Combinational Building Blocks

Index

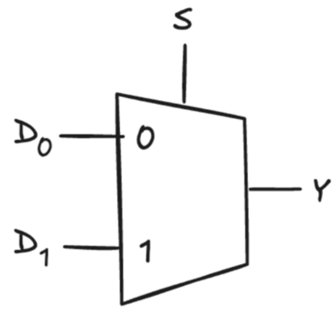
- Building Block definition
- Multiplexer
- Decoders
- Adders
- Subtractor
- Comparators
- Arithmetic/Logical Unit (ALU)
- Shifters and Rotators
- Multiplier and divider
- Floating-Point Addition

Building Block Definition

- Combinational logic can be grouped in **larger building blocks** to build more complex systems
- **Abstraction, hierarchy, modularity, and regularity**
  - **abstract** (hide) gate-level details to **emphasize the function** of the **module** (block)
  - **hierarchically** assembled from simpler components
  - **well-defined interfaces** in order to use them as a black boxes
- Examples:
  - seven-segment display decoders (already considered)
  - multiplexers
  - decoders
  - arithmetic circuits
- We will use many of these building blocks to build a **microprocessor**

Multiplexer (mux)

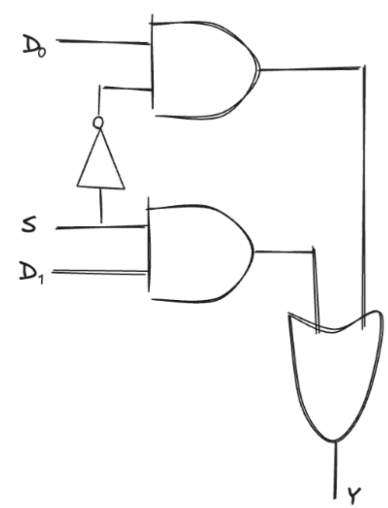
- A device that **choose an output among several possible inputs**, based on the value of a **select signal**
- A 2:1 multiplexer has two data inputs D and one output Y and a select signals S:
  - if S=0, Y=D_0
  - if S=1, Y=D_1



c	D_0	D_1	Y
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

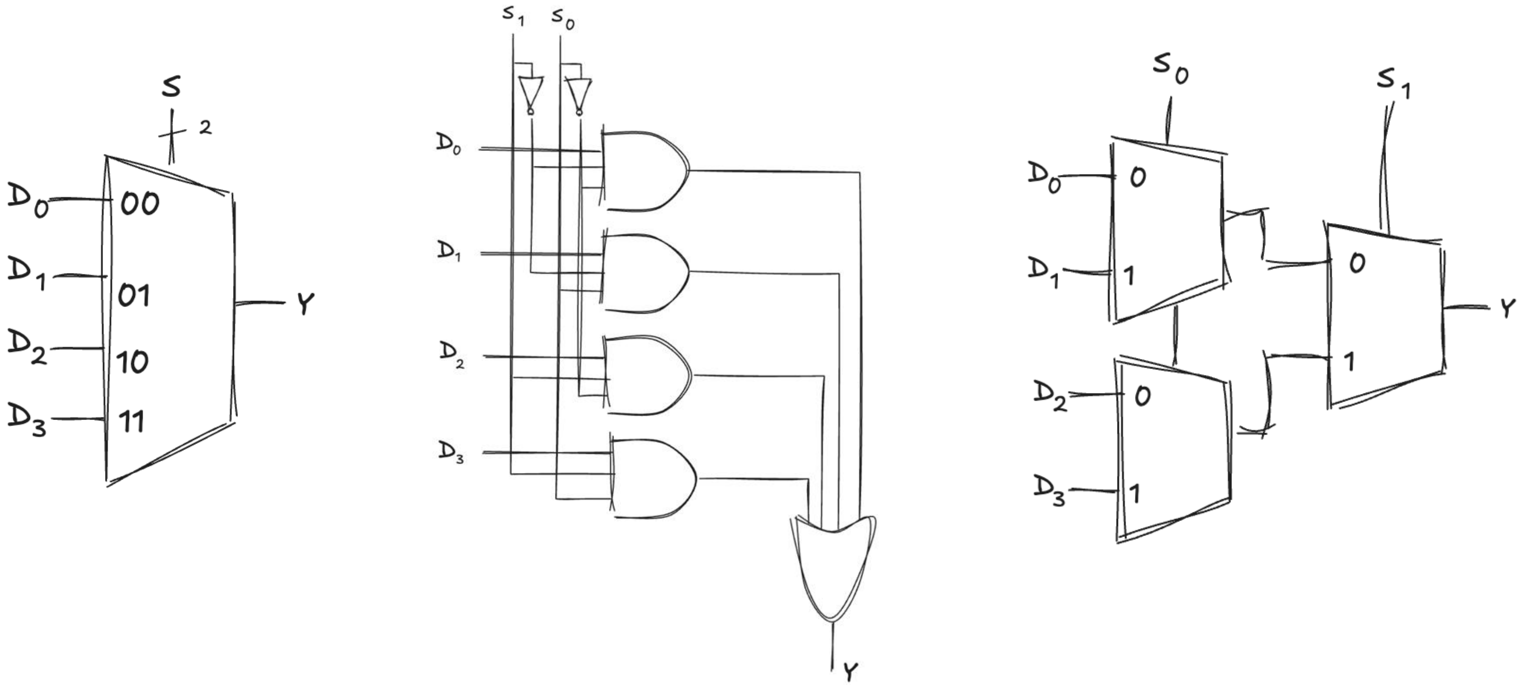
Y: S \ $D_{1:0}$	00	01	11	10
0	0	1	1	0
1	0	0	1	1

$Y = D_0\overline{S} + D_1 S$



Wider multiplexer

- A 4:1 multiplexer has four data inputs and one output and **two select signals** are needed to choose among the four data inputs
  - it can be built using sum-of-product logic, or multiple 2:1 multiplexers

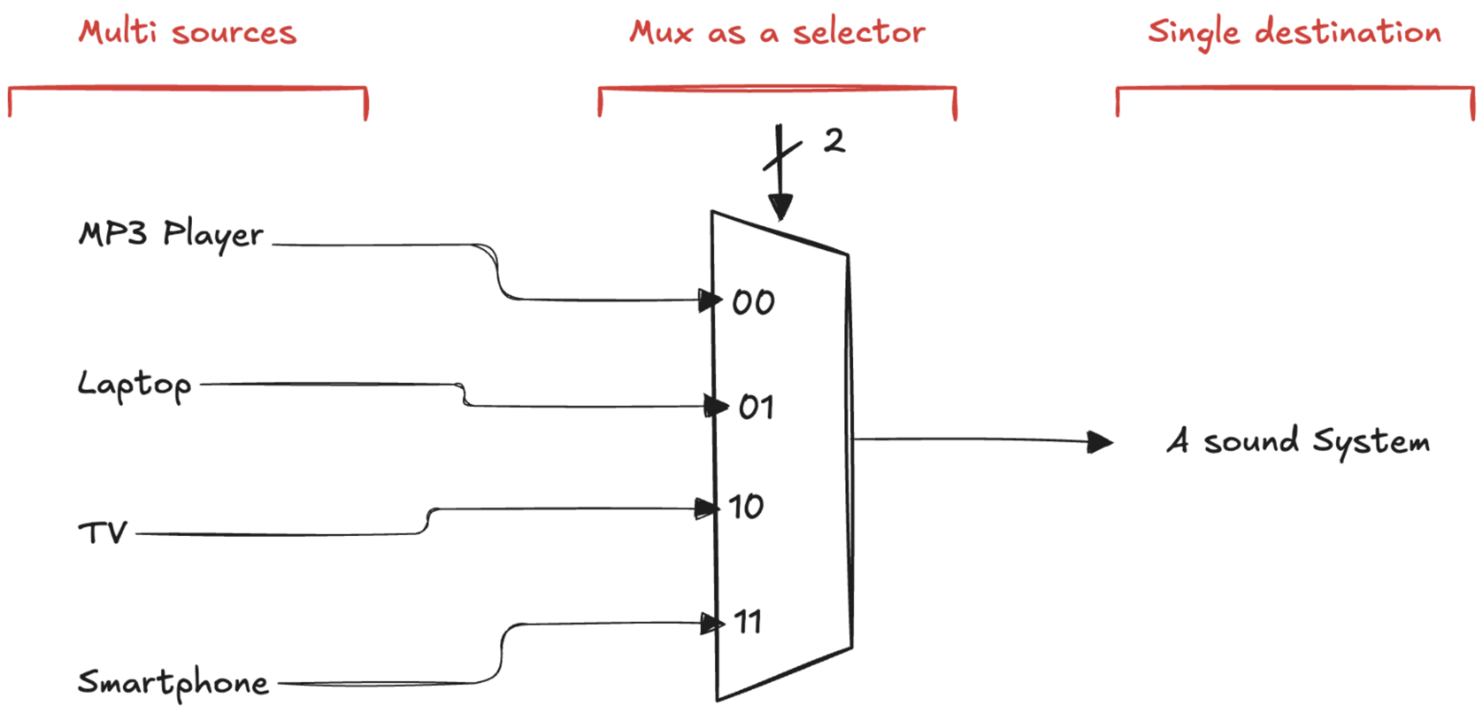


- Wider multiplexers (8:1 and 16:1) can be built by expanding the method
  - in general, **a N:1 multiplexer needs $\log_2 N$ select lines**

Multiplexer example

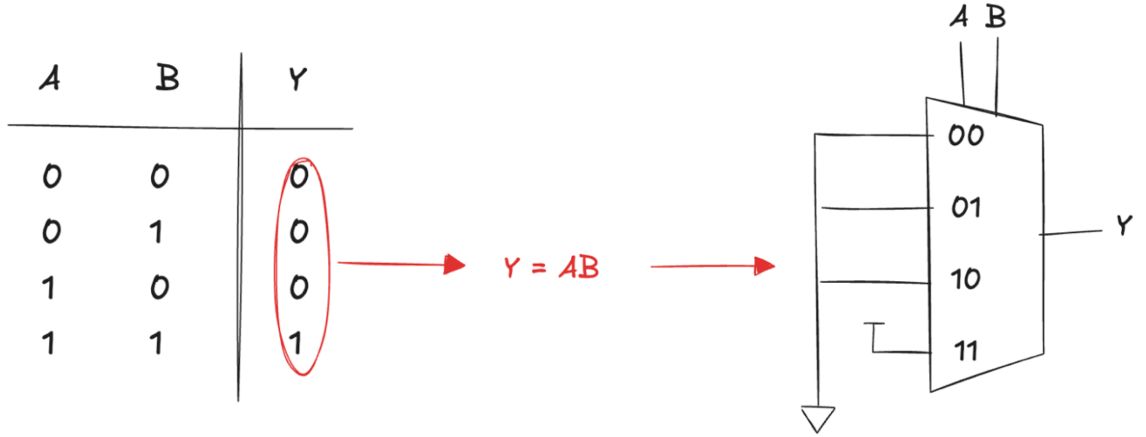
- A multiplexer is like a **data router**: connects only one input at a time, **allowing different data sources to share the same output line efficiently**

- This makes it possible to **handle many inputs in a controlled, sequential way** without needing separate circuits for each one



