Chapter 06:Computer Arithmetic

Hari K.C.

Department of Software Engineering

Gandaki College of Engineering and Science

Chapter 6: Arithmetic Unit.

The arithmetic unit is a part of processor that executes arithmetic operations. A microprocessor is capable of processing numeric data in one or more formats.

* **Unsigned notation** (unsigned representation of numbers)

⇒ In this notation, there is no any separate bit to represent the sign (positivity or negativity) of the number.

Signed representation of 8 bit data

e.g.:-

11111

Signed = -7

1 1001010

sign magnitude.

But in unsigned representation, the whole 8 bit represent data.

unsigned = 15

The unsigned notations may be non-negative notations and 2's complement format.

✓ In non-negative notation, every number is treated as zero or a positive value. An n-bit number can have a value ranging from 0 (all bits are zero) to $2^n - 1$ (all bits are 1)

✓ In 2's complement format, both positive & negative numbers can be represented. An n-bit number can have range from -2^{n-1} to 2^{n-1}

$2\overline{)127}$ 1
$2\overline{)63}$ 1
$2\overline{)31}$ 1
$2\overline{)15}$ 1
$2\overline{)7}$ 1
3 1
1

0 1 1 1 1 1 1 1

Negative numbers have a 1 as the most significant bit and positive numbers (& zero) have 0 as leading bit.

Binary representation	Unsigned non-negative	Unsigned 2's complement
0000 0000	0	0
0000 0001	1	1
0111 1111	127	127
1000 0000	128	-128
1000 0001	129	-127
1111 1110	254	-2
1111 1111	255	-1

$127 = 0\times2^7 + 1\times2^6 + 1\times2^5 + 1\times2^4 + 1\times2^3 + 1\times2^2 + 1\times2^1 + 1\times2^0 = 127.$

Addition & subtraction:

straightforward

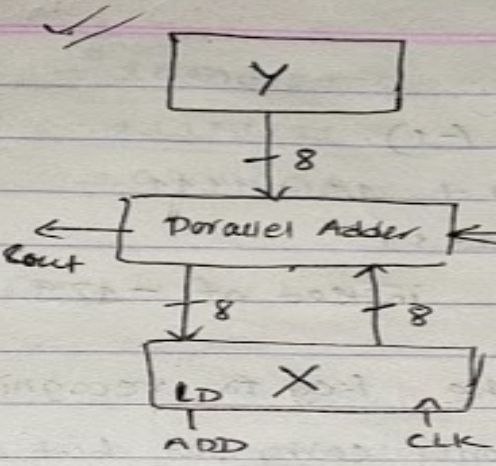
Addition is realized by using a parallel adder.

ADD: $X \leftarrow X + Y$

Here,

X & Y are 8 bit registers

0 – 255
can be represented by 8 bit
$2^8 = 256$



128
1000 0000
2's comp = 0111 1111 + 1 = 1000 0000

129
0111 1110 + 1 = 0111 1111

This circuit works as long as the result is 8-bit. But when

$$255 + 1$$
$$= 1111\ 1111 + 0000\ 0001$$
$$= 1\ 0000\ 0000 \Rightarrow \text{9 bit value}$$

