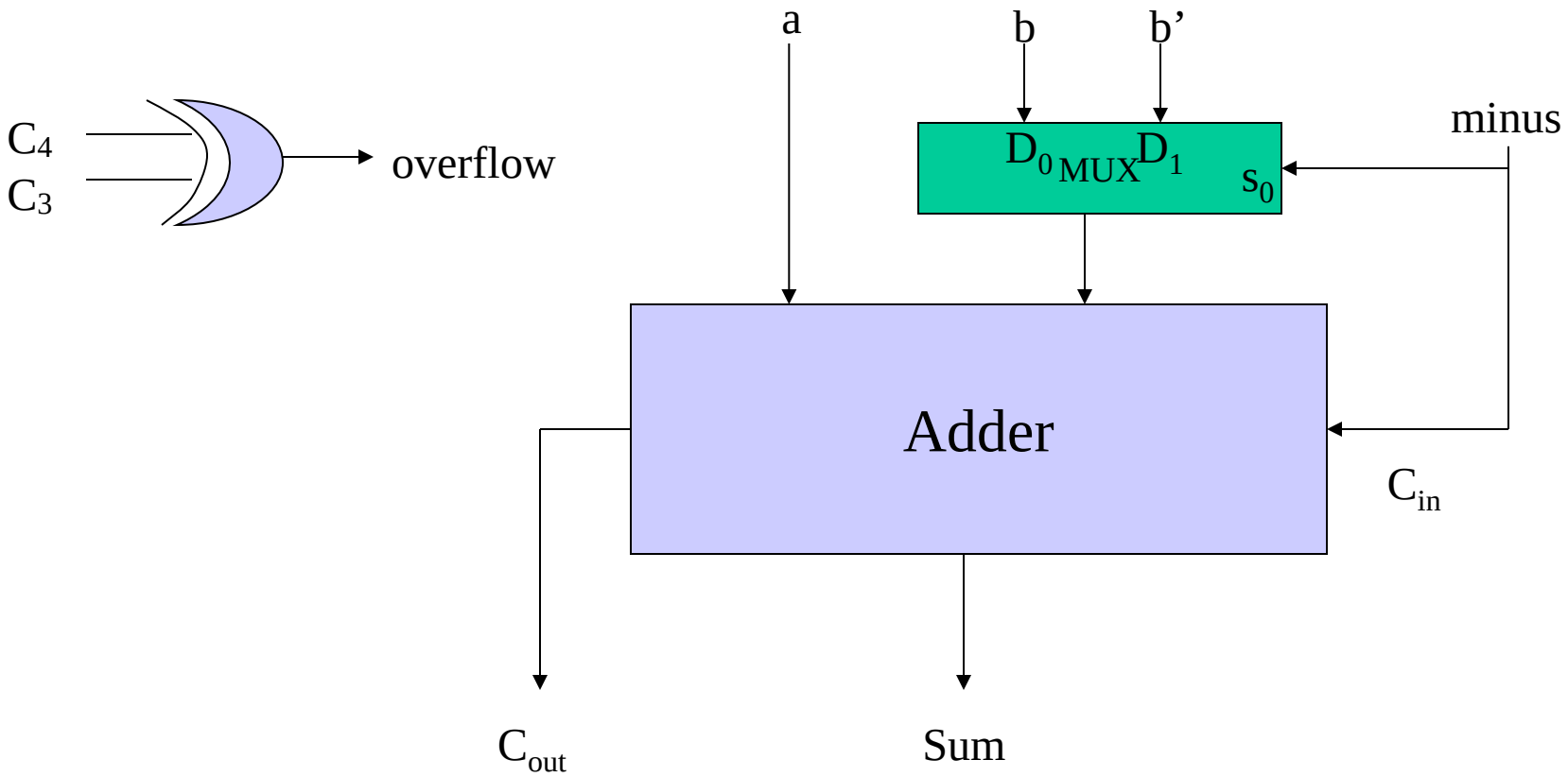
In 2’s complement:

$$\text{overflow} = c_n \oplus c_{n-1}$$

Exercise:

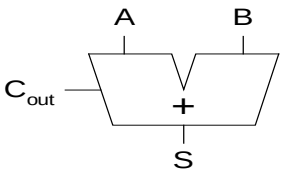
1. Demonstrate the overflow with more examples.
2. Prove the condition.