Multiplexer Logic (1)

- Multiplexers can be used as **lookup tables** to **perform logic functions**
- For example, we can implement a two-input AND gate:

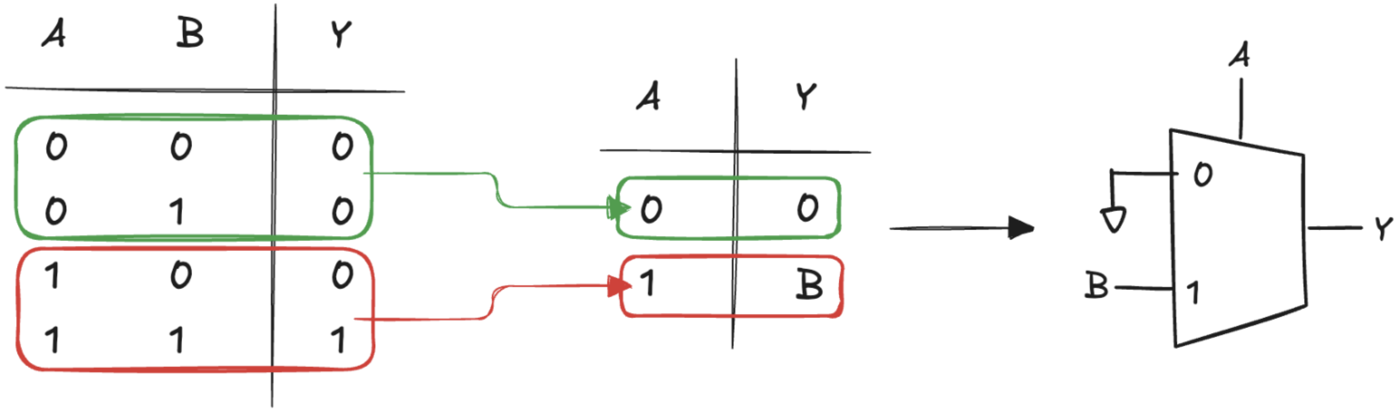


  - A and B serve as select lines
  - inputs are connected to 0 or 1, **according to the truth table**
  - changing the data inputs, the multiplexer **can be reprogrammed** to perform a different function

- In general, a **2^N-input multiplexer can be programmed to perform any N-input logic function** by applying 0 and 1 to the appropriate data inputs

Multiplexer Logic (2)

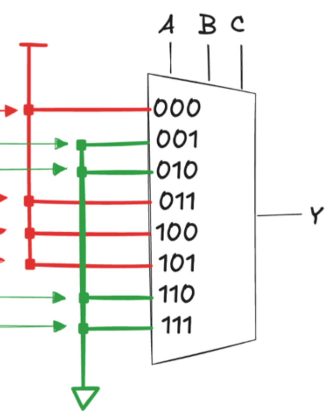
- We can **cut the multiplexer size in half**, using only a 2^{N-1} input multiplexer to perform any N-input logic function
- Provide **one of the literals to the multiplexer inputs**:



Multiplexer Logic (3)

- As an example, implement the following function $Y = A\,\overline{B} + \overline{B}\,\overline{C} + \overline{A}\,B\,C$
  - using a 8:1 multiplexer:

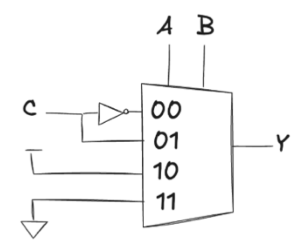
A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0



  - and a 4:1 multiplexer:

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

A	B	Y
0	0	$\overline{C}$
0	1	C
1	0	1
1	1	0



Multiplexer VHDL (1)

```
entity mux_4_1 is
    port( s0, s1: in std_logic;
          a, b, c, d: in std_logic;
          y: out std_logic
    );
end mux_4_1;

architecture rtl of mux_4_1 is
begin
    y <= a when s0='0' and s1='0' else
         b when s0='0' and s1='1' else
         c when s0='1' and s1='0' else
         d when s0='1' and s1='1' else
         '-';
end rtl;
```

Multiplexer VHDL (2)

```
entity mux_4_1_8bit is
    port( s0, s1: in std_logic;
            a, b, c, d: in std_logic_vector(0 to 7);
            y: out std_logic_vector(0 to 7)
    );
end mux_4_1_8bit;

architecture rtl of mux_4_1_8bit is
    signal sel: std_logic_vector(0 to 1);
begin
    with sel select
        y <= a when "00",
             b when "01",
             c when "10",
             d when "11",
             "--------" when others;
end rtl;
```

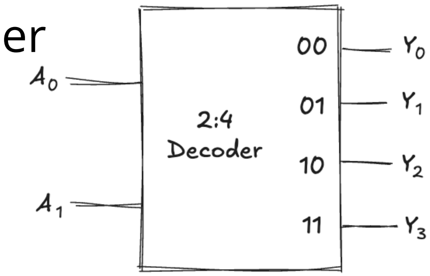
Multiplexer VHDL (3)

```
entity mux_4_1_Nbit is
    generic( N: integer );
    port( sel: in std_logic_vector(0 to 1);
          a, b, c, d: in std_logic_vector(0 to N-1);
          y: out std_logic_vector(0 to N-1)
    );
end mux_4_1_Nbit;
```

```
architecture rtl of mux_4_1_Nbit is
begin
    with sel select
        y <= a when "00",
             b when "01",
             c when "10",
             d when "11",
            (others => '-') when others;
end rtl;
```

Decoder

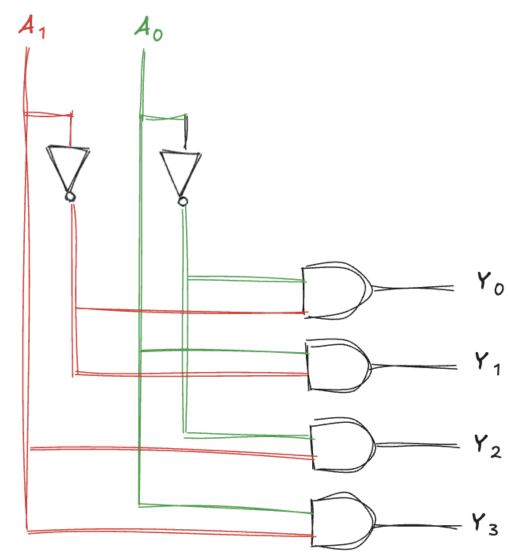
- A device with **N inputs** and **2^N outputs** that asserts **exactly one** of its outputs depending on the input combination
- For example, consider a 2:4 decoder
  - when $A_{1:0}$ = 00, Y_0 is 1
  - when $A_{1:0}$ = 01, Y_1 is 1
  - and so forth



A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

- The outputs are called **one-hot,** because just one is “hot” (HIGH) at a given time

- In the implementation, each gate depends on either the true or the complementary form of each input
  - an **N:2^N decoder** can be constructed from **2^N N-input AND gates** that accept the various combinations of true or complementary inputs
  - each output in a decoder represents a single minterm
    - for example, Y_3 represents the minterm A_1 A_0.

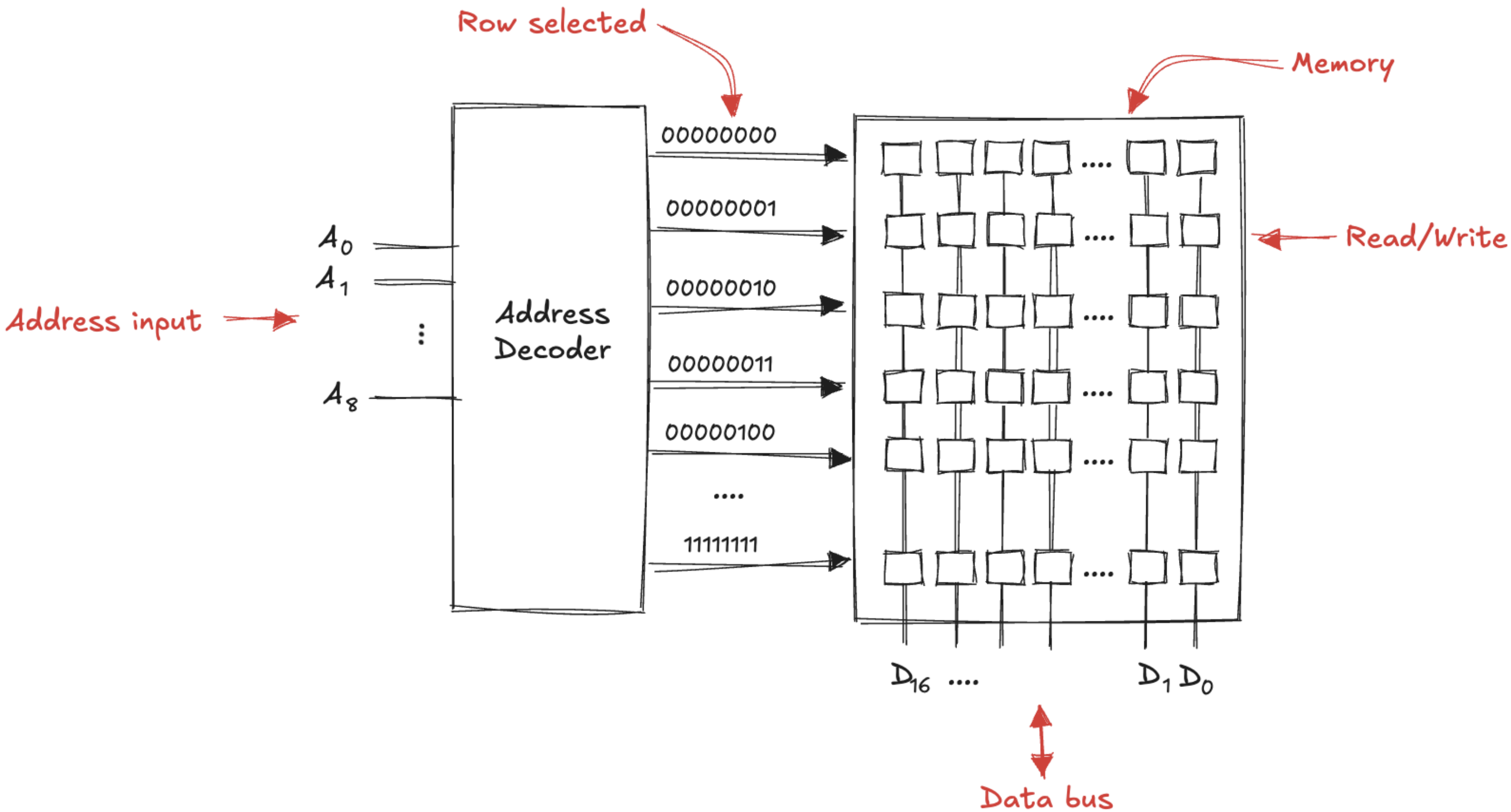


Decoder VHDL

```
entity decoder_2_4 is
    port( dec_in: in std_logic_vector(0 to 1);
          dec_out: out std_logic_vector(0 to 3)
    );
end decoder_2_4;

architecture rtl of decoder_2_4 is
begin
    with dec_in select
    dec_out <= "0001" when "00",
               "0010" when "01",
               "0100" when "10",
               "1000" when "11",
               "----" when others;
end rtl;
```

