Arithmetic Circuits

Design of Digital Circuits 2014
Srdjan Capkun
Frank K. Gürkaynak

http://www.syssec.ethz.ch/education/Digitaltechnik_14

In This Lecture

- **Why are arithmetic circuits so important**
- **Adders**
  - Adding two binary numbers
  - Adding more than two binary numbers
  - Circuits Based on Adders
- **Multipliers**
- **Functions that do not use adders**
- **Arithmetic Logic Units**

Motivation: Arithmetic Circuits

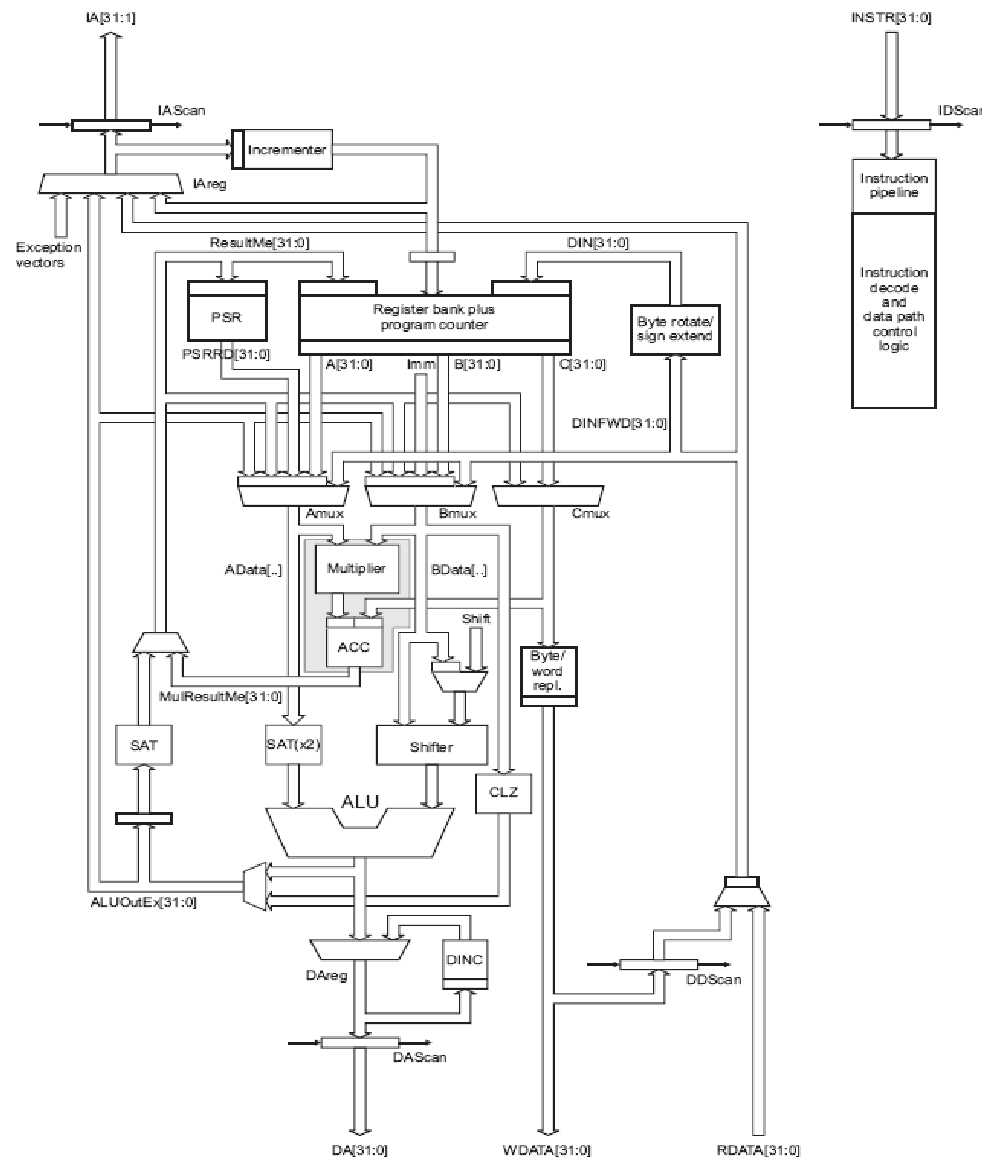
- **Core of every digital circuit**
  - Everything else is side-dish, arithmetic circuits are the heart of the digital system

- **Determines the performance of the system**
  - Dictates clock rate, speed, area
  - If arithmetic circuits are optimized performance will improve

- **Opportunities for improvement**
  - Novel algorithms require novel combinations of arithmetic circuits, there is always room for improvement

Example: ARM Microcontroller

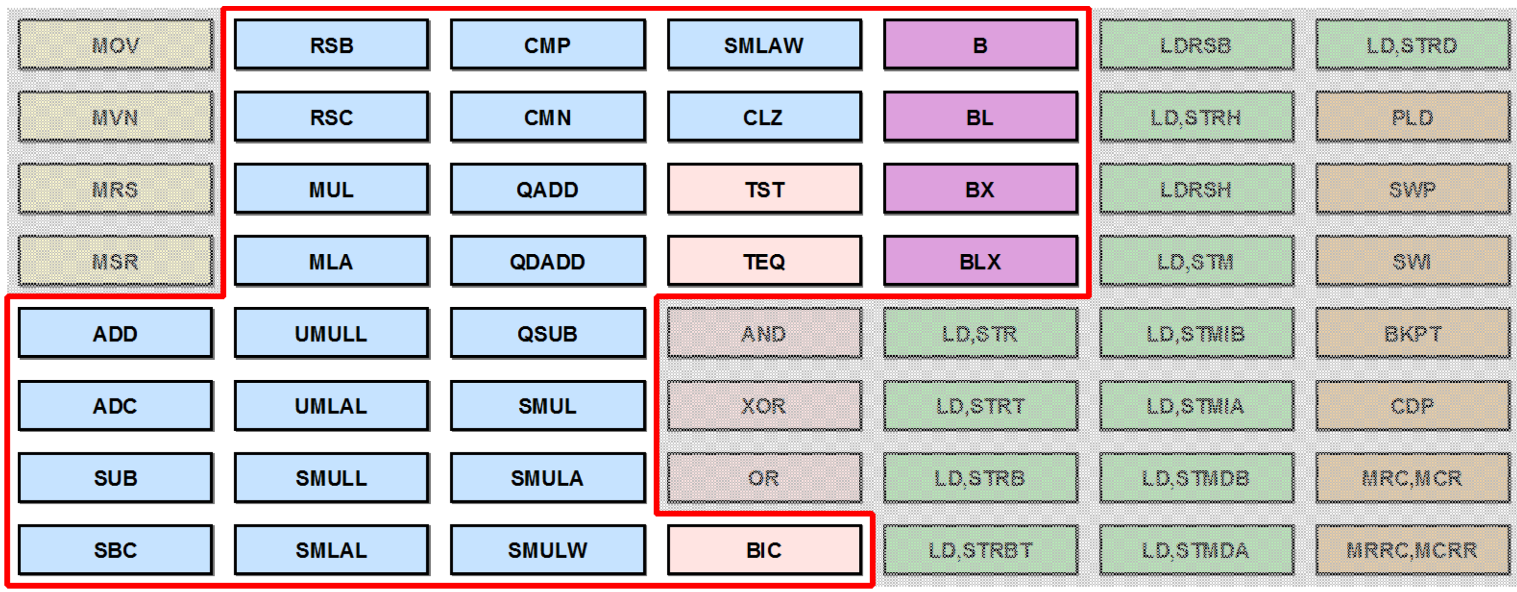
- **Most popular embedded micro controller.**
- **Contains:**
  - Multiplier
  - Accumulator
  - ALU/Adder
  - Shifter
  - Incrementer



Example: ARM Instructions

MOV	RSB	CMP	SMLAW	B	LDRSB	LD,STRD
MVN	RSC	CMN	CLZ	BL	LD,STRH	PLD
MRS	MUL	QADD	TST	BX	LDRSH	SWP
MSR	MLA	QDADD	TEQ	BLX	LD,STM	SWI
ADD	UMULL	QSUB	AND	LD,STR	LD,STMIB	BKPT
ADC	UMLAL	SMUL	XOR	LD,STRT	LD,STMIA	CDP
SUB	SMULL	SMULA	OR	LD,STRB	LD,STMDB	MRC,MCR
SBC	SMLAL	SMULW	BIC	LD,STRBT	LD,STMDA	MRRC,MCRR