Addition and Subtraction using 2's Complement



1-Bit Adders

Half Adder

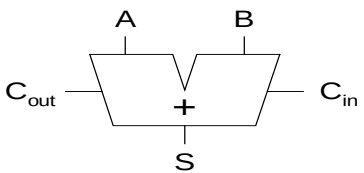


A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$S = A \oplus B$

$C_{out} = AB$

Full Adder

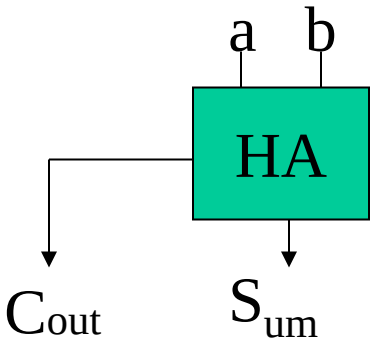


C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

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

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

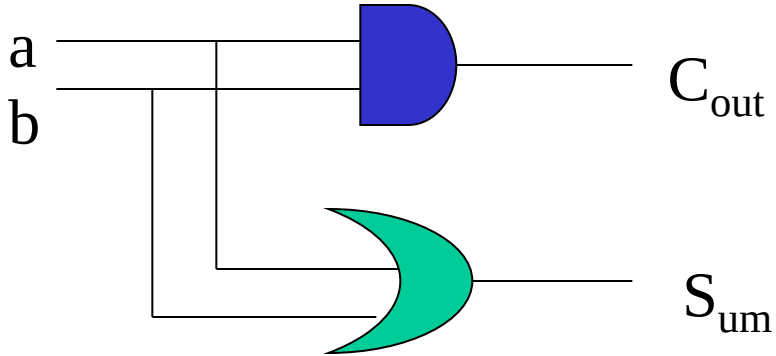
Half Adder



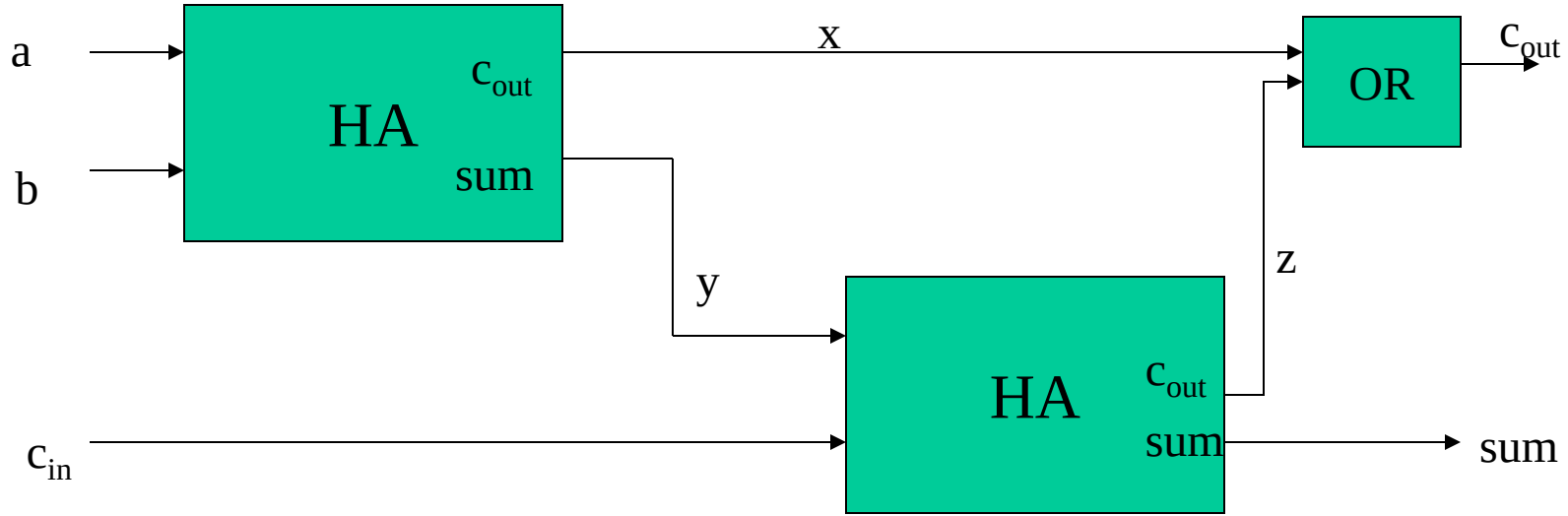
a	b	C_{out}	S_{um}
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$S_{um} = ab' + a'b = a \oplus b$

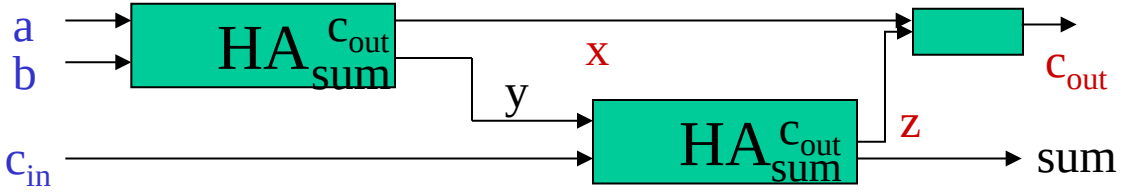
$C_{out} = ab$



Full Adder Composed of Half Adders



Full Adder Composed of Half Adders

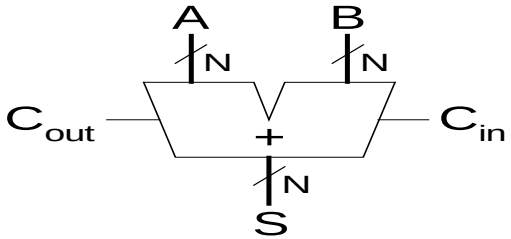


Id	a	b	c_{in}	x	y	z	c_{out}	s_{um}
0	0	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0	1
2	0	1	0	0	1	0	0	1
3	0	1	1	0	1	1	1	0
4	1	0	0	0	1	0	0	1
5	1	0	1	0	1	1	1	0
6	1	1	0	1	0	0	1	0
7	1	1	1	1	0	0	1	1

Id	x	z	c_{out}
0	0	0	0
1	0	1	1
2	1	0	1
3	1	1	-

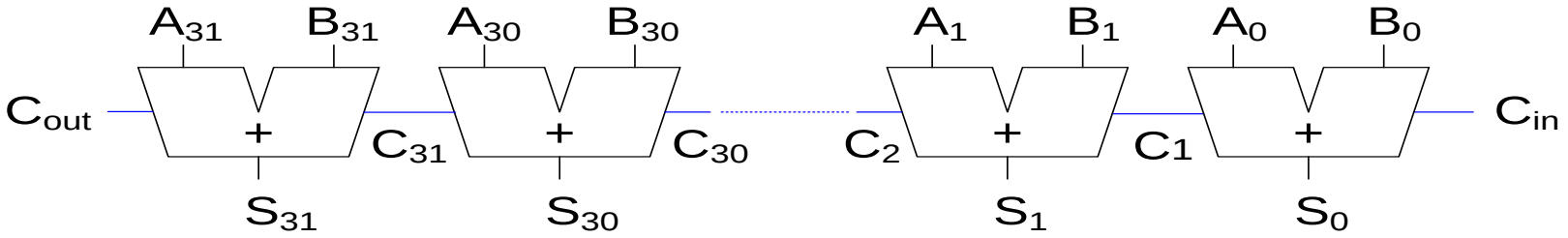
Adder

- Several types of carry propagate adders (CPAs) are:
  - Ripple-carry adders (slow)
  - Carry-lookahead adders (fast)
  - Prefix adders (faster)
- Carry-lookahead and prefix adders are faster for large adders but require more hardware.