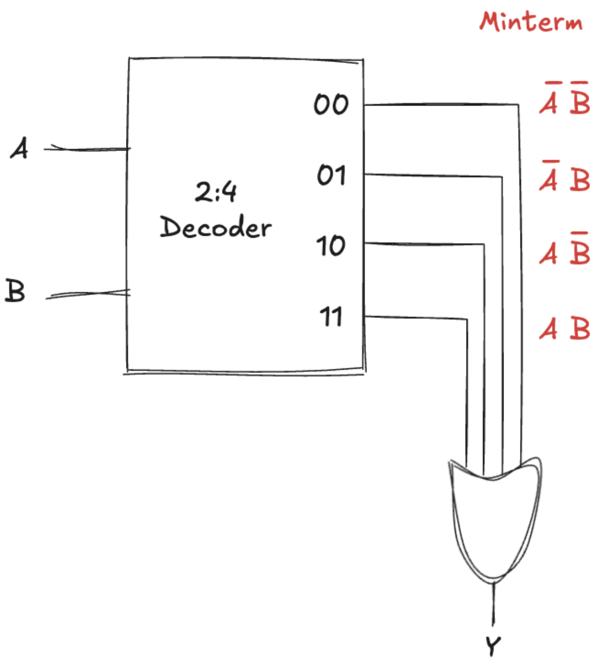
Decoder example

- A decoder can be used to control which of several lines is “on” based on the binary value of the input. It can be used in **memory address selection**:



Decoder Logic

- Decoders can be combined with OR gates **to build logic functions**
  - because each output of a decoder represents a single minterm, the function is built as the OR of all of the minterms in the function
  - example: a two-input XNOR function using a 2:4 decoder and a single OR gate



- An N-input function with M 1s in the truth table can be built with an N:2N decoder and an M-input OR.

Half and Full Adders

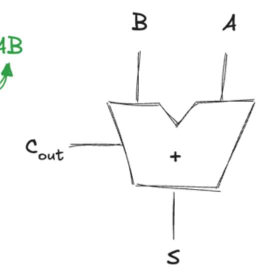
- Addition is one of the most common operations in digital systems

- **Half adder**
  - two inputs, A and B, and two outputs, S and C_{out}
  - S is the sum of A and B
  - C_{out} is the eventual carry out

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

$C_{out} = AB$

$S = A \oplus B$



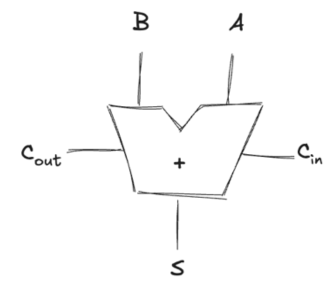
- In a multi-bit adder, C_{out} is added (carried) in to the next most significant bit
  - half adder lacks a C_{in} input to accept the C_{out} of the previous column

- **Full Adder**
  - accepts also the carry in C_{in}

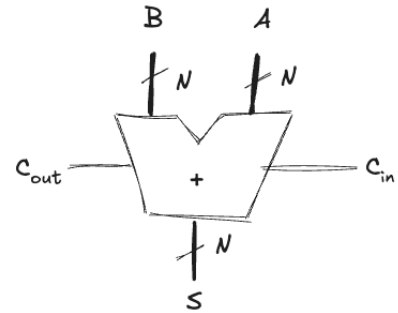
C_{in}	A	B	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

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

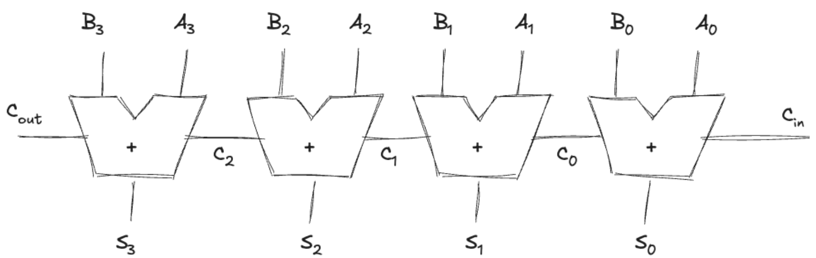
$S = A \oplus B \oplus C_{in}$



Carry Propagate Adder



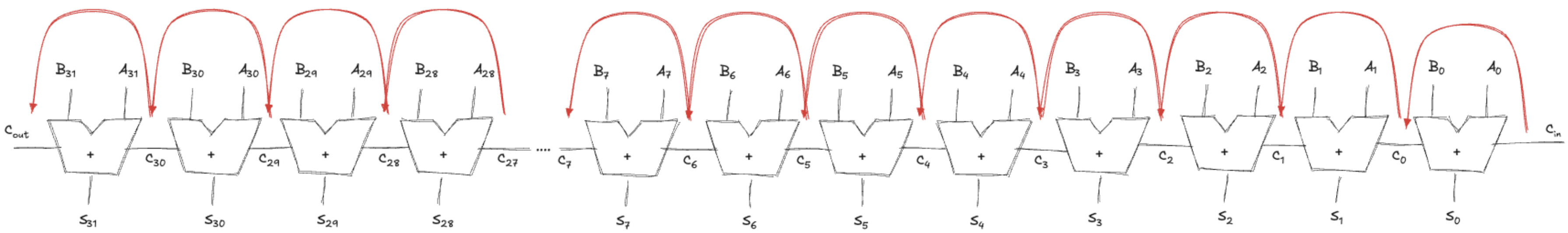
- A **N-bit adder** that sums two N-bit inputs and a carry-in to produce an N-bit result and a carry-out
  - the carry bit propagates into the adder
  - drawn like a full adder except that input/output are **busses** rather than single bits
- Simplest way to build: chain together N full adders **(Ripple-Carry Adder RCA)**
  - C_{ouy} of one stage acts as the C_{in} of the next stage



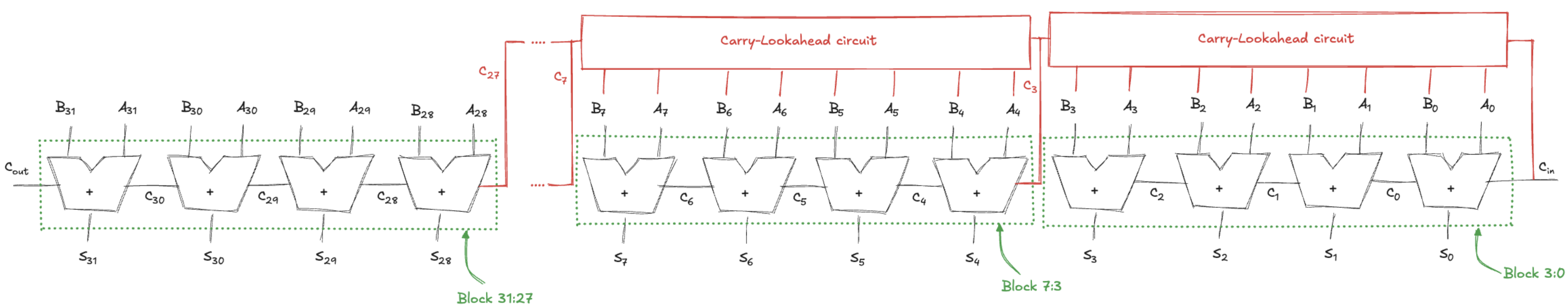
- A good application of modularity and regularity
  - full adder module is reused many times to form a larger system
- It is **slow when N is large**.
  - S_3 depends on C_2, which depends on C_1, and so forth
  - the delay of the adder $t_{riple} = N * t_{FA}$ grows directly with the number of bits

Carry-Lookahead Adder (1)

- RCA are slow because the carry must propagate:

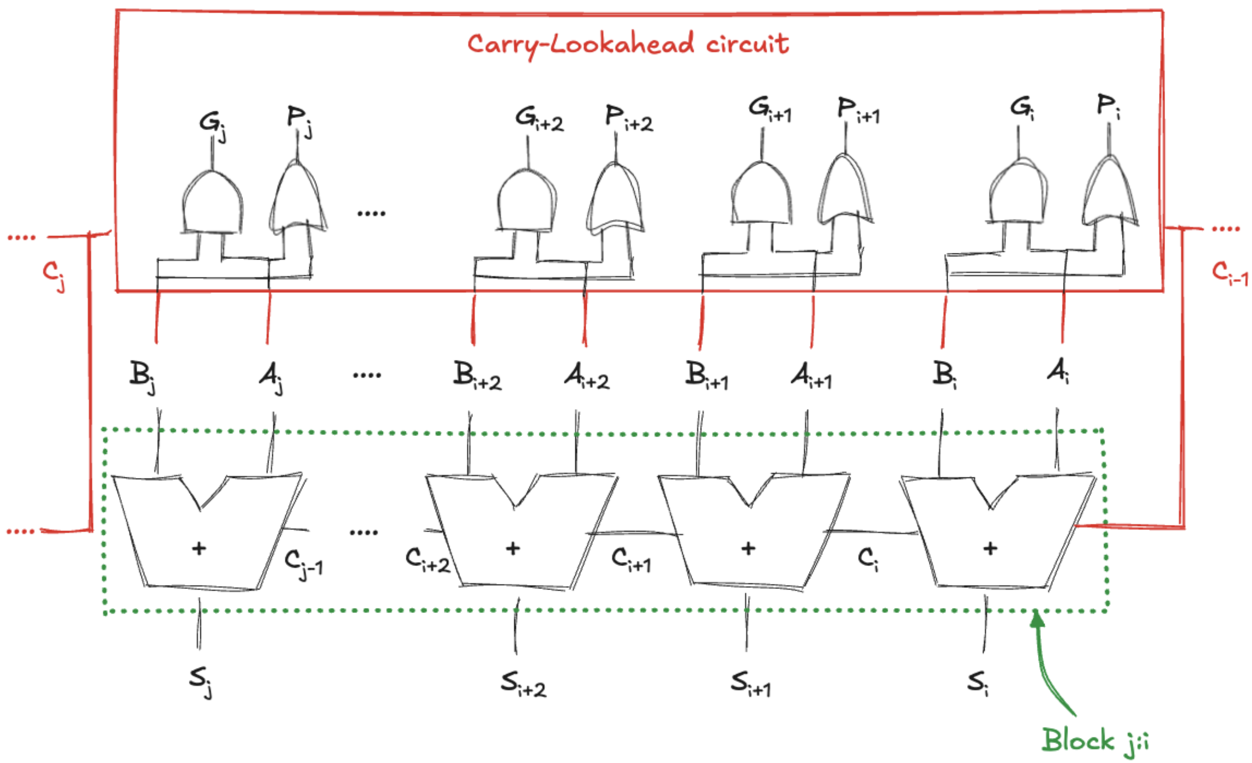


- The idea is to **divide the adder into blocks** and determine the carry out of a block **as soon as** the carry in is known
  - it looks ahead across the blocks rather than wait to ripple through all the full adders



Carry-Lookahead Adder (2)