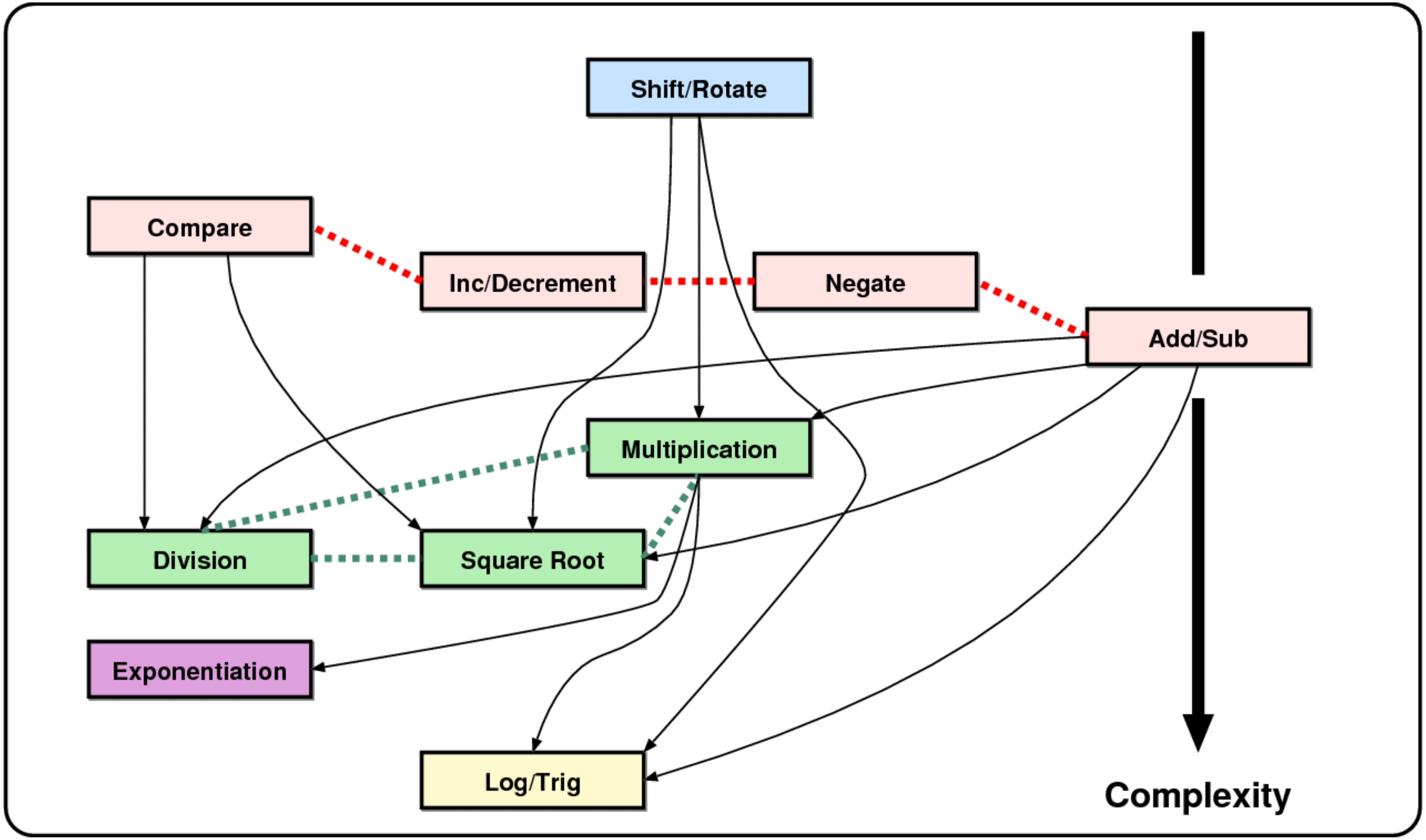
Arithmetic Based Instructions of ARM



Types of Arithmetic Circuits

- **In order of complexity:**
  - Shift / Rotate
  - Compare
  - Increment / Decrement
  - Negation
  - Addition / Subtraction
  - Multiplication
  - Division
  - Square Root
  - Exponentation
  - Logarithmic / Trigonometric Functions

Relation Between Arithmetic Operators



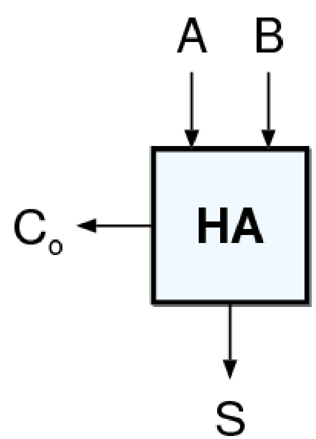
Addition

- **Addition is the *most important* operation in computer arithmetic. Our topics will be:**
  - **Adding 1-bit numbers** : Counting bits
  - **Adding two numbers** : Basics of addition
  - **Circuits based on adders** : Subtractors, Comparators
  - **Adding multiple numbers** : Chains of Adders
- **Later we will also talk about fast adder architectures**

Half-Adder (2,2) Counter

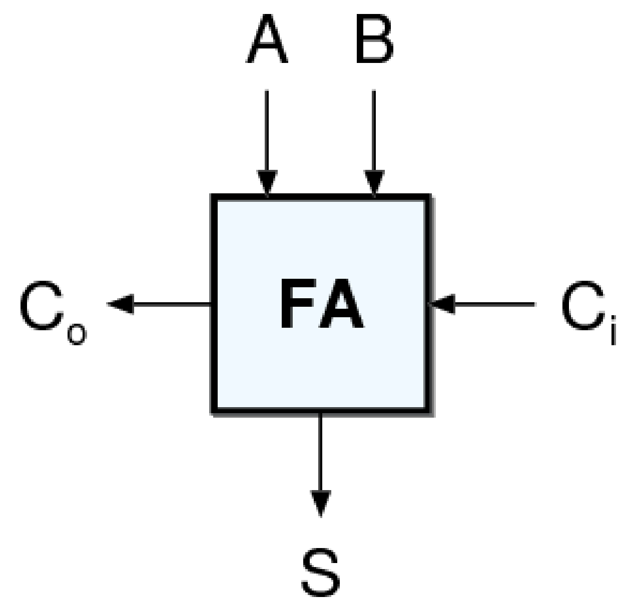
- The ***Half Adder*** (HA) is the simplest arithmetic block
- It can add two 1-bit numbers, result is a 2-bit number
- Can be realized easily

A	B	C_o	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

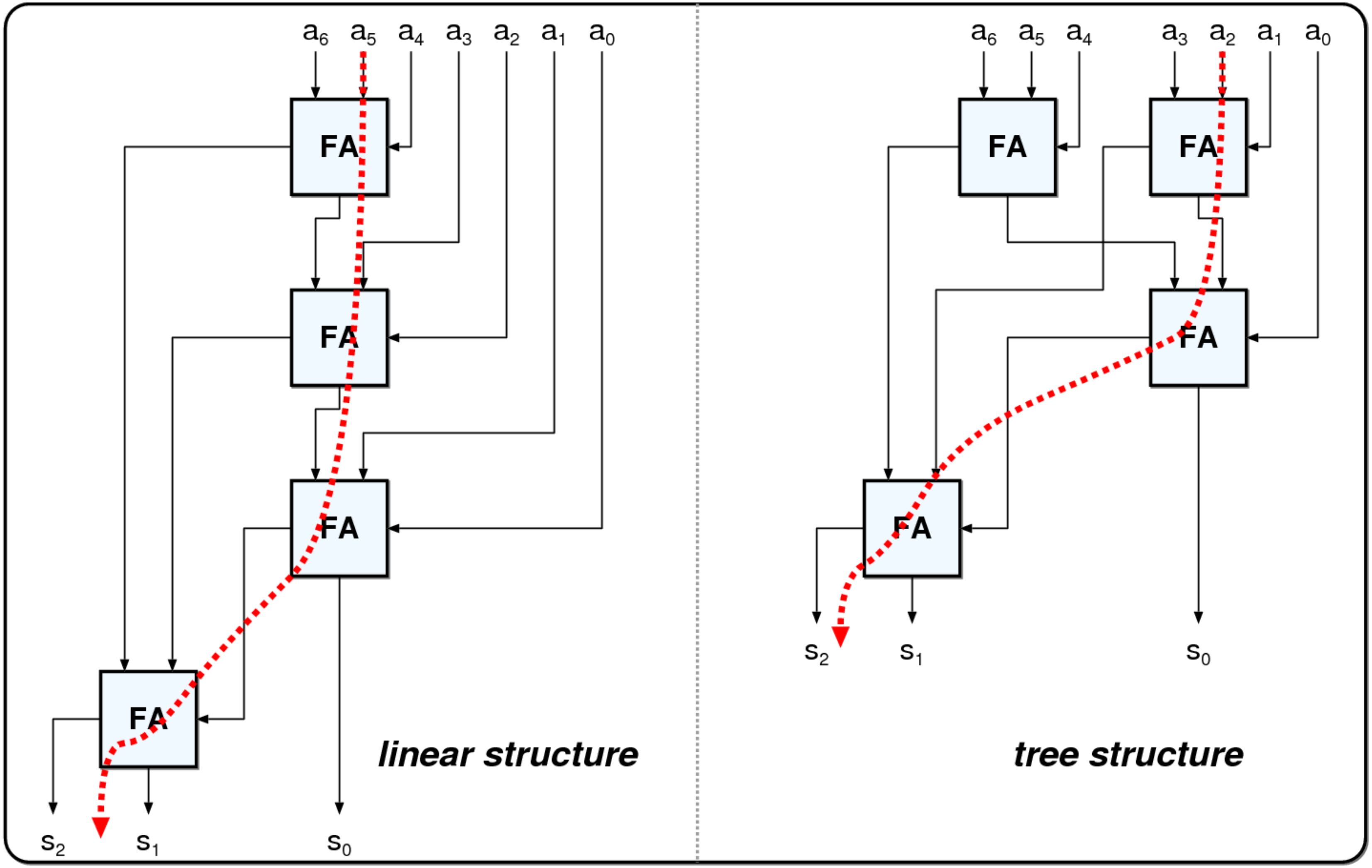


Full-Adder (3,2) Counter

- The Full Adder (FA) is the ***essential*** arithmetic block
- It can add three 1-bit numbers, result is a 2-bit number
- There are many realizations both at gate and transistor level.
- Since it is used in building many arithmetic operations, the performance of one FA influences the overall performance greatly.