Ripple-Carry Adder

- Chain 1-bit adders together
- Carry ripples through entire chain
- Disadvantage: slow



Ripple-Carry Adder Delay

- The delay of an *N*-bit ripple-carry adder is:

$$t_{\text{ripple}} = Nt_{FA}$$

where t_{FA} is the delay of a full adder

Carry-Lookahead Adder

- Compress the logic levels of C_{out}
- **Some definitions:**
  - Generate (G_i) and propagate (P_i) signals for each column:
    - A column will generate a carry out if A_i AND B_i are both 1.

$$G_i = A_i B_i$$

    - A column will propagate a carry in to the carry out if A_i OR B_i is 1.

$$P_i = A_i + B_i$$

    - The carry out of a column (C_i) is:

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

Carry Look Ahead Adder

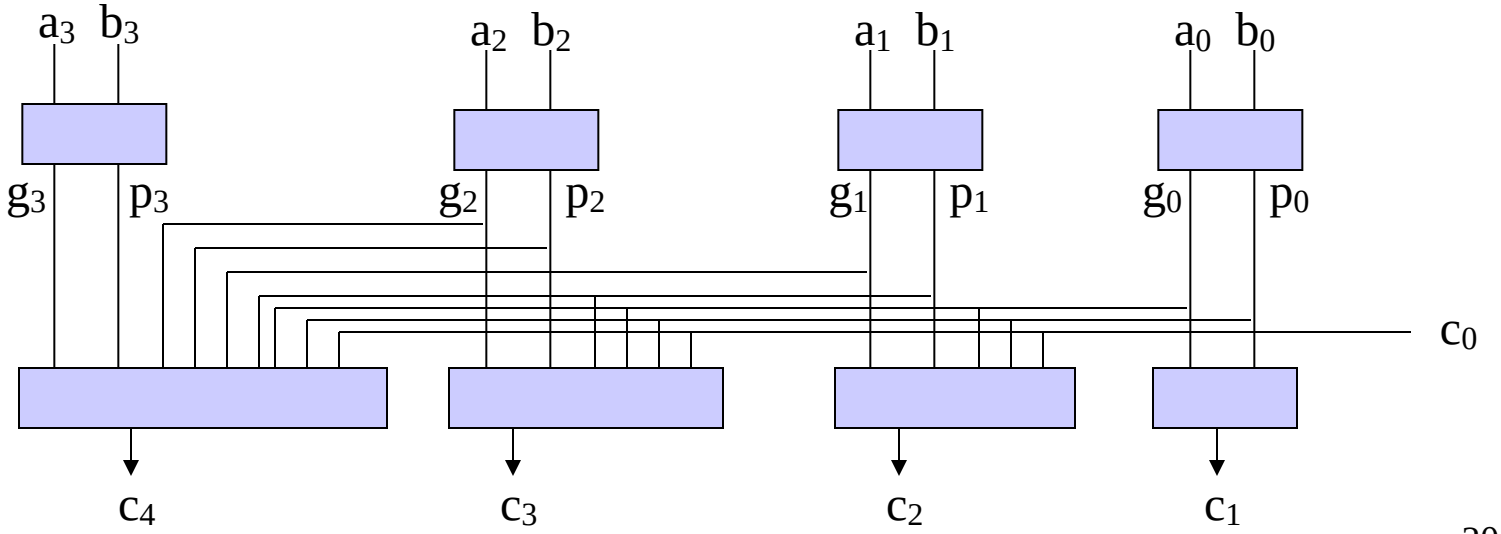
$C_1 = a_0b_0 + (a_0+b_0)c_0 = g_0 + p_0c_0$

$C_2 = a_1b_1 + (a_1+b_1)c_1 = g_1 + p_1c_1 = g_1 + p_1g_0 + p_1p_0c_0$

$C_3 = a_2b_2 + (a_2+b_2)c_2 = g_2 + p_2c_2 = g_2 + p_2g_1 + p_2p_1g_0 + p_2p_1p_0c_0$

$C_4 = a_3b_3 + (a_3+b_3)c_3 = g_3 + p_3c_3 = g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0 + p_3p_2p_1p_0c_0$