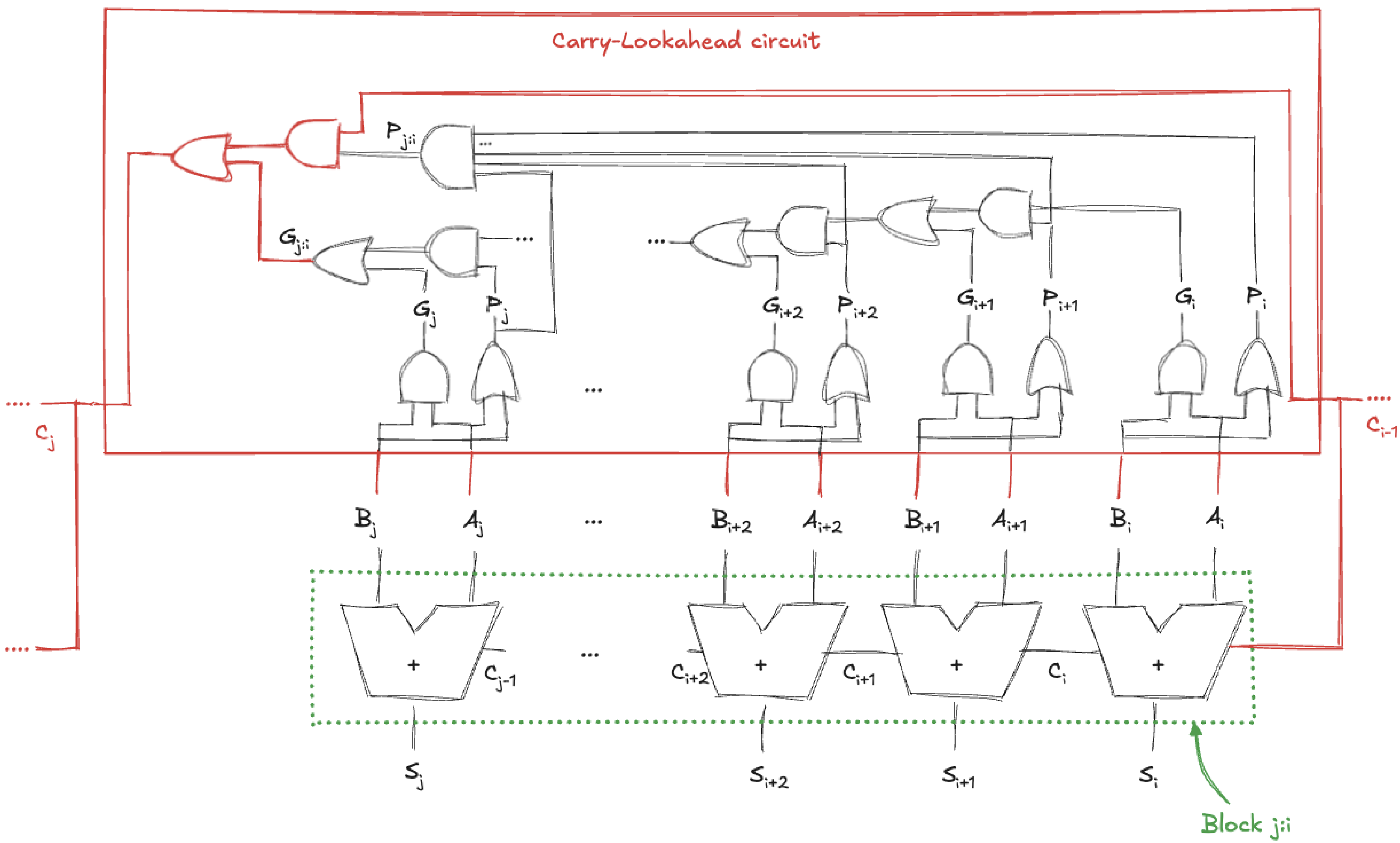
- **Carry-Lookahead Adder CLA** use **generate** (G) and **propagate** (P) signals
- A column "**generates**" a carry if it produces a carry independent of its carry-in
  - both input are 1 -> $\mathbf{G_i = A_i \text{ AND } B_i}$
- A column "**propagates**" a carry if it produces a carry whenever there is a carry-in
  - either one of the input is 1 -> $\mathbf{P_i = A_i \text{ OR } B_i}$



Carry-Lookahead Adder (2)

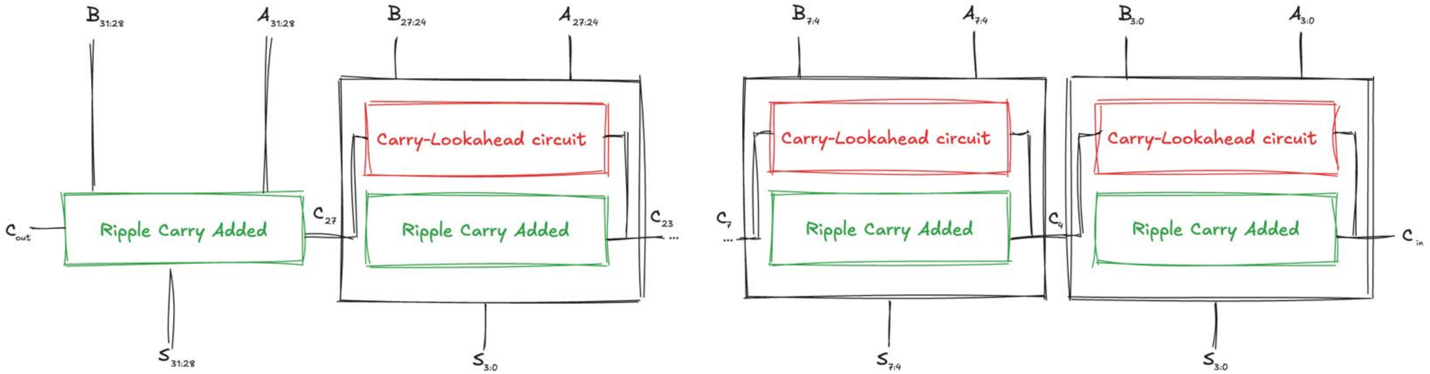
- We can extend to multiple-bit blocks
  - a block "generates" a carry if it produces a carry out independent of the carry-in
  - a block "propagate" a carry if it produces a carry out whenever there is a carry-in
- A block j:i (spanning columns i through j) **generates** a carry
  - if the most significant column generates a carry
  - if the previous column generated a carry and the most significant column propagates it
  - and so forth
  - $\mathbf{G_{j:i} = G_j + P_j(G_{j-1} + P_{j-1} (G_{j-2} + P_{j-2} (.... (G_{i+2} + P_{i+2}(G_{i+1} + P_{i+1}G_i))}$
    - e.g. $G_{7:4} = G_7 + P_7(G_6 + P_6(G_5 + P_5G_4)$
- A block i:j **propagates** a carry
  - if all the columns in the block propagate the carry
  - $\mathbf{P_{j:i} = P_jP_{j-1}P_{j-2}...P_{i+2}\, P_{i+1}\, P_i}$
    - e.g. $P_{3:0} = P_3P_2P_1P_0$
- So, we can quickly compute the carry out of the block
  - $\mathbf{C_j = G_{j:i} + P_{j:i}C_{i-1}}$

Carry-Lookahead Adder (3)



Carry-Lookahead Adder (4)

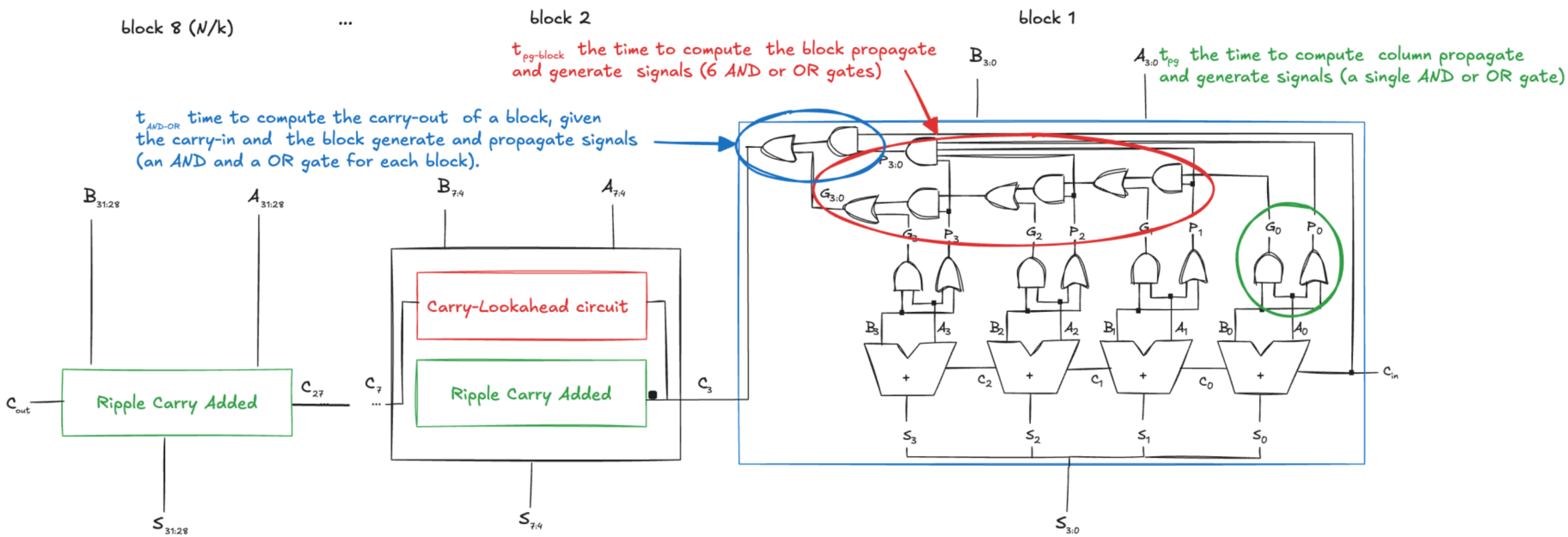
- A 32-bit carry-lookahead adder composed of eight 4-bit blocks:



- Each block contains a 4-bit ripple-carry adder and the **lookahead** circuit to compute the carry out of the block given the carry-in
- All blocks compute column generate and propagate signals **simultaneously**
- Then C_{in} advances to C_{out} through each block until the last
  - C_{in} proceeds through the lookahead gates to produce C_3
  - C_3 proceeds through its lookahead block to produce C_7
  - C_7 proceeds through its lookahead block to produce C_{11}
  - and so on until C_{27}, the carry-in to the last block
- The last block contains a short ripple-carry adder (no more the lookahead logic)

Carry-Lookahead Adder delay

- An N-bit adder divided into k-bit blocks has a delay
  - t_{pg} is the delay to generate signals P_i and G_i
  - t_{pg_block} is the delay to to generate signals $P_{j:i}$ and $G_{j:i}$ for a k-bit block
  - $t_{lookahead}$ is the delay from C_{in} to C_{out} through the logic of the k-bit CLA block



- $t_{CLA} = t_{pg} + t_{pg\_block} + (N/k - 1)\, t_{lookahead} + k\, t_{FA}$
  - For N>16, the carry-lookahead adder is much faster than the ripple-carry adder
  - however, the adder delay still increases linearly with N
- **Faster** adders: **more hardware, more expensive** and **power-hungry**
  - **trade-offs** must be considered when choosing an appropriate adder for a design