Adding Multiple 1-bit Numbers



Adding Multiple Digits

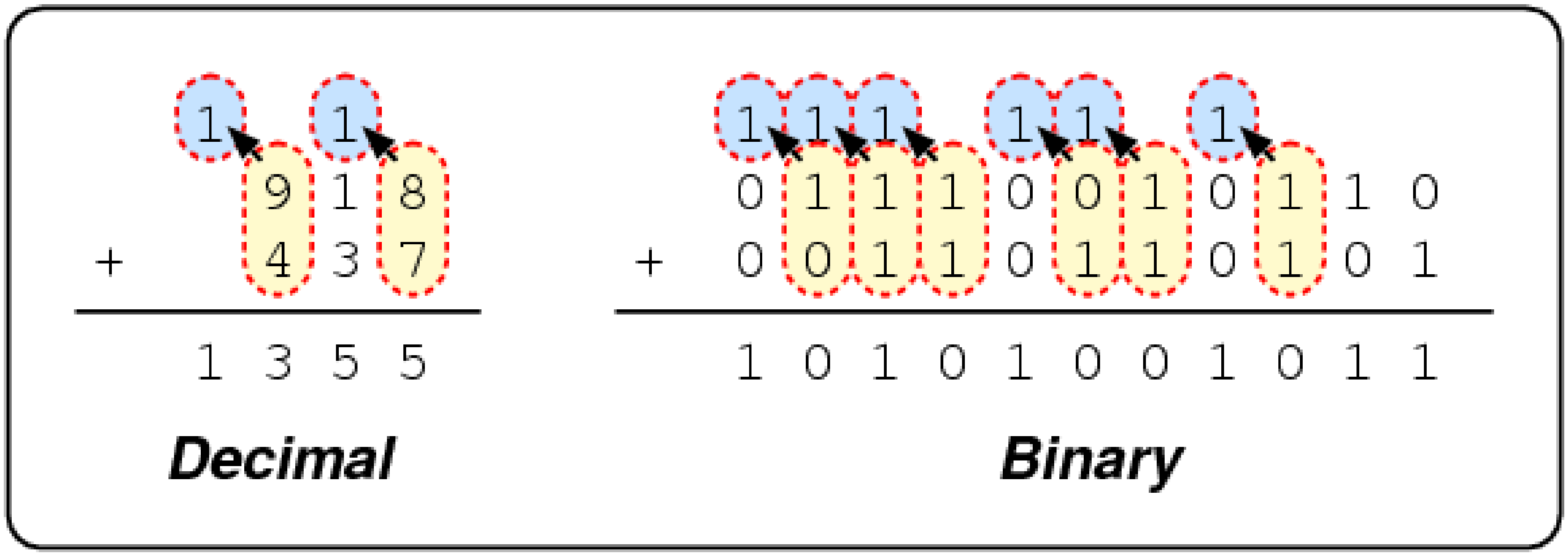
- **Similar to decimal addition**
- **Starting from the right, each digit is added**
- **The carry from one digit is added to the digit to the left**

```
      9 1 8          0 1 1 1 0 0 1 0 1 1 0
+     4 3 7      +   0 0 1 1 0 1 1 0 1 0 1
-----------      -------------------------
    1 3 5 5          1 0 1 0 1 0 0 1 0 1 1
```

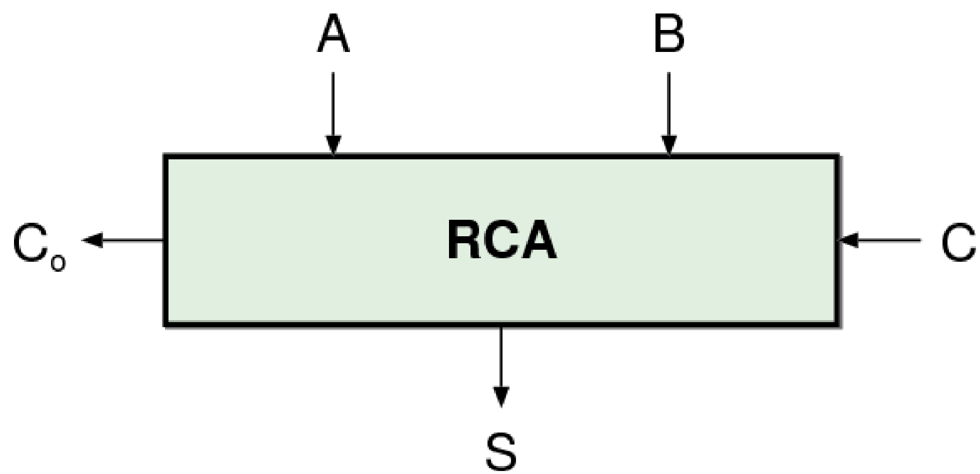
Decimal ***Binary***

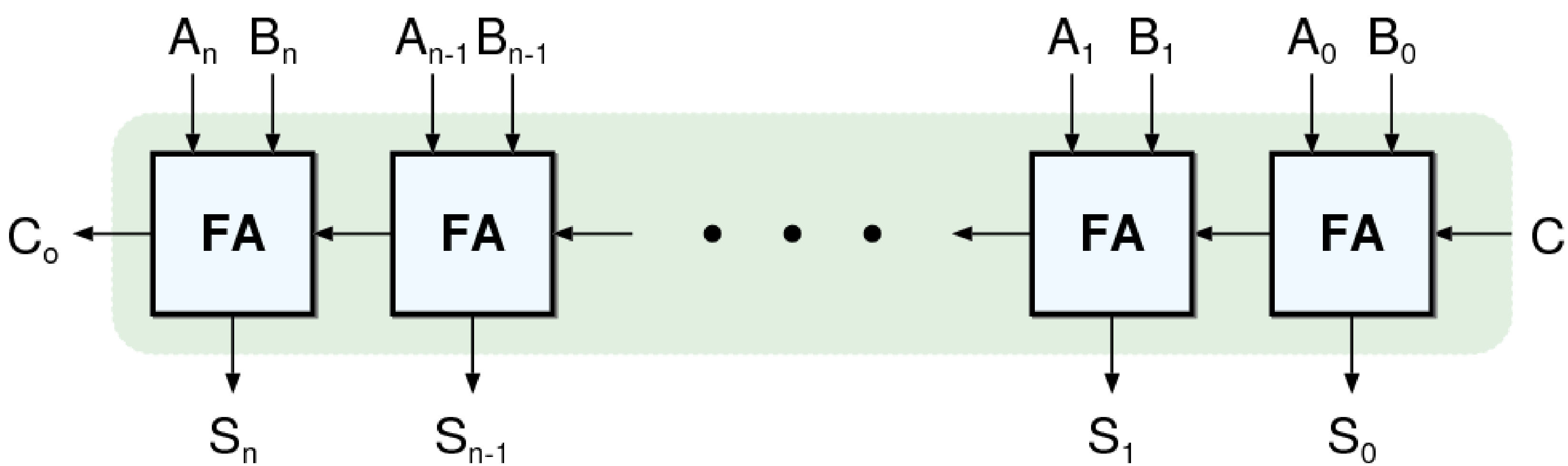
Adding Multiple Digits

- Similar to decimal addition
- Starting from the right, each digit is added
- The carry from one digit is added to the digit to the left