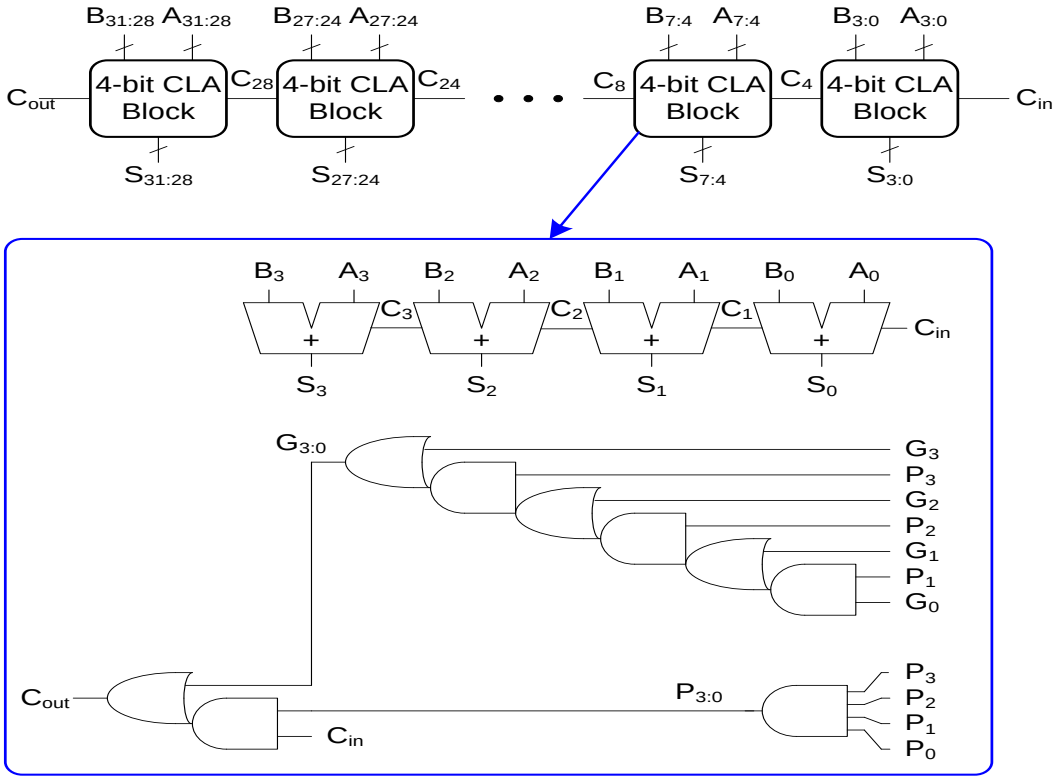
$q_i = a_ib_i \qquad p_i = a_i + b_i$



Carry-Lookahead Addition

- Step 1: compute *generate* (G) and *propagate* (P) signals for columns (single bits)
- Step 2: compute G and P for k-bit blocks
- Step 3: C_{in} propagates through each k-bit propagate/generate block

32-bit CLA with 4-bit blocks



Carry-Lookahead Adder Delay

- Delay of an N-bit carry-lookahead adder with k-bit blocks: $\boldsymbol{t_{CLA} = t_{pg} + t_{pg\_block} + (N/k - 1)t_{AND\_OR} + kt_{FA}}$

where

  - t_{pg} : delay of the column generate and propagate gates
  - t_{pg_block} : delay of the block generate and propagate gates
  - t_{AND_OR} : delay from C_{in} to C_{out} of the final AND/OR gate in the k-bit CLA block

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

Prefix Adder

- Computes the carry in (C_{i-1}) for each of the columns as fast as possible and then computes the sum:

$$S_i = (A_i \oplus B_i) \oplus C_i$$

- Computes G and P for 1-bit, then 2-bit blocks, then 4-bit blocks, then 8-bit blocks, etc. until the carry in (generate signal) is known for each column
- Has $\log_2 N$ stages

Prefix Adder

- A carry in is produced by being either *generated* in a column or *propagated* from a previous column.
- Define column -1 to hold C_{in}, so $\boldsymbol{G_{-1} = C_{in}, P_{-1} = 0}$
- Then, the carry in to col. i = the carry out of col. i-1:

  $$\boldsymbol{C_{i-1} = G_{i-1:-1}}$$

  $G_{i-1:-1}$ is the generate signal spanning columns i-1 to -1.

  There will be a carry out of column i-1 (C_{i-1}) if the block spanning columns i-1 through -1 generates a carry.
- Thus, we rewrite the sum equation: $\boldsymbol{S_i = (A_i \oplus B_i) \oplus G_{i-1:-1}}$
- **Goal:** Compute $G_{0:-1}$, $G_{1:-1}$, $G_{2:-1}$, $G_{3:-1}$, $G_{4:-1}$, $G_{5:-1}$, … (These are called the *prefixes)*

Prefix Adder

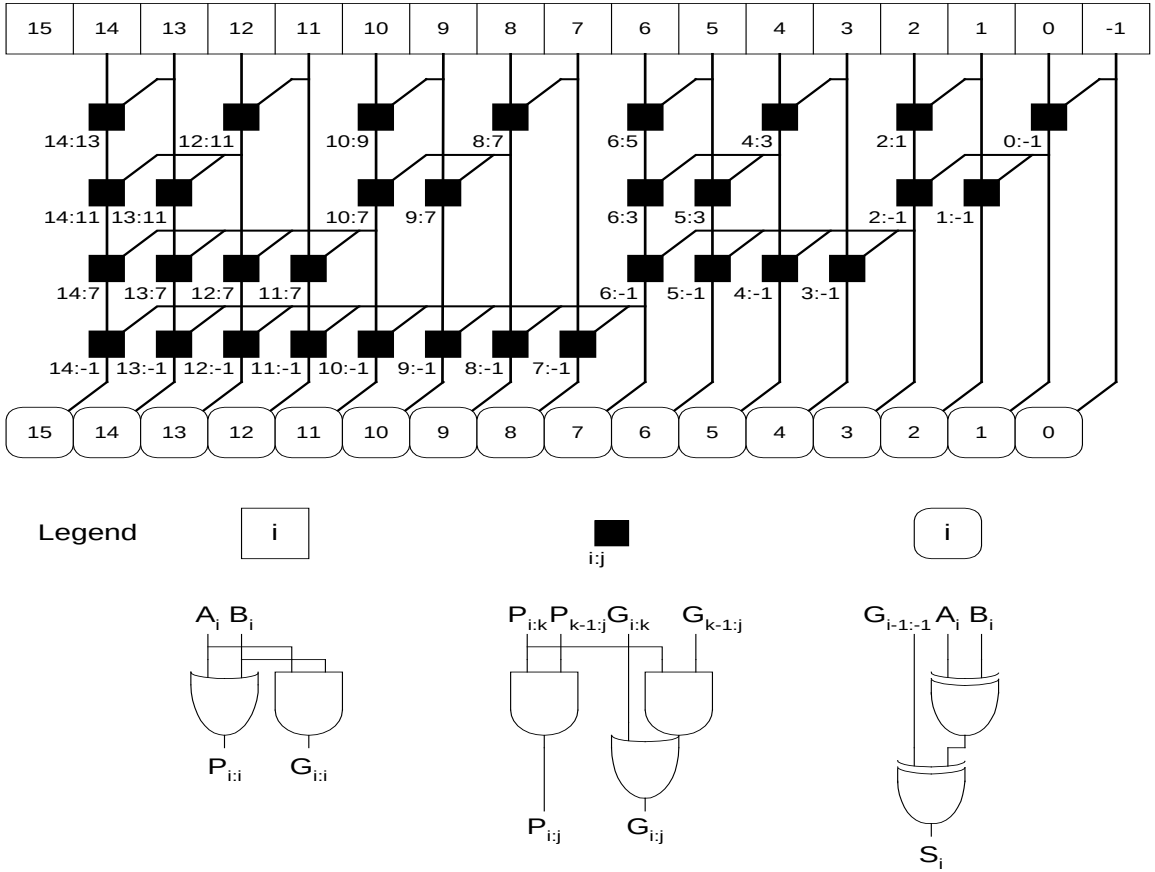
- The generate and propagate signals for a block spanning bits $i{:}j$ are:

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

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

- In words, these prefixes describe that:
  - A block will generate a carry if the upper part ($i{:}k$) generates a carry or if the upper part propagates a carry generated in the lower part (k-1:j).
  - A block will propagate a carry if both the upper and lower parts propagate the carry.

Prefix Adder Schematic



Prefix Adder Delay

- The delay of an *N*-bit prefix adder is:

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

where

  - t_{pg} is the delay of the column generate and propagate gates (AND or OR gate)
  - t_{pg_prefix} is the delay of the black prefix cell (AND-OR gate)

Adder Delay Comparisons

- Compare the delay of 32-bit ripple-carry, carry-lookahead, and prefix adders. The carry-lookahead adder has 4-bit blocks. Assume that each two-input gate delay is 100 ps and the full adder delay is 300 ps.

Adder Delay Comparisons

- Compare the delay of 32-bit ripple-carry, carry-lookahead, and prefix adders. The carry-lookahead adder has 4-bit blocks. Assume that each two-input gate delay is 100 ps and the full adder delay is 300 ps.

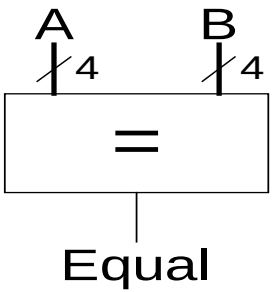
$$t_{\text{ripple}} = Nt_{FA} = 32(300\text{ ps}) = 9.6\text{ ns}$$

$$t_{CLA} = t_{pg} + t_{pg\_\text{block}} + (N/k - 1)t_{\text{AND\_OR}} + kt_{FA}$$
$$= [100 + 600 + (7)200 + 4(300)]\text{ ps} = 3.3\text{ ns}$$

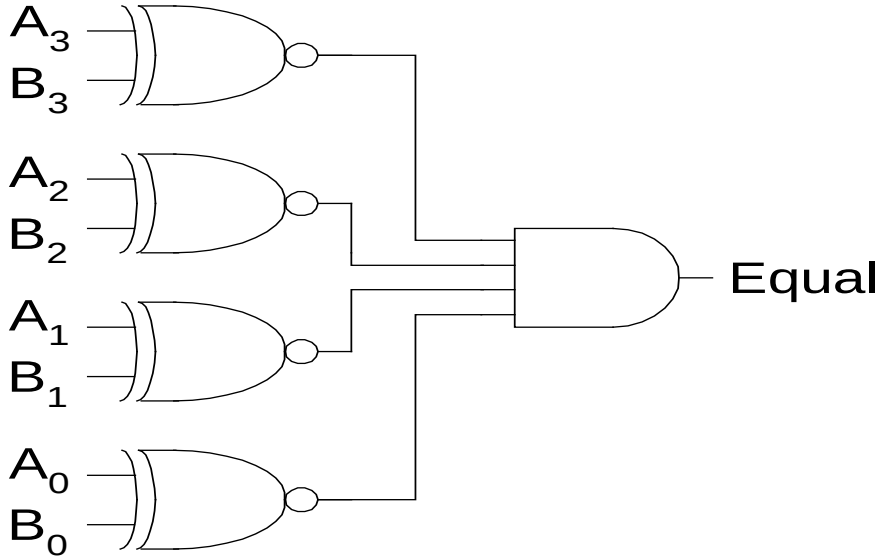
$$t_{PA} = t_{pg} + \log_2 N(t_{pg\_\text{prefix}}) + t_{\text{XOR}}$$
$$= [100 + \log_2 32(200) + 100]\text{ ps} = 1.2\text{ ns}$$