Ripple-Carry Adder vs Carry-Lookahead Adder

- Compare the delays of a 32-bit ripple-carry adder and a 32-bit carry-lookahead adder with 4-bit blocks

- Assume that each gate delay is 100ps and that a full adder delay is 300ps

- The propagation delay of the 32-bit ripple-carry adder is
  - $t_{riple} = N * T_{FA} = 32 * 300ps = 9.6$ ns

- The carry-lookahead adder has
  - $t_{pg} = 100ps$ (one AND or one OR)
  - $t_{pg\_block} = 6 * 100$ ps $= 600ps$ (six AND/OR gates)
  - $t_{lookahead} = 2 * 100$ ps $= 200ps$ (one OR and one AND)
  - $t_{CLA}=t_{pg}+t_{pg\_block}+(N/k-1)t_{lookahead}+kt_{FA} = 100 + 600 + (32/4 - 1)*200 + 4*300 = 3.3ns$

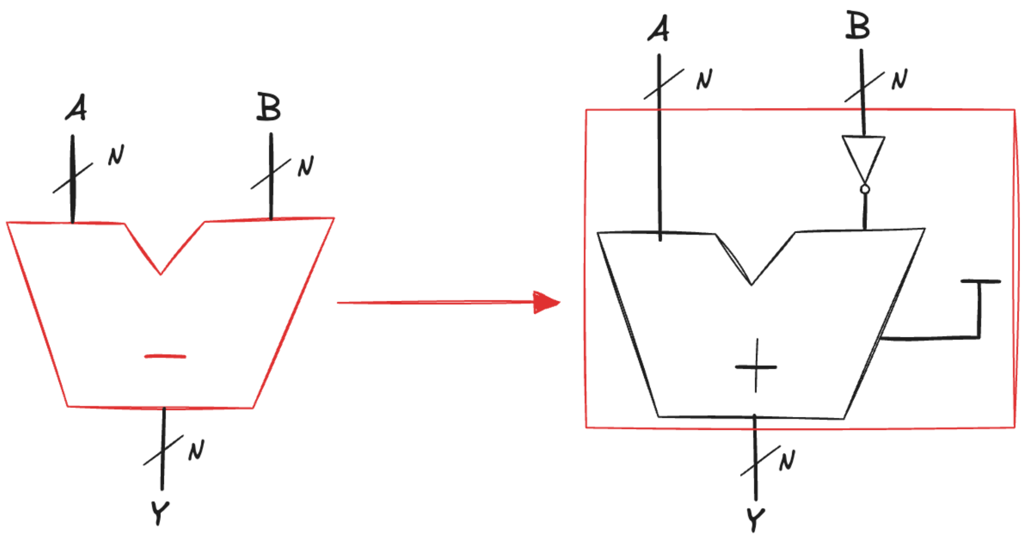
- The carry-lookahead adder is almost 3 times faster than the ripple-carry adder

CPA VHDL

```
entity adder is

    generic(N: integer := 8);

    port(A, B: in std_logic_vector(N-1 downto 0);
            C_in: in std_logic;
            S: out std_logic_vector(N-1 downto 0);
            C_out: out std_logic);
end;

architecture synth of adder is
    signal result: std_logic_vector(N downto 0);
begin
    result <= ("0" & A) + ("0" & B) + C_in;
    S <= result(N-1 downto 0);
    Cout <= result(N);
end;
```

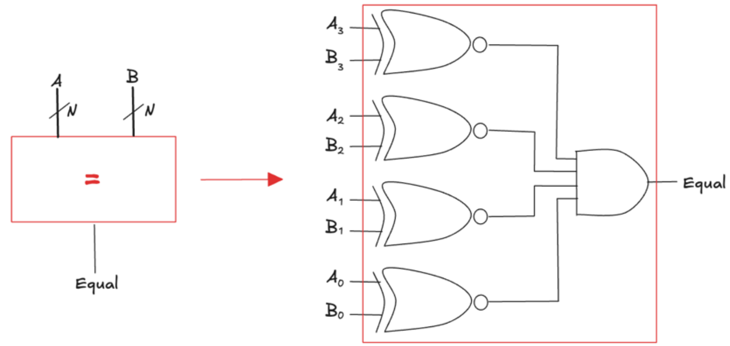
Subtractor

- Performed by taking the two's complement of the second number, then adding
- To compute Y = A − B
  - first create the two's complement of B
    - invert the bits of B to obtain $\bar{B}$
    - add 1 to get $-B = \bar{B} + 1$
  - add this quantity to A
    - $Y = A + \bar{B} + 1 = A - B$
- We can use a single carry-lookahead adder by adding $A + \bar{B}$ with $C_{in} = 1$



Comparators (1)

- Determines whether two binary numbers are equal or if one is greater or less than the other
  - receives two N-bit binary numbers A and B
  - an **equality comparator** produces a single output, indicating A is equal to B
  - a **magnitude comparator** produces one or more outputs, indicating the relative values of A and B
- The equality comparator is the simpler piece of hardware

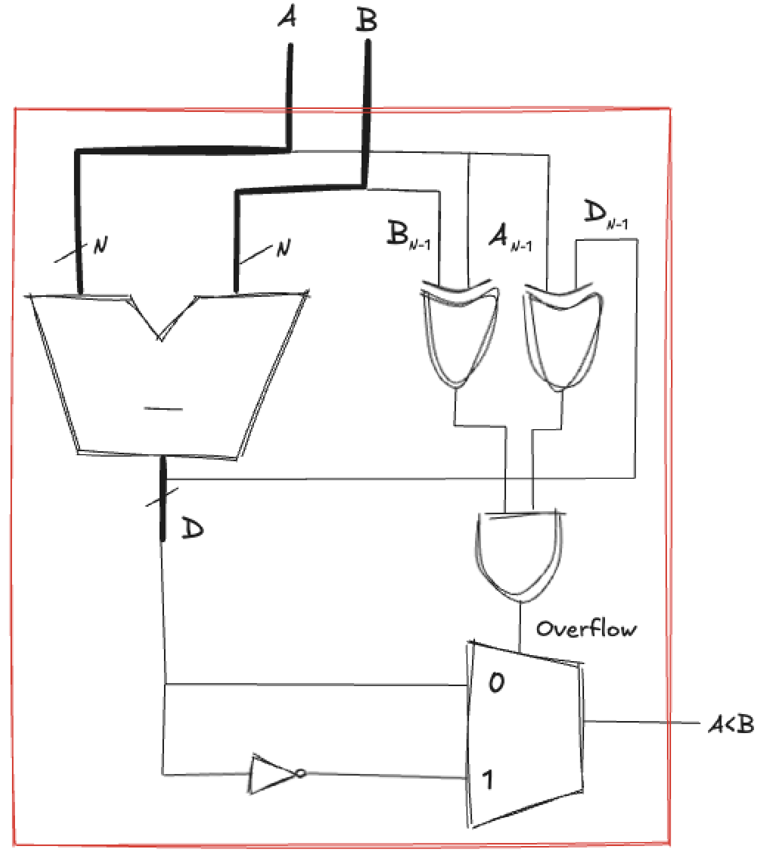


  - it checks whether the corresponding bits in each column of A and B are equal
    - using XNOR gates
  - the numbers are equal if all the columns are equal

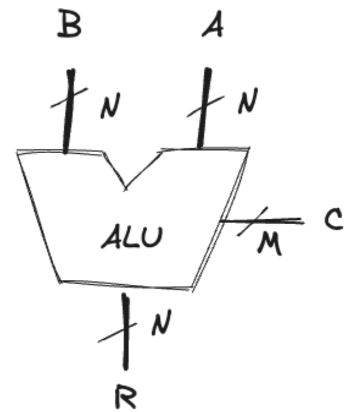
Comparators (2)

- Magnitude comparison of signed numbers is usually done by computing A − B and looking at the sign (most significant bit) of the result
  - if the result is negative (i.e., the sign bit is 1), then A is less than B
  - otherwise, A is greater than or equal to B

- This comparator functions **incorrectly upon overflow**
  - overflow occurs when
    - the two inputs have different signs
    - and the sign of the subtraction result has a different sign than the A input

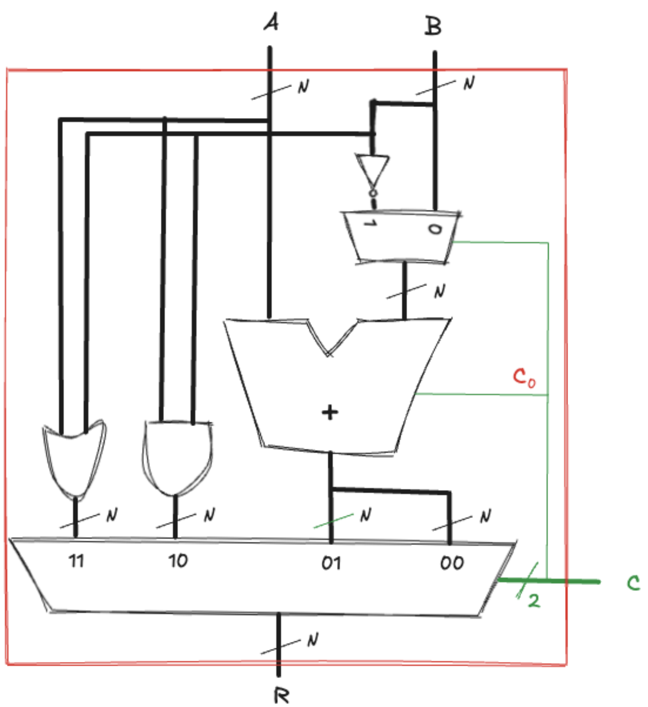


Arithmetic/Logical Unit (1)

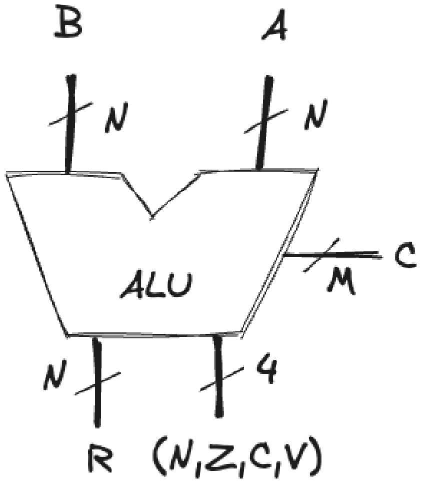


- ALU **combines** a variety of **mathematical** and **logical** operations into a single unit
  - addition, subtraction, AND, and OR operations
  - **control signal** to specifies the function to perform
  - the heart of most computer systems
- The following implementation contains
  - an N-bit adder, N 2-input AND and OR gates