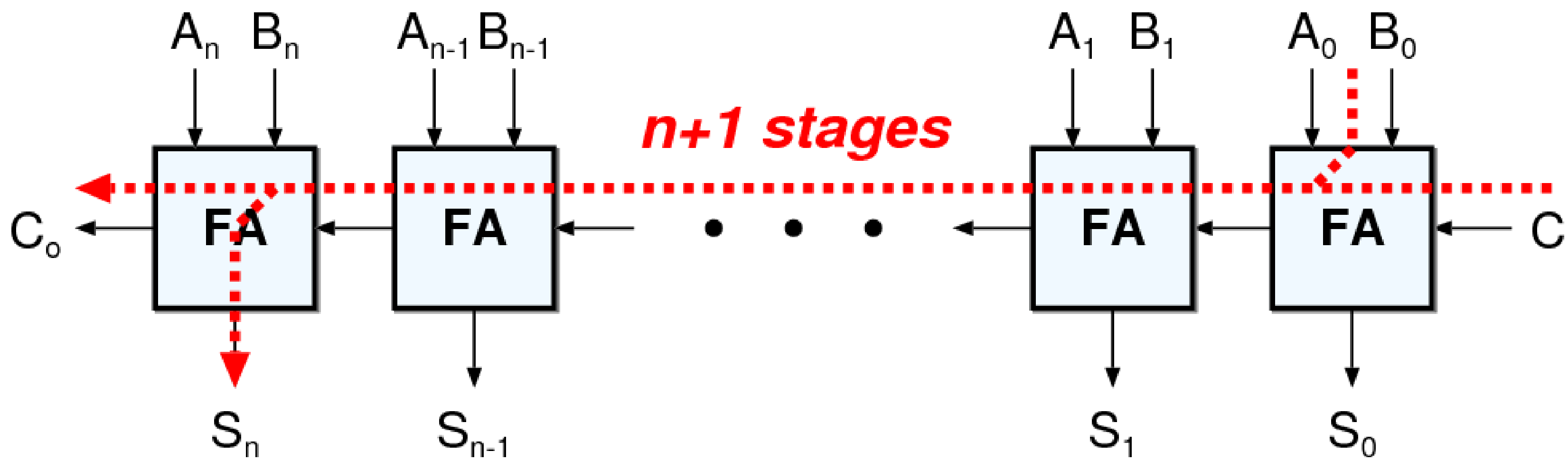
Ripple Carry Adder (RCA)





Curse of the Carry

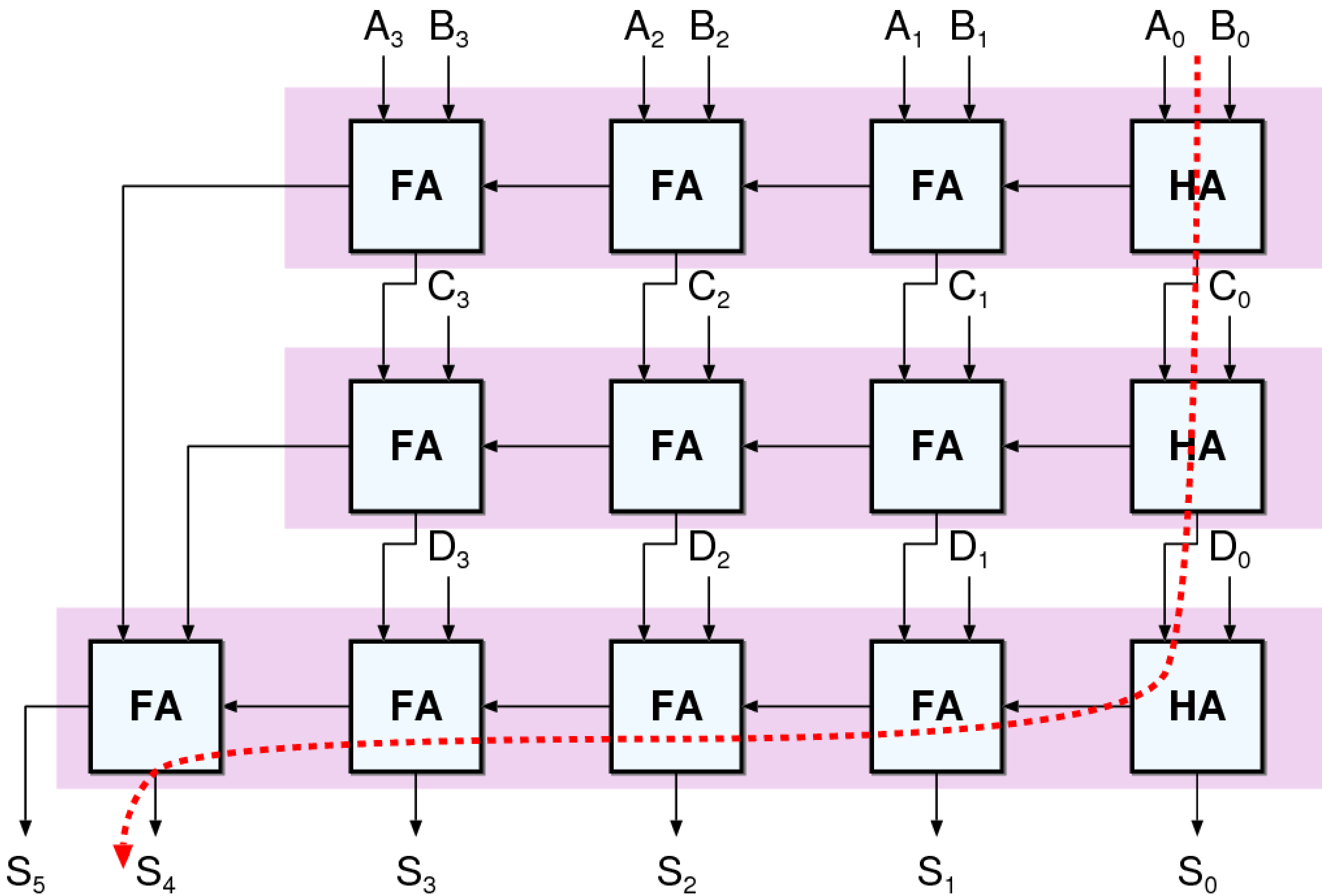
The most significant outputs of the adder depends on the least significant inputs



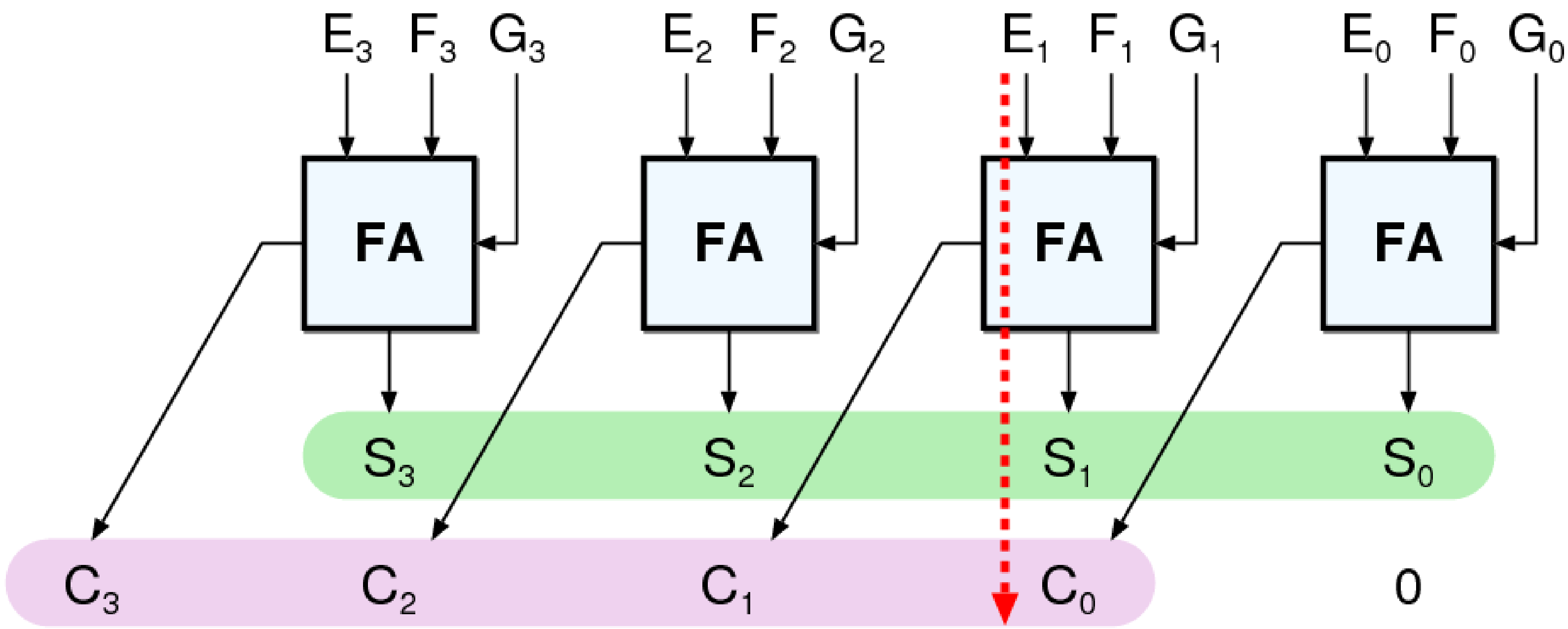
Adding Multiple Numbers

- **Multiple fast adders not a good idea**
  - If more than 2 numbers are to be added, multiple fast adders are not really efficient
- **Use an array of ripple carry adders**
  - Popular and efficient solution
- **Use carry save adder trees**
  - Instead of using carry propagate adders (the adders we have seen so far), ***carry save adders*** are used to reduce multiple inputs to two, and then a single carry propagate adder is used to sum up.