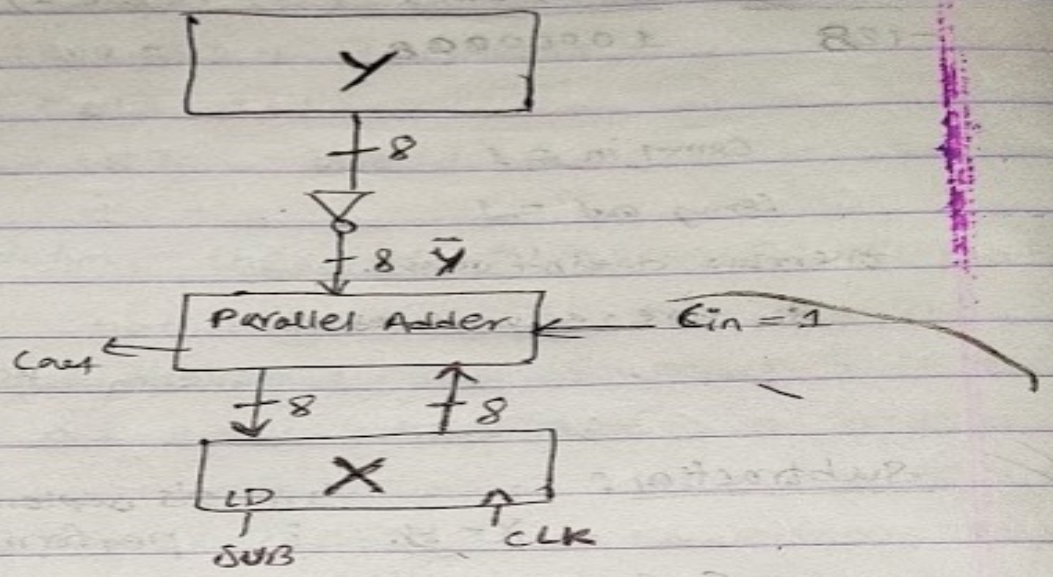
The 9 bit value cannot be stored in 8-bit register. The extra bit is Cout which indicates arithmetic overflow. This bit can set an overflow flag.

In 2's complement notation, an overflow can occur at either ~~side~~ end of numeric range. At positive end, adding

$$127 + 1$$
$$0111\ 1111 + 0000\ 0001$$
$$= 1000\ 0000$$

. In 2's complement notation, this value is -128 not +128.

SUB: $X \leftarrow X - Y$



$2 \rightarrow$ 0000 0010

$(-1) \rightarrow$ 1111 1111

1 0000 0001

255 1111 1111

-254 0000 0010

1 0000 0001

1 0000 0001

-2 + 1111 1110

0000 0000

0111 1111

Multiplication :

Let x = 27

y = 253

```
   27
  253
  ---
   81
 135 *
 54 x x
 ----
 6831
```

⇒ shift add multiplication

shifting left

This normal calculation is calculated as

~~*253~~

~~81~~

~~81~~ ← ~~shifted one position right~~

```
   27
  253
  ---
   81
```

shifting right

81 ← shifted one position right

```
+ 135
  ----
  1431   ← 1 is carried down, not added
```

~~81~~431 ← shifted one position right

```
 +54
 ---
 6831   ← 31 is carried down, not added
```

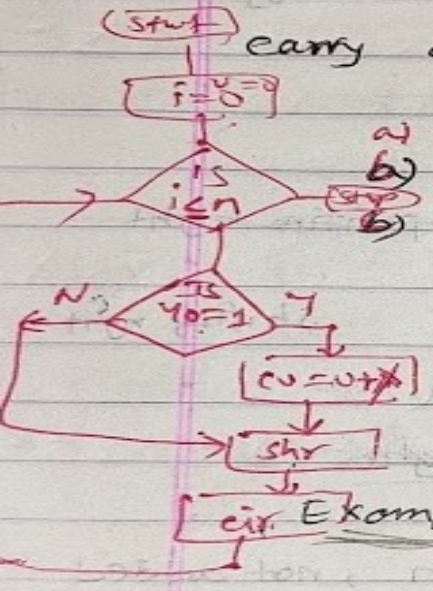
6831 one final shift

* Shift-add multiplication algorithm

⇒ Let X & Y be two n-bit registers (Input) contents.

The result is stored in two n-bit register, U & V. U contains high order half of result & V contains low-order half.

C is 1-bit register used to store carry of an addition.



steps:

a) start

b) Initialize U=0, V=0

b) perform i=1 to n

i) if $Y_0 = 1$ Then (C,U) = U + X

ii) linear shift right (CU.V)

iii) circular shift right (Y)