Comparator: Equality

Symbol

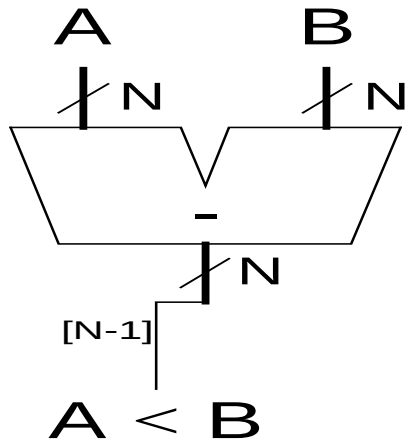


Implementation

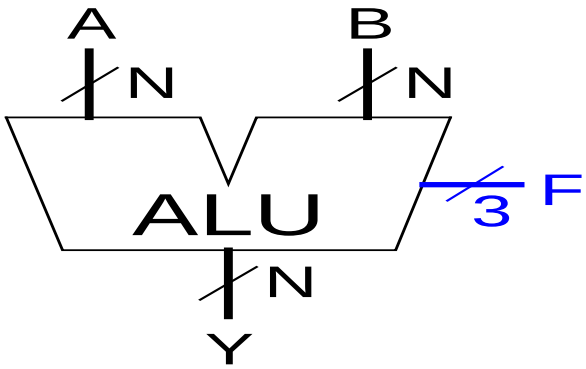


Comparator: Less Than

- Compare two numbers

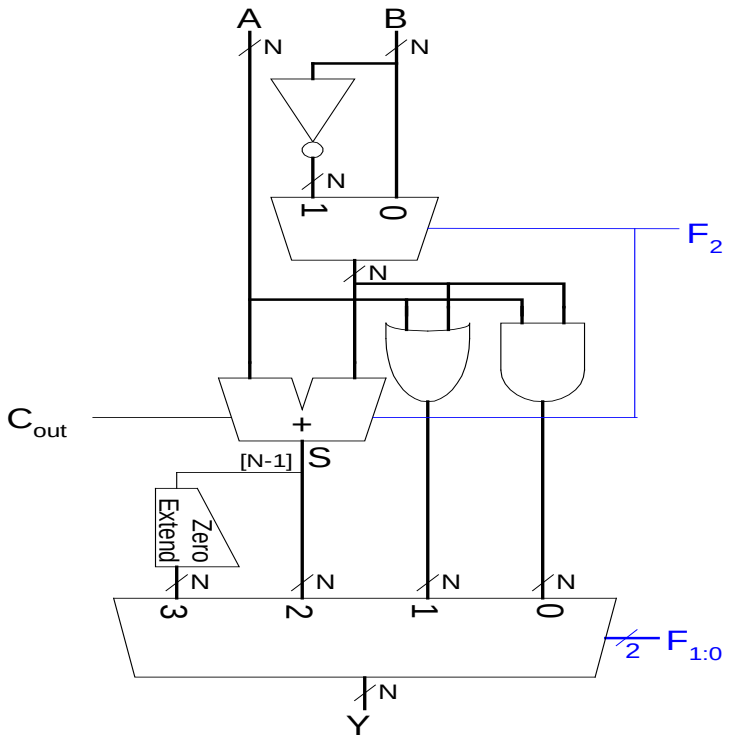


Arithmetic Logic Unit (ALU)



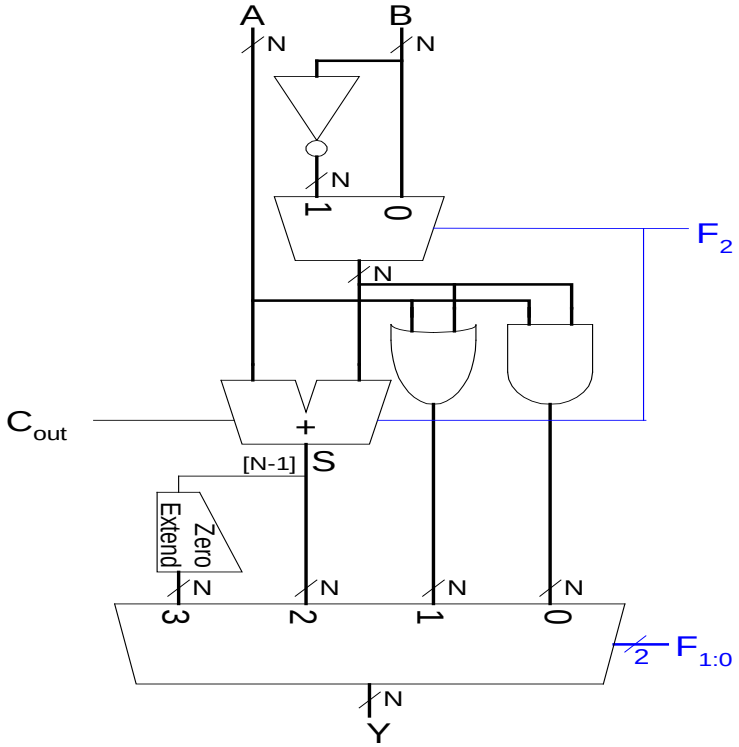
$F_{2:0}$	Function
000	A & B
001	A | B
010	A + B
011	not used
100	A & ~B
101	A | ~B
110	A - B
111	SLT

ALU Design



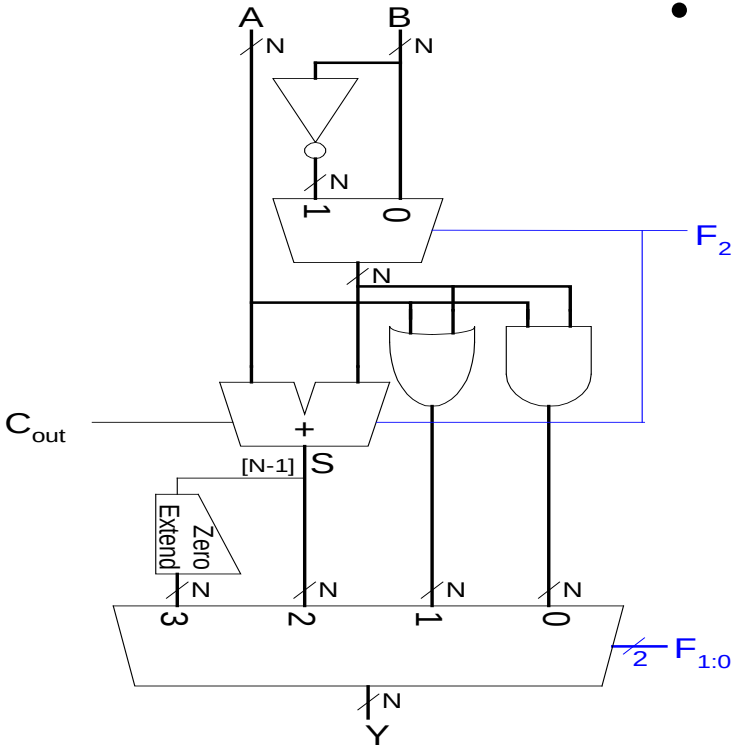
$F_{2:0}$	Function
000	A & B
001	A | B
010	A + B
011	not used
100	A & ~B
101	A | ~B
110	A - B
111	SLT

Set Less Than (SLT) Example



- Configure a 32-bit ALU for the set if less than (SLT) operation. Suppose $A = 25$ and $B = 32$.