Control C_1	C_0	Funtion
0	0	Add
0	1	Subtract
1	0	AND
1	1	OR

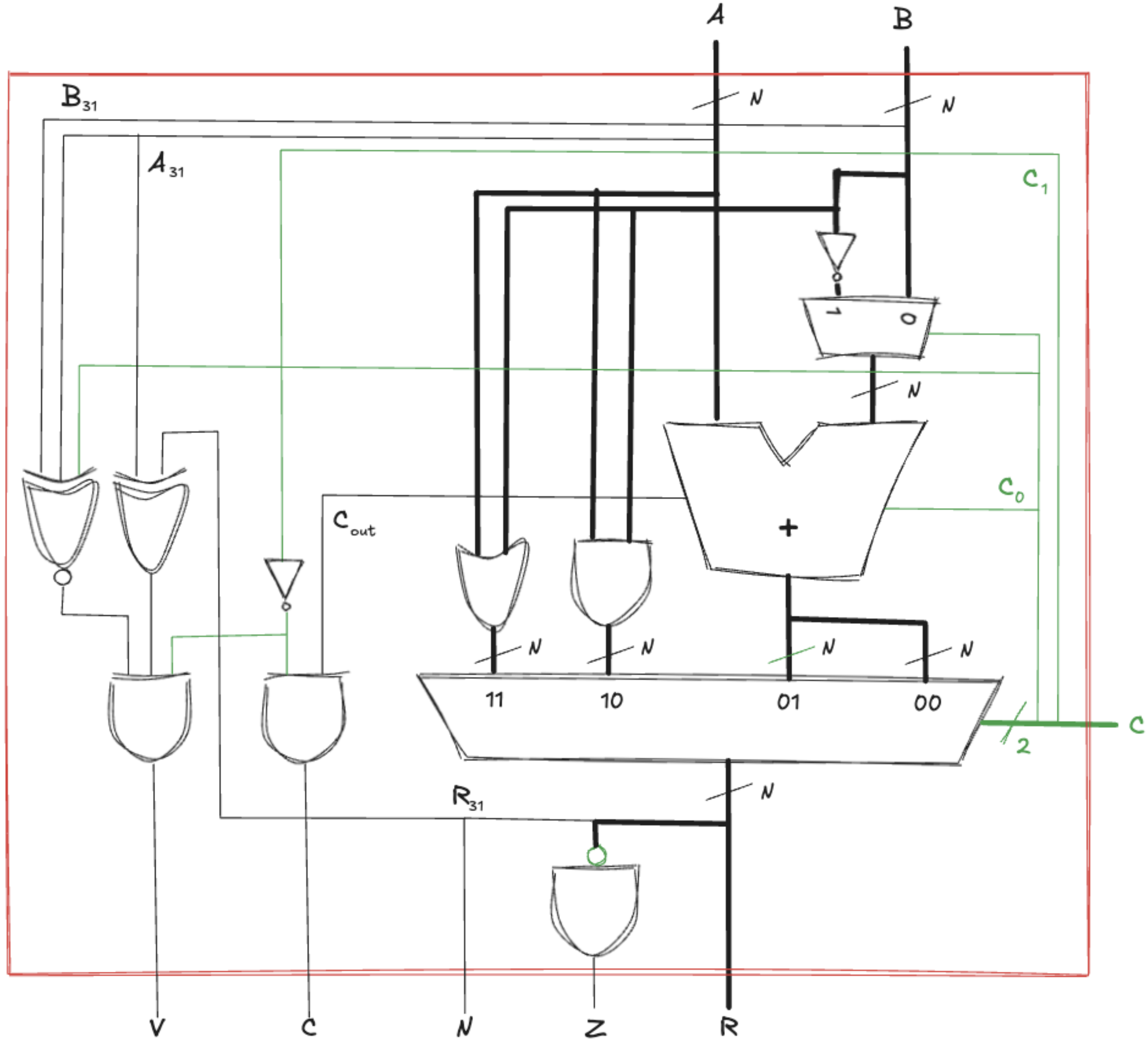


Arithmetic/Logical Unit (2)



- ALU can produce **flags** to provide information about the result
- Z (**zero**)
  - all the bits of the output are 0
- N (**negative**)
  - the most significant bit of the output
- C (**carry**)
  - adder produces a carry out and the ALU is performing addition or subtraction
- V (**overflow**)
  - overflow occurs when the addition of two same signed numbers produces a result with the opposite sign
    - ALU is performing addition or subtraction (C_1=0)
    - A and Sum have opposite signs
    - A and B have the same sign, and the adder is performing addition (C_0=0)
    - or A and B have opposite sign, and the adder is performing subtraction (C_0=1)

Arithmetic/Logical Unit (3)



Arithmetic/Logical Unit (4)

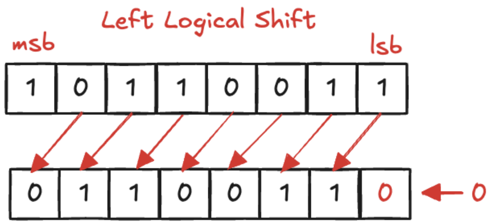
- The ALU flags can also be used for comparisons
  - ALU computes A-B and look at the flags:
    - (1) if Z is asserted, the result is 0, so A==B
    - (2) otherwise, A is not equal to B
    - (3) magnitude comparison is messier, we see whether the answer is negative (N). However, if overflow occurs, the N flag will be incorrect. Hence, A is less than B if the answer is negative and there is no overflow or if the answer is positive but overflow occurred
    - (4) A is less then or equal to B if occur the OR of condition (1) and (3)
    - (5) A is greater then B, if it is not less then or equal to B, the negation of condition (4)
    - (6) A is greater then or equal to B, if it is not less the... B, the negation of condition (3)

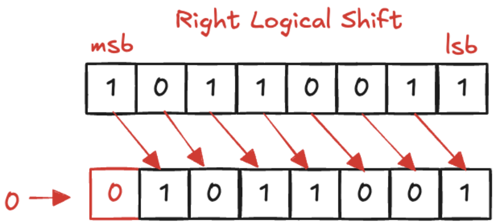
	Comparison	Operation
(1)	==	Z
(2)	$\neq$	$\overline{Z}$
(3)	$<$	$N \oplus V$
(4)	$\leq$	$Z + (N \oplus V)$
(5)	$>$	$\overline{Z}\,(\overline{N \oplus V})$
(6)	$\geq$	$\overline{N \oplus V}$

- Processors can use combinations of status bits for **conditional branching** (e.g., checking if a value is less than zero or within bounds)
- **Many variations** on this basic ALU exist that support other functions

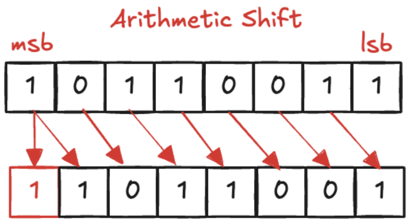
Shifters (1)

- Shifters **moves binary data**, crucial in arithmetic operations, logical operations, and data manipulation. The main types are:
  - **logical shifter**: shifts all bits to the left (or right), inserting a 0 into lsb (or msb) and discarding msb (or lsb)





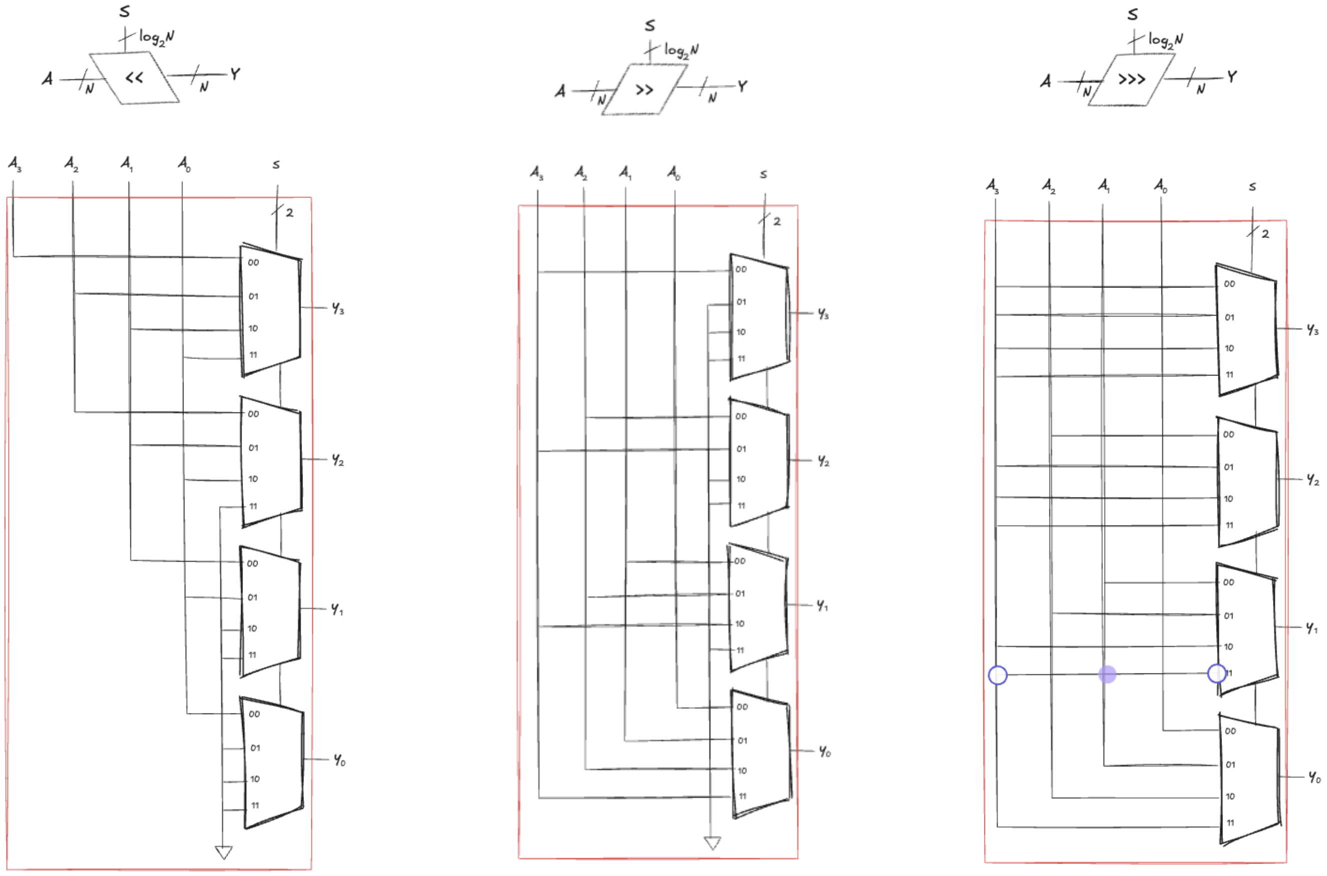
  - **arithmetic shifter**: like a logical shift with sign bit preserved if possible



- A left shift by N bits multiplies the number by 2^N
  - $000011_2 << 4 = 110000_2$ is equivalent to $3_{10} \times 2^4 = 48_{10}$
- A right shift by N bits multiplies the number by 2^N
  - $11100_2 >>> 2 = 11111_2$ is equivalent to $-4_{10}/2^2 = -1_{10}$

Shifters (2)

- An N-bit shifter can be built from N N:1 multiplexers
- Depending on the value of a log2(N)-bit shift amount control signal, the output receives the input shifted by 0 to N-1 bits



Rotators

- Rotator rotates (left or right) a number **in a circle** such that empty spots are filled with bits shifted off the other end