Array of Ripple Carry Adders



Carry Save Principle



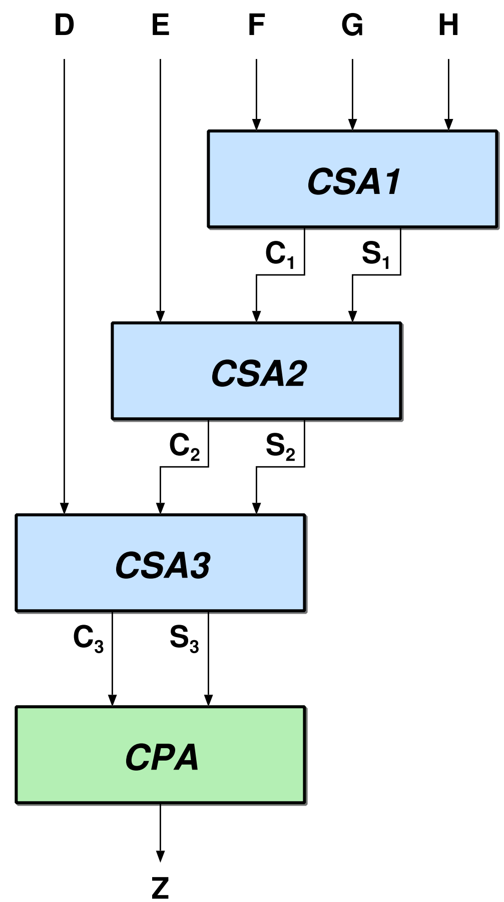
- Reduces three numbers to two with a single gate delay

$$C + S = E + F + G$$

Carry Save Principle

$Z = D + E + F + G + H$

- An array of carry save adders reduce the inputs to two
- A final (fast) carry propagate adder (CPA) merges the two numbers
- Performance mostly dictated by CPA



Multipliers

- **Largest common arithmetic block**
  - Requires a lot of calculation
- **Has three parts**
  - Partial Product Generation
  - Carry Save Tree to reduce partial products
  - Carry Propagate Adder to finalize the addition
- **Adder performance (once again) is important**
- **Many optimization alternatives**

Decimal Multiplication

```
        2 4 1 7
    ×   1 4 0 3
    -----------
        7 2 5 1
      0 0 0 0
    9 6 6 8
+ 2 4 1 7
---------------
  3 3 9 1 0 5 1
```

Partial Products

Binary Multiplication

									0	1	0	1	1	0	0	1	*89*
								×	0	0	1	1	0	1	1	1	*55*
									0	1	0	1	1	0	0	1	
								0	1	0	1	1	0	0	1		
							0	1	0	1	1	0	0	1			
						0	0	0	0	0	0	0	0				
					0	1	0	1	1	0	0	1					
				0	1	0	1	1	0	0	1						
			0	0	0	0	0	0	0	0							
+		0	0	0	0	0	0	0	0								
	0	0	0	1	0	0	1	1	0	0	0	1	1	1	1	1	*4895*

Partial Products

For n-bit Multiplier m-bit Multiplicand

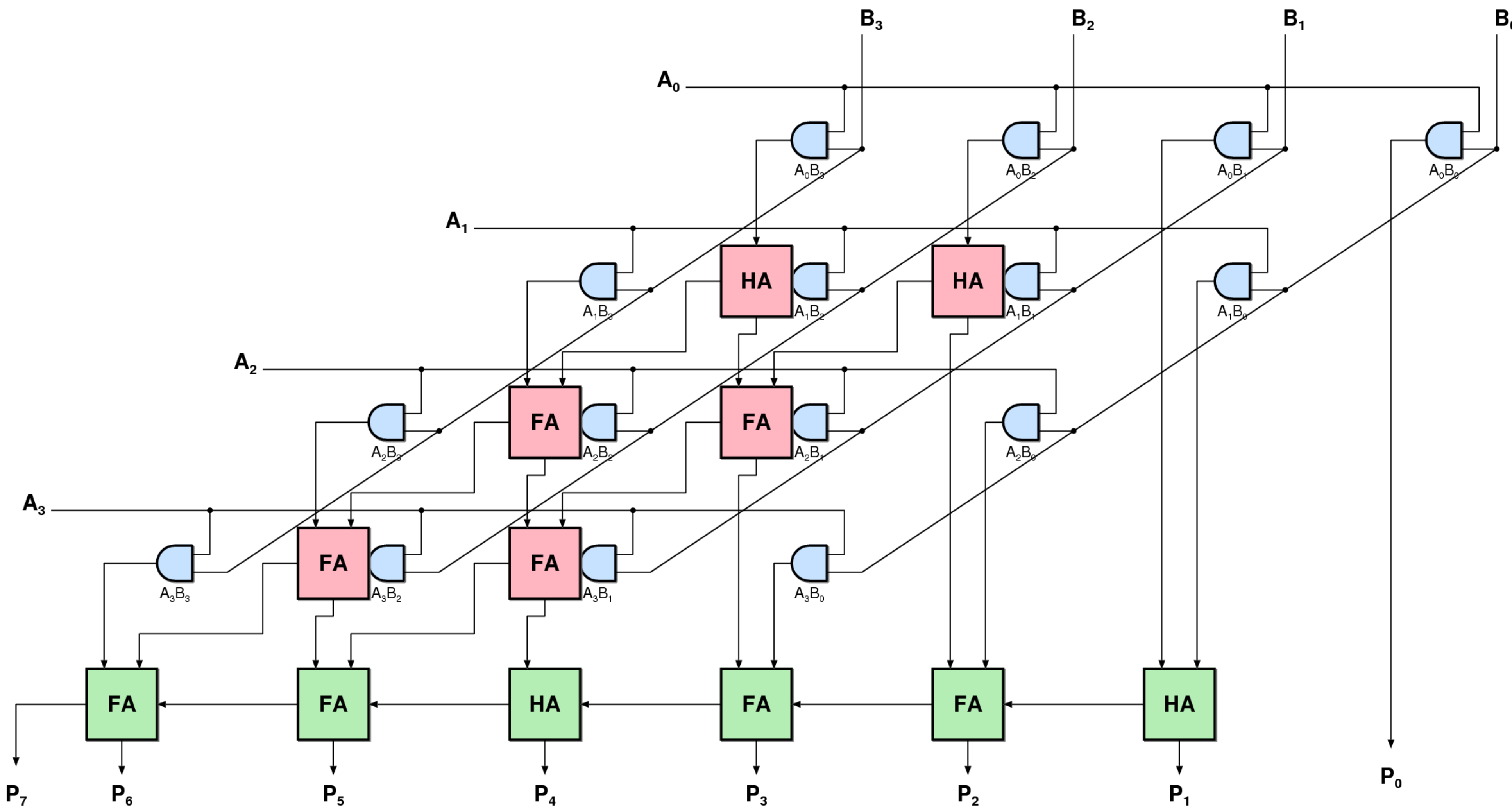
- **Generate Partial Products**
  - For each bit of the multiplier the partial product is either
    - when ‘0’: all zeroes
    - when ‘1’: the multiplicand