Set Less Than (SLT) Example

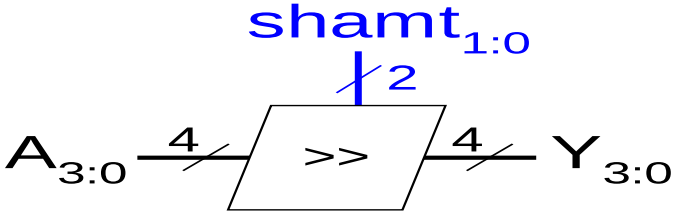


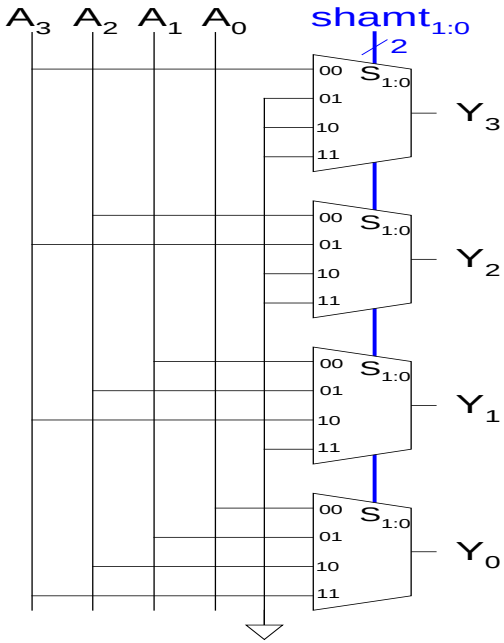
- Configure a 32-bit ALU for the set if less than (SLT) operation. Suppose $A = 25$ and $B = 32$.
  - A is less than B, so we expect Y to be the 32-bit representation of 1 (0x00000001).
  - For SLT, $F_{2:0} = 111$.
  - $F_2 = 1$ configures the adder unit as a subtracter. So 25 - 32 = -7.
  - The two's complement representation of -7 has a 1 in the most significant bit, so $S_{31} = 1$.
  - With $F_{1:0} = 11$, the final multiplexer selects $Y = S_{31}$ (zero extended) = 0x00000001.

Shifters

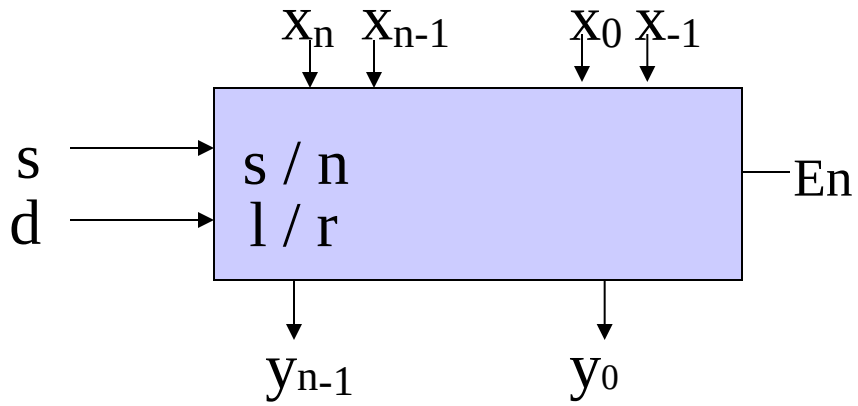
- **Logical shifter:** shifts value to left or right and fills empty spaces with 0's
  - Ex: 11001 >> 2 = 00110
  - Ex: 11001 << 2 = 00100
- **Arithmetic shifter:** same as logical shifter, but on right shift, fills empty spaces with the old most significant bit (msb).
  - Ex: 11001 >>> 2 = 11110
  - Ex: 11001 <<< 2 = 00100
- **Rotator:** rotates bits in a circle, such that bits shifted off one end are shifted into the other end
  - Ex: 11001 ROR 2 = 01110
  - Ex: 11001 ROL 2 = 00111