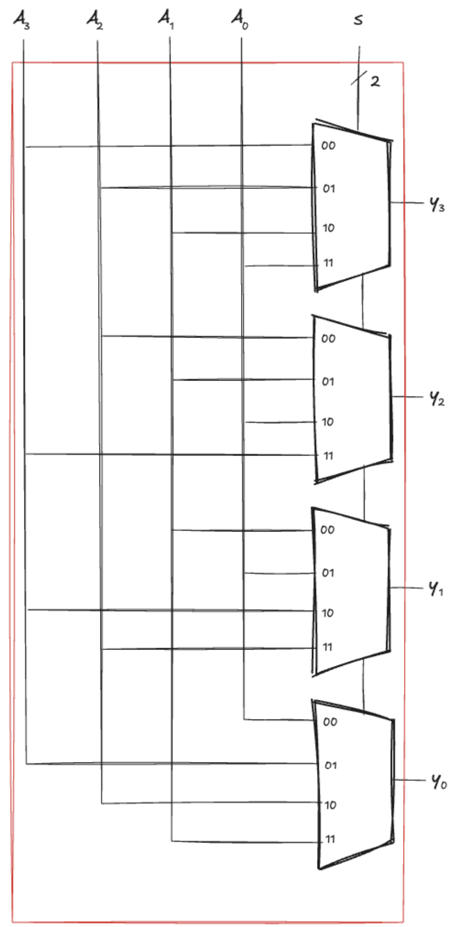
←

	Y_3	Y_2	Y_1	Y_0
s=00 →	A_3	A_2	A_1	A_0
s=01 →	A_2	A_1	A_0	A_3
s=10 →	A_1	A_0	A_3	A_2
s=11 →	A_0	A_3	A_2	A_1

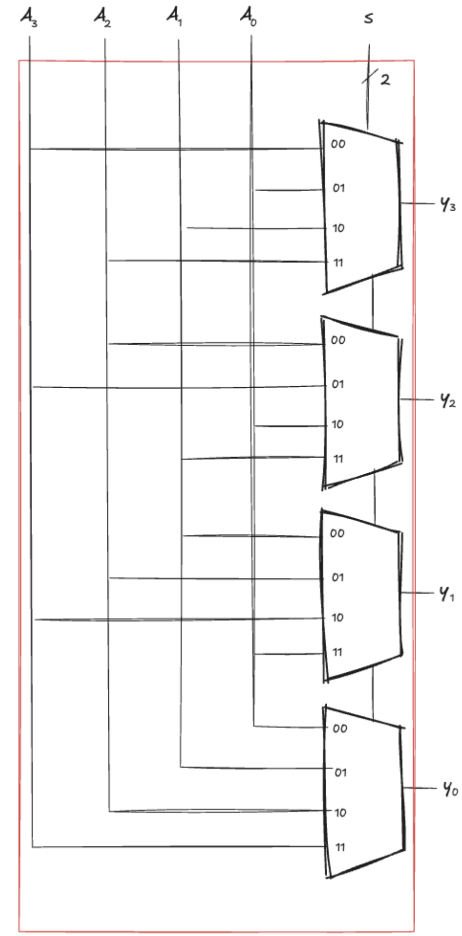
→

	Y_3	Y_2	Y_1	Y_0
s=00 →	A_3	A_2	A_1	A_0
s=01 →	A_0	A_3	A_2	A_1
s=10 →	A_1	A_0	A_3	A_2
s=11 →	A_2	A_1	A_0	A_3

Rotate Left

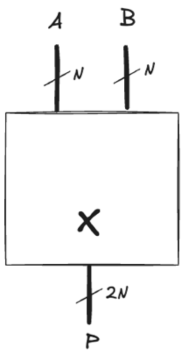


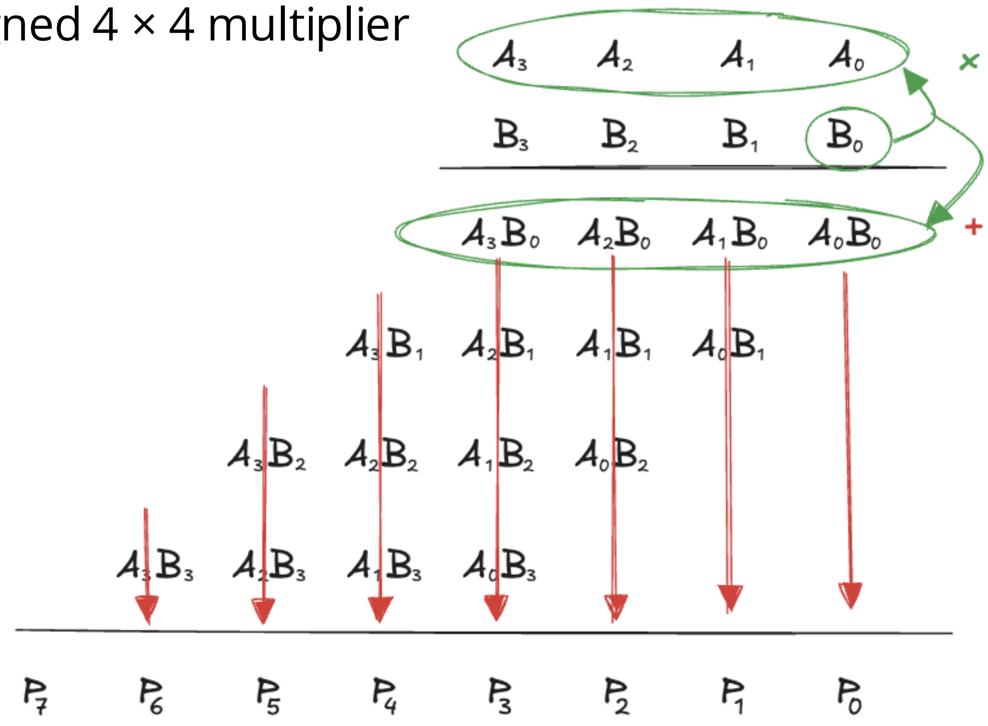
Rotate Right



Multiplier (1)

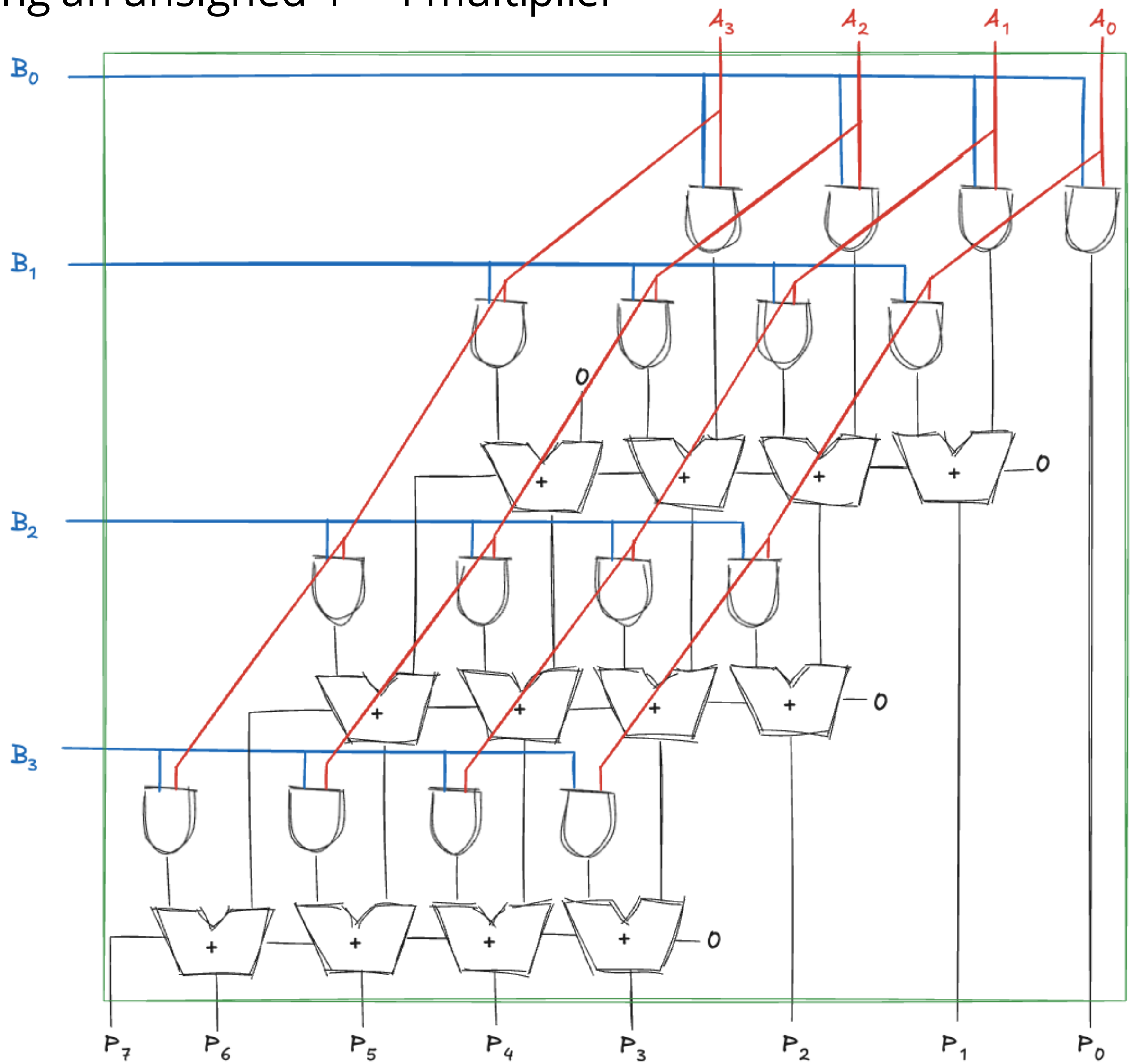
- Multiplication of 1-bit numbers is equivalent to the AND operation
- A N×N multiplier produces a 2N-bit result
    - partial product is the AND of a single multiplier bit (B_3, B_2,...) with the multiplicand bits (A_3, A_2,...)
    - each partial products is added to the shifted next partial product
        - B_0 AND (A_3, A_2, A_1, A_0) is added to B_1 AND (A_3, A_2, A_1, A_0)
- Considering an unsigned 4 × 4 multiplier





Multiplier (2)