    achieved easily by AND gates

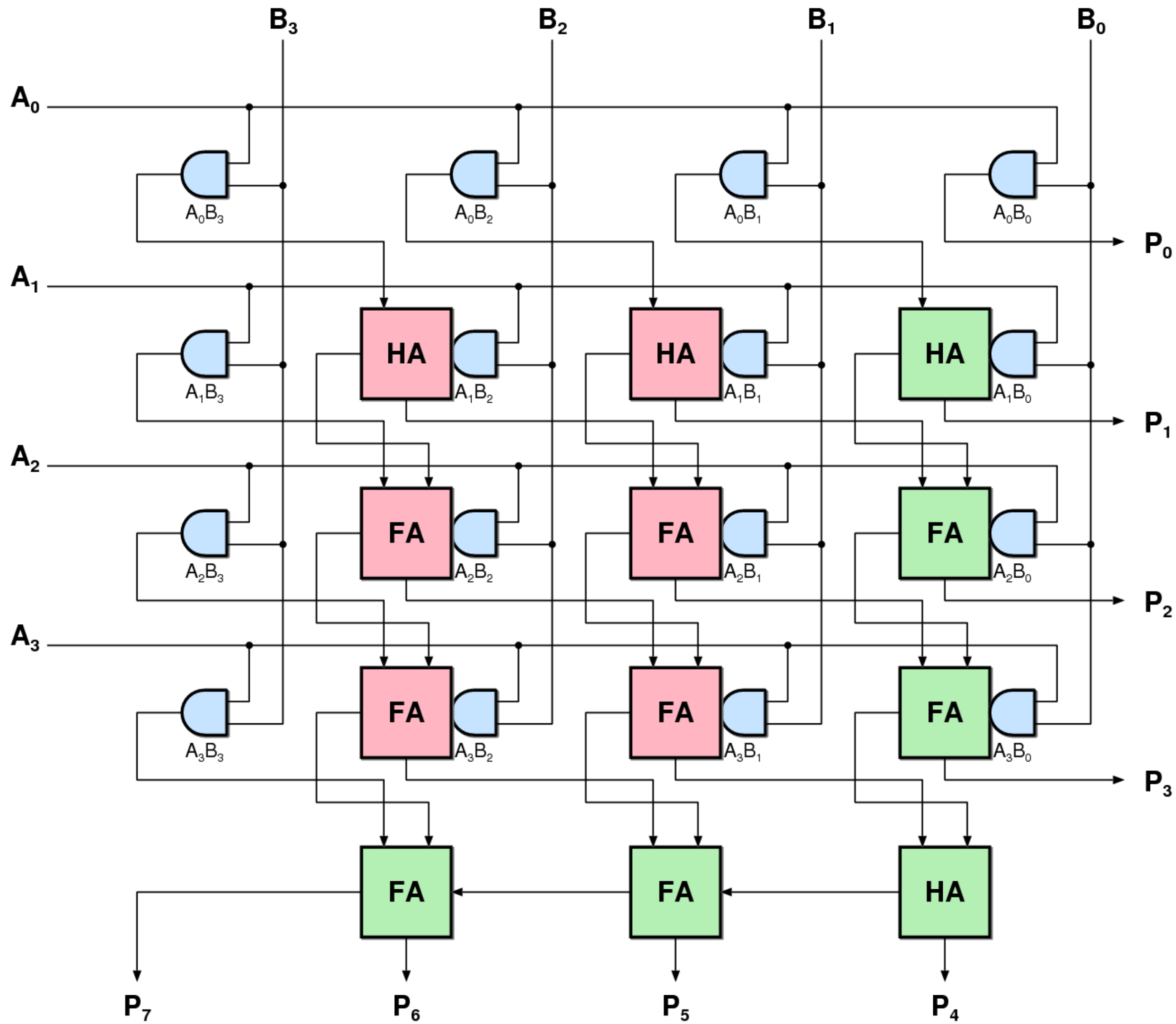
- **Reduce Partial Products**
  - This is the job of a carry save adder

- **Generate the Result (n + m bits)**
  - This is a large, fast Carry Propagate Adder

Parallel Multiplier



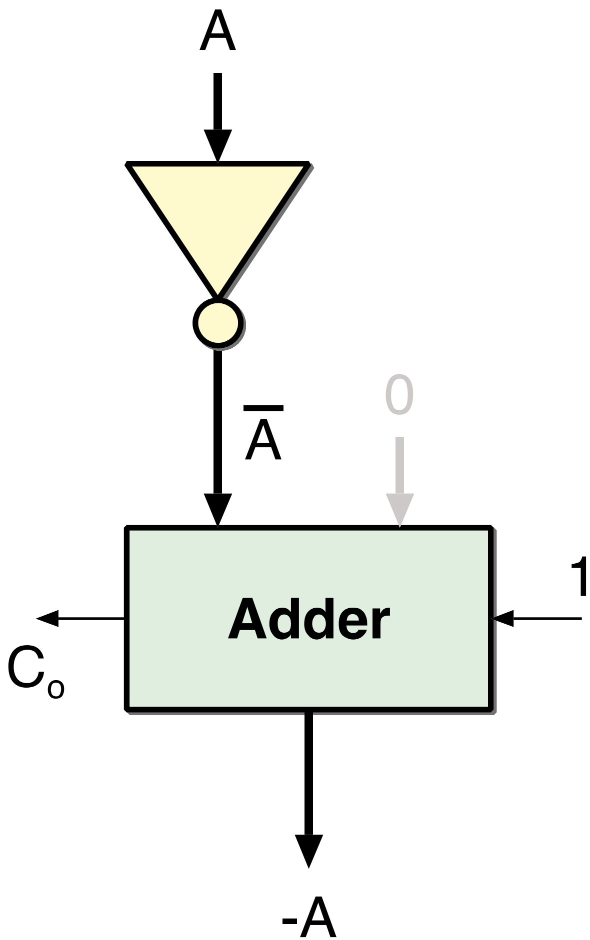
Parallel Multiplier



Operations Based on Adders

- **Several well-known arithmetic operation are based on adders:**
  - Negator
  - Incrementer
  - Subtracter
  - Adder Subtracter
  - Comparator

Negating Two's Complement Numbers

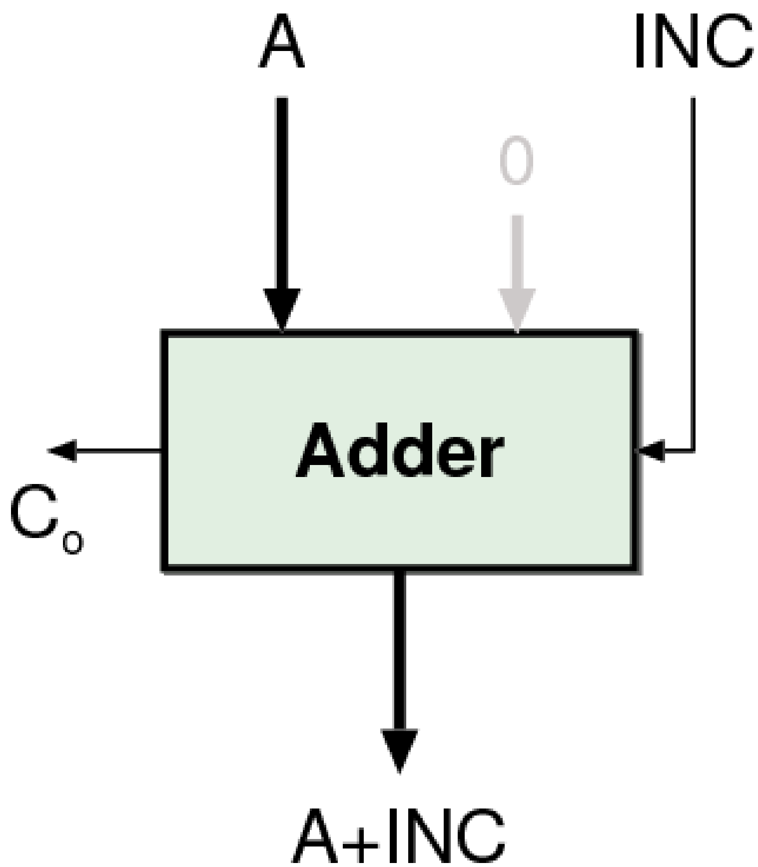


- To negate a two's complement number

$$-A = \overline{A} + 1$$

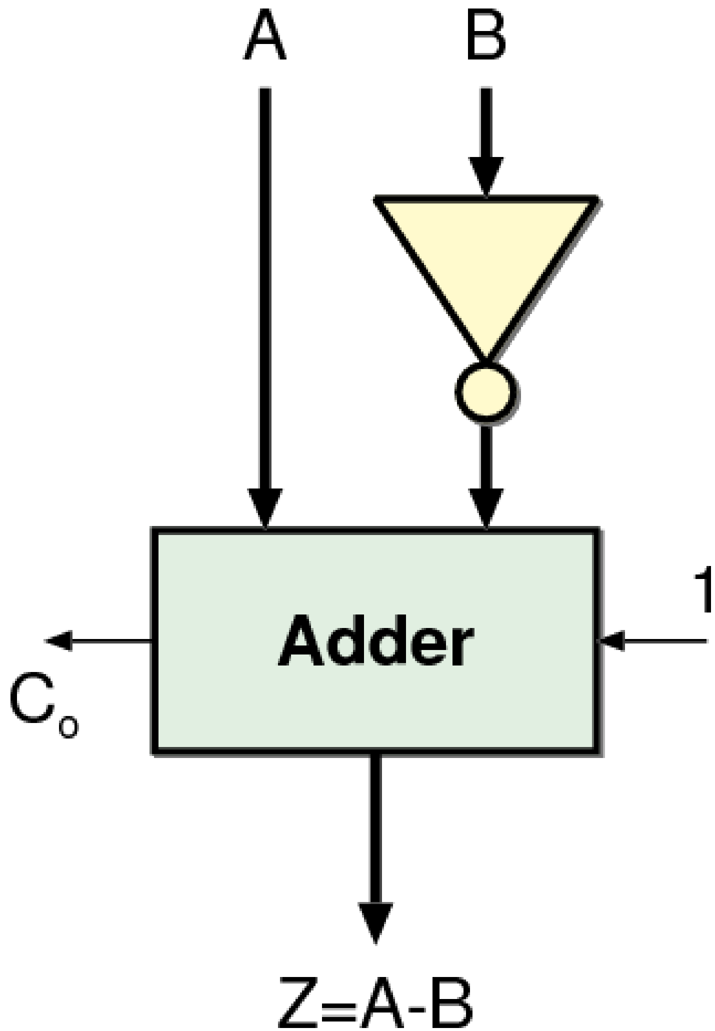
- All bits are inverted
- One is added to the result
- Can be realized easily by an adder.
- B input is optimized away