Shifter Design



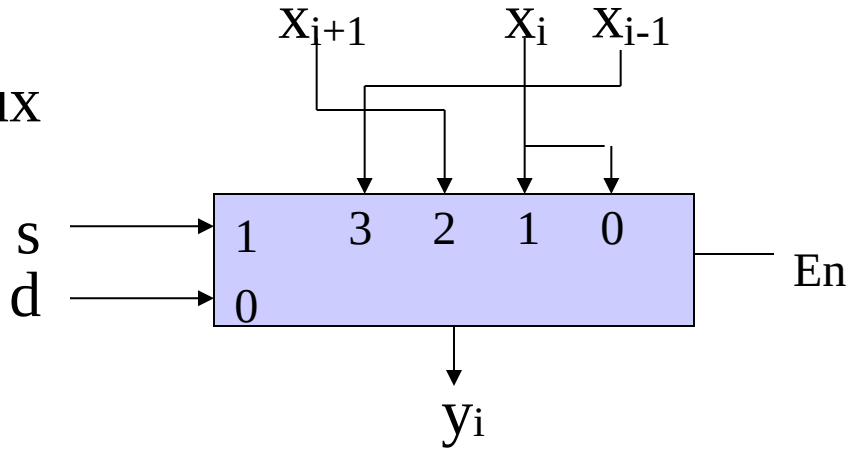


Shifter

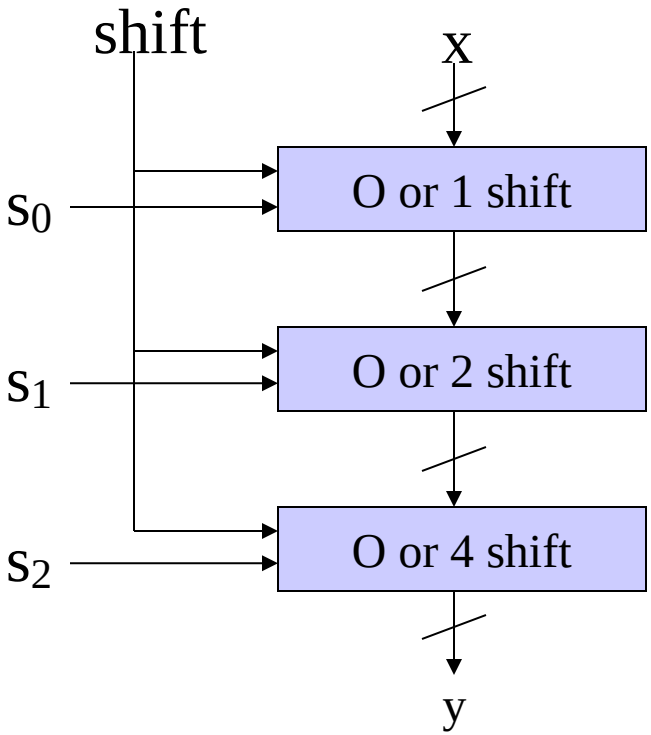


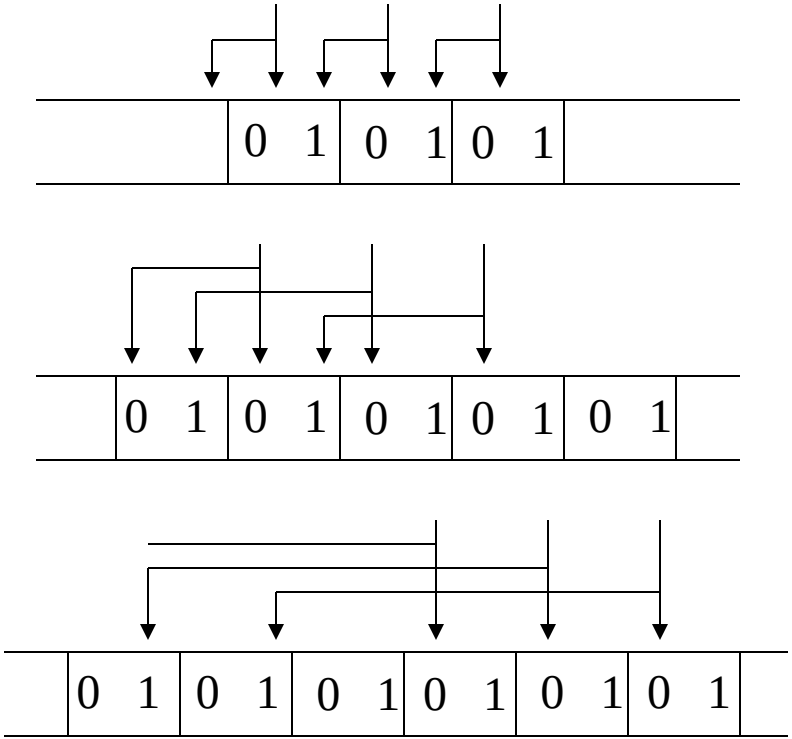
$y_i = x_{i-1}$ if En = 1, s = 1, and d = L
$= x_{i+1}$ if En = 1, s = 1, and d = R
$= x_i$ if En = 1, s = 0
= 0 if En = 0

Can be implemented with a mux



Barrel Shifter





Shifters as Multipliers and Dividers

- A left shift by N bits multiplies a number by 2^N
  - Ex: 00001 << 2 = 00100 $(1 \times 2^2 = 4)$
  - Ex: 11101 << 2 = 10100 $(-3 \times 2^2 = -12)$

- The arithmetic right shift by N divides a number by 2^N
  - Ex: 01000 >>> 2 = 00010 $(8 \div 2^2 = 2)$
  - Ex: 10000 >>> 2 = 11100 $(-16 \div 2^2 = -4)$

Multipliers

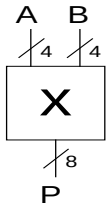
- Steps of multiplication for both decimal and binary numbers:
  - Partial products are formed by multiplying a single digit of the multiplier with the entire multiplicand
  - Shifted partial products are summed to form the result

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

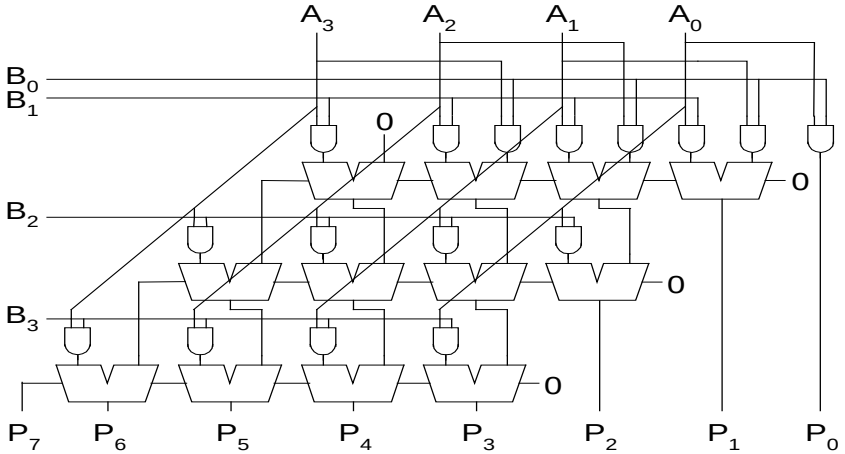
230 x 42 = 9660

5 x 7 = 35

4 x 4 Multiplier



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



Division Algorithm

- $Q = A/B$
- R: remainder
- D: difference

$R = A$

for $i = N$-1 to 0

$D = R - B$

if $D < 0$ then $Q_i = 0$, $R' = R$ // $R < B$

else $Q_i = 1$, $R' = D$ // $R \geq B$

$R = 2R'$

4 x 4 Divider