- Considering an unsigned 4 × 4 multiplier

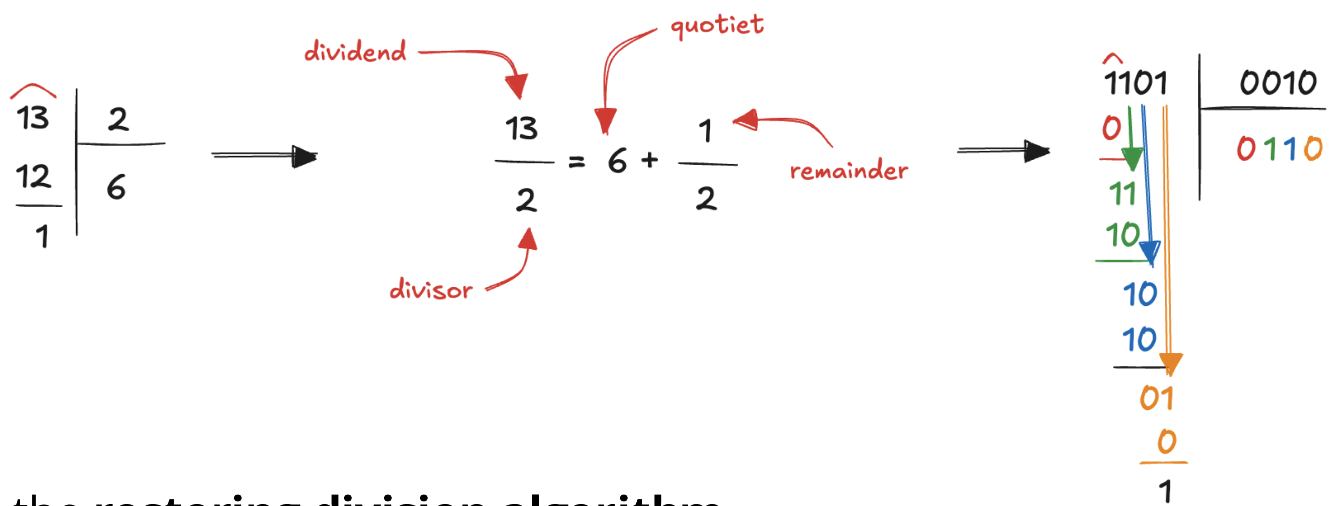


Division (1)

- The most complex operation to perform among all the arithmetic operations

$$\frac{A}{B} = Q + \frac{R}{B}$$

  - the basic idea is the same as long division in decimal
    - how many times the divisor can fit into sections of the dividend without exceeding it?
    - each time it fits, we add a 1 to the quotient
    - if it doesn't fit, we add a 0



- This is the **restoring division algorithm**

Division (2)

```
R' = 0
for i=N-1 to 0:
    R = R' << 1, Ai
    D = R - B
    if D<0:
        Qi = 0
        R' = R
    else:
        Qi=1
        R'=D
    R = R'
```

A = 1101, B = 0010

R' = 0000

i = N-1 = 3

R = R' << 1, A_3 = 0000 << 1, 1 = 0001

D = R - B = 0001 - 0010 < 0

Q_3 = 0 and R' = R = 0001

i = 2

R = R' << 1, A_2 = 0001 << 1, 1 = 0011

D = R - B = 0011 - 0010 = 0001 >= 0

Q_2 = 1 and R' = D = 0001

i = 1

R = R' << 1, A_1 = 0001 << 1, 0 = 0010

D = R - B = 0010 - 0010 = 0000 >= 0

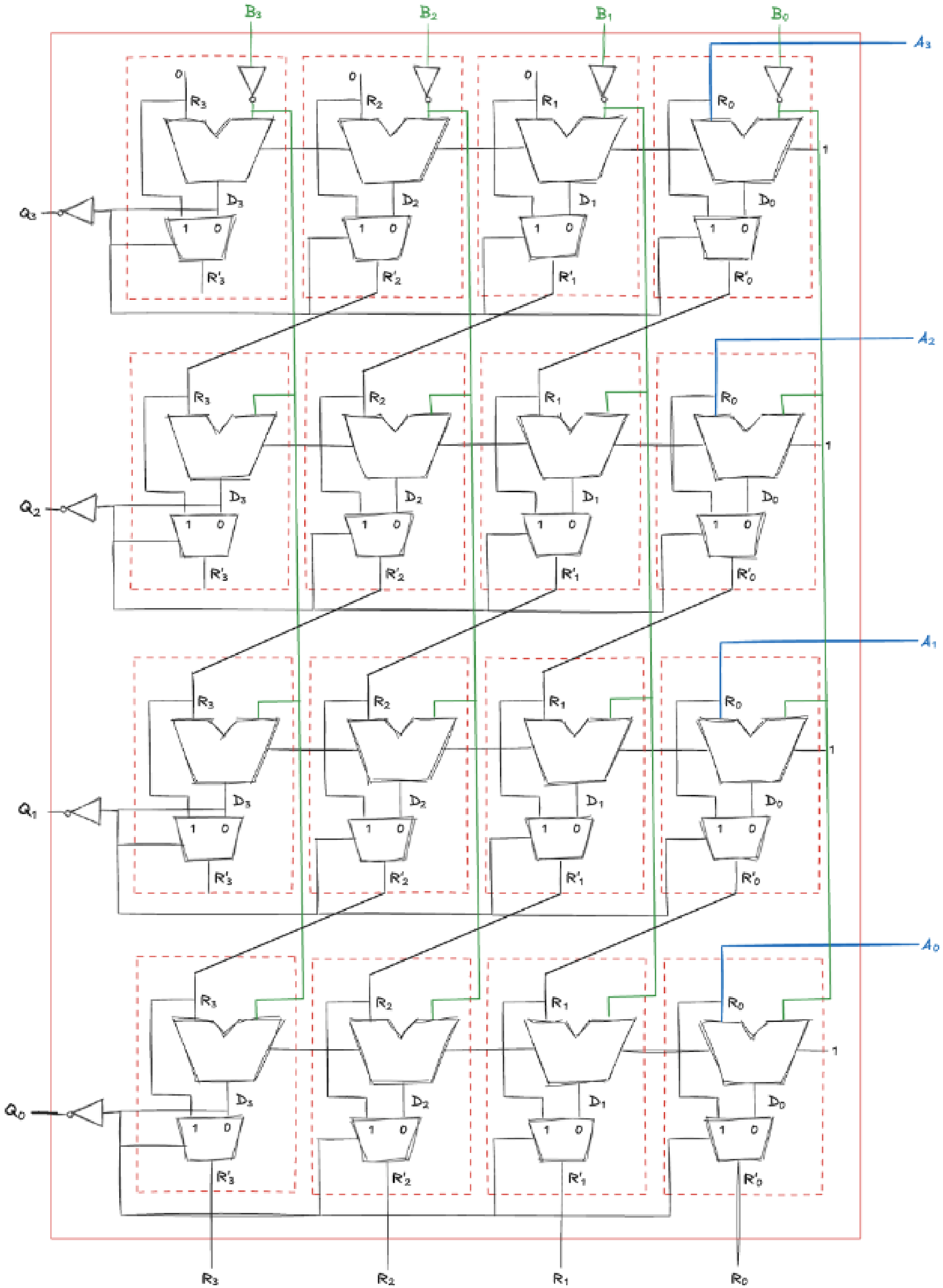
Q_1 = 1 and R' = D = 0000

i = 0

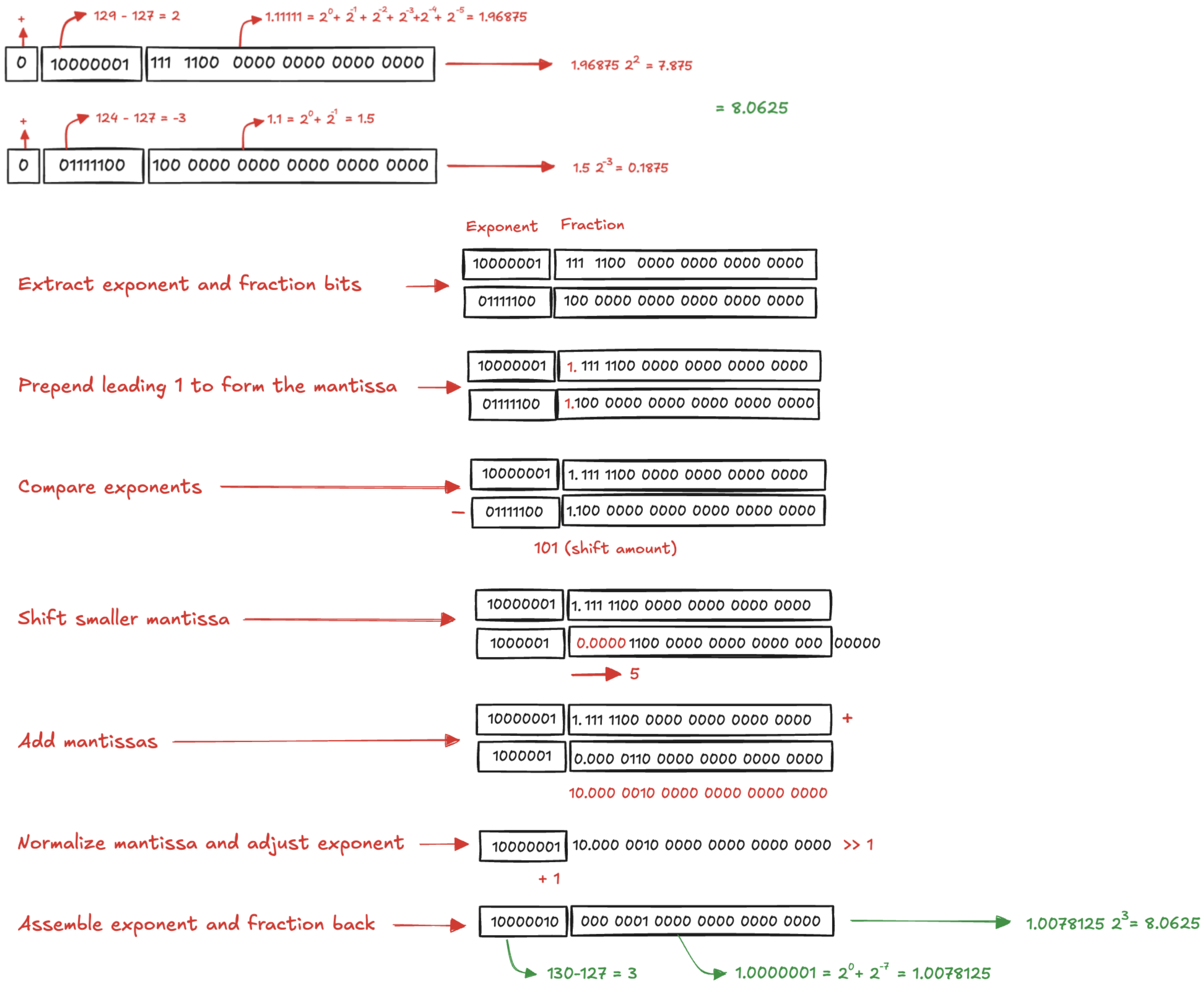
R = R' << 1, A_0 = 0000 << 1, 1 = 0001

D = R - B = 0001 - 0010 < 0

Q_0 = 0 and R' = R = 0001



Floating-Point Addition (1)



Floating-Point Addition (2)

- Floating-point arithmetic is usually done in hardware to make it fast
  - **floating-point unit** (FPU)
  - typically, distinct from the **central processing unit** (CPU)
- The **infamous floating-point division bug** (1994)
  - the processor returns incorrect binary floating point results when dividing certain pairs of high-precision numbers…
  - missing values in a lookup table used by the FPU's division algorithm led to calculations acquiring small errors
  - these errors only occur rarely producing small deviations from the correct output
  - in certain circumstances the errors can occur frequently and lead to more significant deviations
  - **Byte magazine** estimated that **1 in 9 billion floating point divides with random parameters would produce inaccurate results**
  - **December 1994**, Intel recalled the defective processors in what was **the first full recall of a computer chip**: it cost $475 million