Incrementer



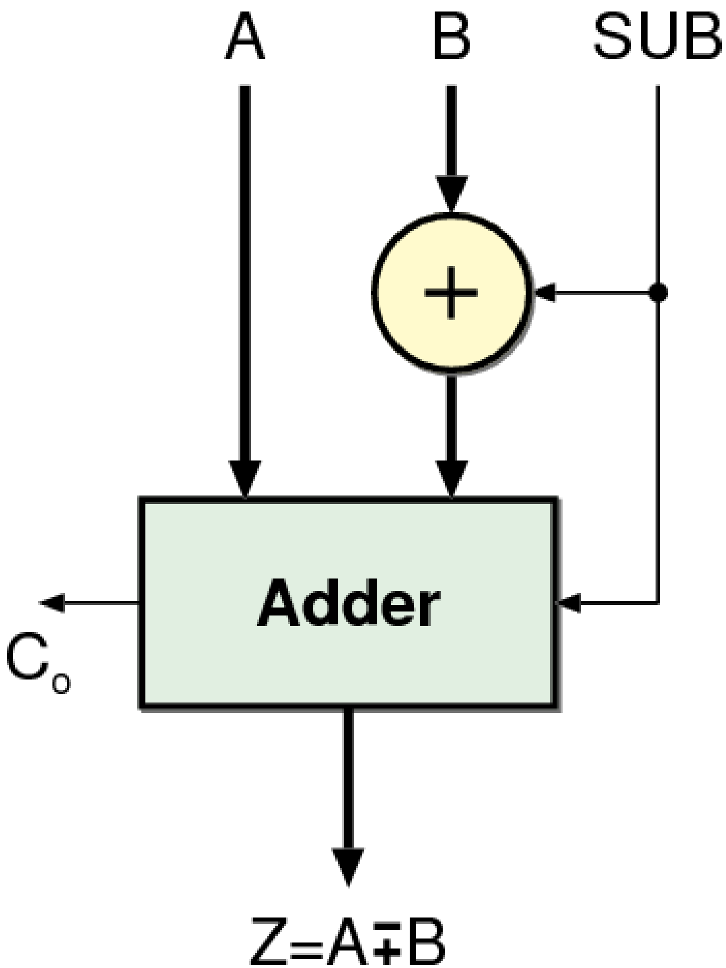
- B input is zero
- Carry In (C_{in}) of the adder can be used as the Increment (Inc) input
- Decrementer similar in principle

Subtracter



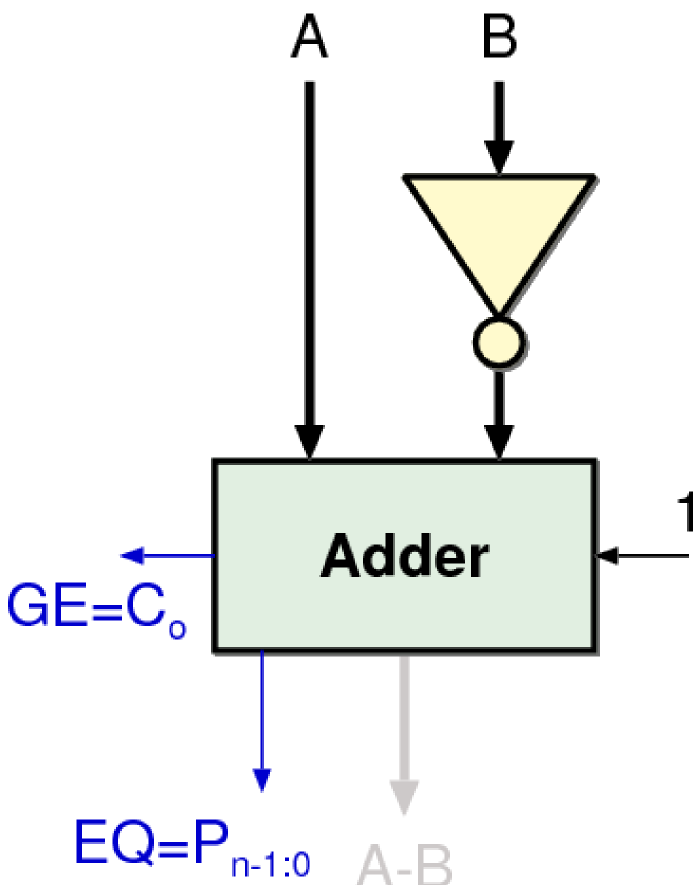
- B input is inverted
- C_{in} of the adder is used to complement B

Subtracter



- B input is inverted
- C_{in} of the adder is used to complement B
- It can be made programmable so that both additions and subtractions can be performed at the same time

Comparator



- **Based on a Subtractor**

$(A = B) = EQ$

$(A \mathrel{!=} B) = \overline{EQ}$

$(A > B) = GE\ \overline{EQ}$

$(A >= B) = GE$

$(A < B) = \overline{GE}$

$(A <= B) = \overline{GE} + EQ$

Functions Realized Without Adders

- **Not all arithmetic functions are realized by using adders**
  - Shift / Rotate Units
- **Binary Logic functions are also used by processors**
  - AND
  - OR
  - XOR
  - NOT

These are implemented very easily

Shifters

- **Logical shifter: shifts value to left or right and fills empty spaces with 0's**
  - `Ex: 11001 >> 2 = ??`
  - `Ex: 11001 << 2 = ??`

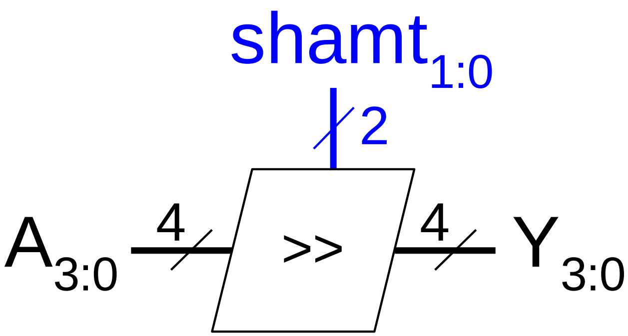
- **Arithmetic shifter: same as logical shifter, but on right shift, fills empty spaces with the old most significant bit (msb).**
  - `Ex: 11001 >>> 2 = ??`
  - `Ex: 11001 <<< 2 = ??`

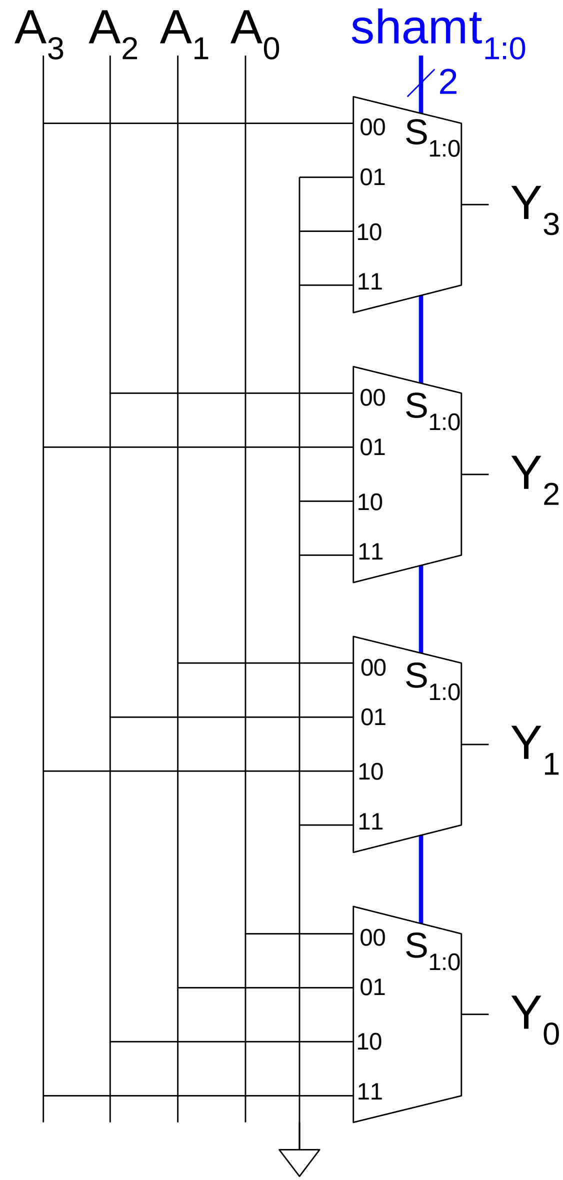
- **Rotator: rotates bits in a circle, such that bits shifted off one end are shifted into the other end**
  - `Ex: 11001 ROR 2 = ??`
  - `Ex: 11001 ROL 2 = ??`

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





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` ($-16 \div 2^2 = -4$)

Other Functions

- **We have covered 90% of the arithmetic functions commonly used in a CPU**

- **Division**
  - Dedicated architectures not very common
  - Mostly implemented by existing hardware (multipliers, subtractors comparators) iteratively

- **Exponential, Logarithmic, Trigonometric Functions**
  - Dedicated hardware (less common)
  - Numerical approximations:

    $\exp(x) = 1 + x^2/2! + x^3/3! + \dots$
  - Look-up tables (more common)

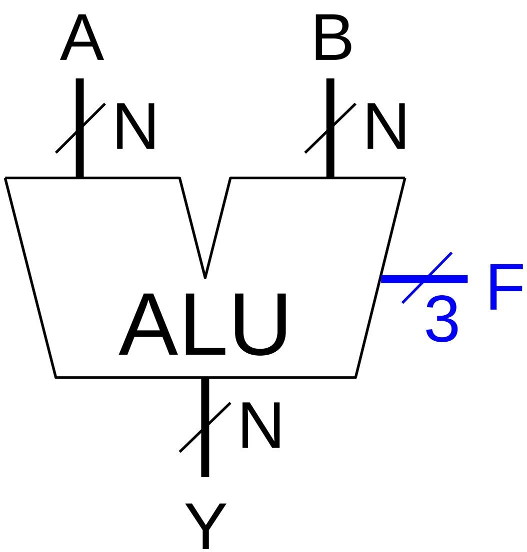
Arithmetic Logic Unit

The reason why we study digital circuits:
the part of the CPU that does something (other than copying data)

- **Defines the basic operations that the CPU can perform directly**
  - Other functions can be realized using the existing ones iteratively. (i.e. multiplication can be realized by shifting and adding)

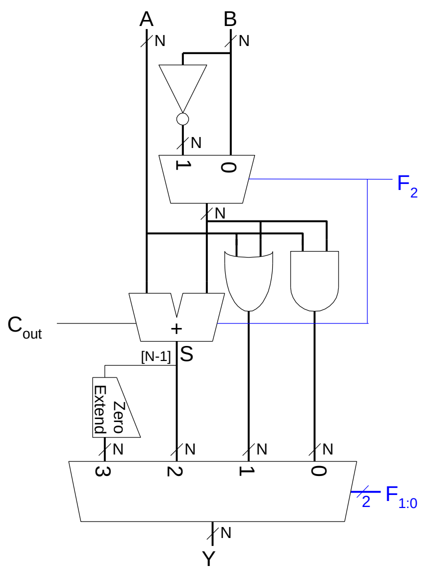
- **Mostly, a collection of resources that work in parallel.**
  - Depending on the operation one of the outputs is selected

Example: Arithmetic Logic Unit (ALU), pg243



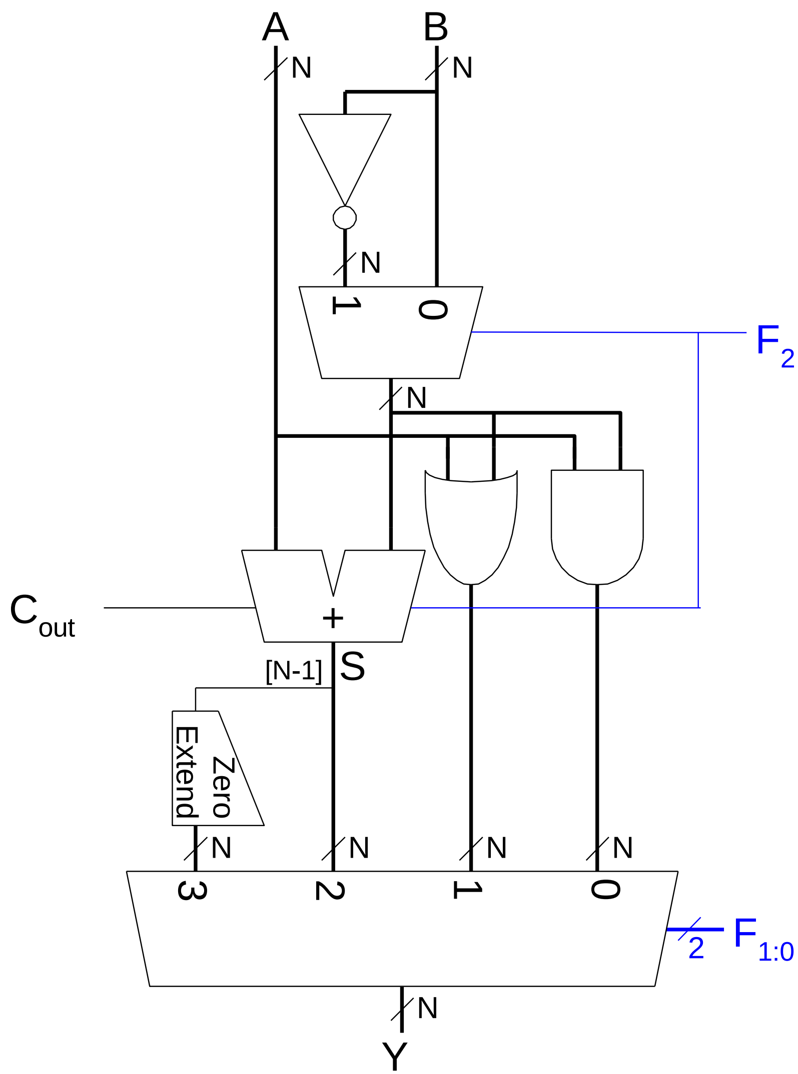
$F_{2:0}$	Function
000	A & B
001	A | B
010	A + B
011	not used
100	A & ~B
101	A | ~B
110	A - B
111	SLT

Example: ALU Design



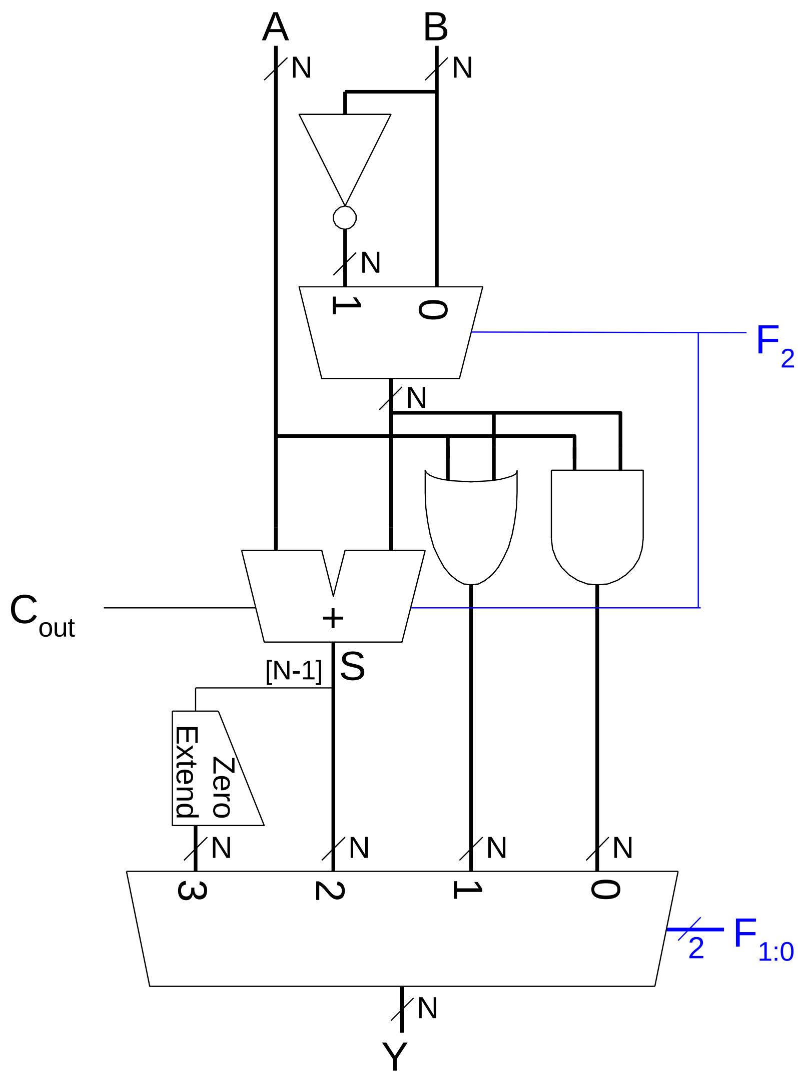
$F_{2:0}$	Function
000	A & B
001	A | B
010	A + B
011	not used
100	A & ~B
101	A | ~B
110	A - B
111	SLT

Set Less Than (SLT) Example



- **Configure a 32-bit ALU for the set if less than (SLT) operation. Suppose A = 25 and B = 32.**
  - A is less than B, so we expect Y to be the 32-bit representation of 1 (`0x00000001`).

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`.
  - F2 = 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`

What Did We Learn?

- **How can we add, subtract, multiply binary numbers**
- **What other circuits depend on adders**
  - Subtracter
  - Incrementer
  - Comparator
  - Important part of Multiplier
- **Other functions (shifting)**
- **How is an Arithmetic Logic Unit constructed**