Example:- X = 13 * Y = 11

1101 * 1011 → Y_0

check two Y_0

Function	i	C	U	V	Y	Comments
when U=0			0000	0000	1011	
if $Y_0=1$	1	0	1101			C,U=U+X
shr (CUV)		0	0110	1000		Y_0=1, add
cir (Y)					1101	Y_0

```
        0110
      + 1101
      -------
      1 0011
      C  U
```

if $Y_0 = 0$, do not perform $CU = U + X$

since
if $Y_0 = 1$ 2 1 0011 — $Y_0 = 1$, add $CU = U + X$
shr(CUV) 0 1001 1100
cir(Y) 1110 — $Y_0 = 0$

since $Y_0 = 0$
here $Y_0 = 0$
if $Y_0 = 1$ 3 — now $Y_0 = 0$
shr(CUV) 0 0100 1110 — $Y_0 = 0$
cir(X) 0111

since
if $Y_0 = 1$ 4 1 0001 — $Y_0 = 1$, $CU = U + X$
shr(CUV) 1000 1111
cir(Y) 1011 original value
DONE 1000 1111 Result = 143
U V

RTL code for this algorithm

1: $U \leftarrow 0$, $i \leftarrow n$
Y_0 2: $CU \leftarrow U + X$
2: $i \leftarrow i - 1$
3: shr(CUV) , cir(Y)
2'3: GOTO 2
Z3: FINISH $\leftarrow 1$.

1101 → 1's complement 0010 + 1 = 0011 = -3.

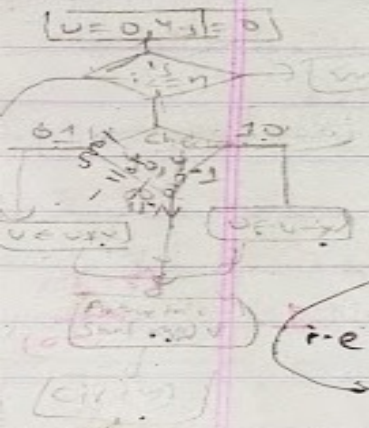
Booth Algorithm

- speed up multiplication

Previous algorithm doesn't work for negative numbers.

For example: (13) 1101 & (11) 1011 would represent -3 & -5 respectively in 2's complement notation

Booth algorithm directly works on two's complement numbers, eliminating conversion bet'n positive & negative numbers.



Steps:-

i) Start

ii) Initialise $U = 0$; $Y_{-1} = 0$ (1 bit register)

iii) Perform for $i = 1$ to n

a) if start of a string of 1's in Y then $U = U - X$
i.e Y_0, Y_{-1} = 1 0

if end of a string of 1's in Y then $U = U + X$
i.e Y_0, Y_{-1} = 0 1

b) Arithmetic shift right (UV)

c) circular shift right (Y) AND copy Y_0 to Y_{-1}.

$$\begin{array}{r} 0000 \quad U \\ -\ 1101 \quad X \\ \hline 0011 \end{array}$$

$X = -3 \quad Y = -5$

$= 1101 \quad 1011$

$-3 \times -5 = 15$

Trace of Booth's algorithm

Function	i (loop)	U	V	Y	Y_{-1} (1 bit)	Comments
U = 0		0000	0000	1011	0	
$Y_{-1} = 0$				Y_0		
if start of string	1	0011				U = U − X
if end of string						start of string
ashr (UV)		0001	1000			
cir (Y), $Y_{-1} \leftarrow Y_0$				1101	1	
				Y_0		
if start of string	2					still within string
if end of string						
ashr (UV)		0000	1100			
cir (Y), $Y_{-1} \leftarrow Y_0$				1110	1	
if start of string	3					U = U + X
if end of string		1101				End of string
ashr (UV)		1110	1110			
cir (Y), $Y_{-1} \leftarrow Y_0$				0111	0	
if start of string	4	0001				U = U − X
if end of string						start of string
ashr (UV)		0000	1111			
cir (Y), $Y_{-1} \leftarrow Y_0$				1011	1	original value
DONE		0000	1111			Result = +15

$\gg 15$

ashr → arithmetic shift right

1100 → 1110

1 1 0 0 (n)

RTL code for Booth algorithm

1: $U \leftarrow 0$, $Y_{-1} \leftarrow 0$, $i \leftarrow n$

$Y_0 Y_{-1}'$ 2: $U \leftarrow U + \bar{X} + 1$ } $U \leftarrow U - X$

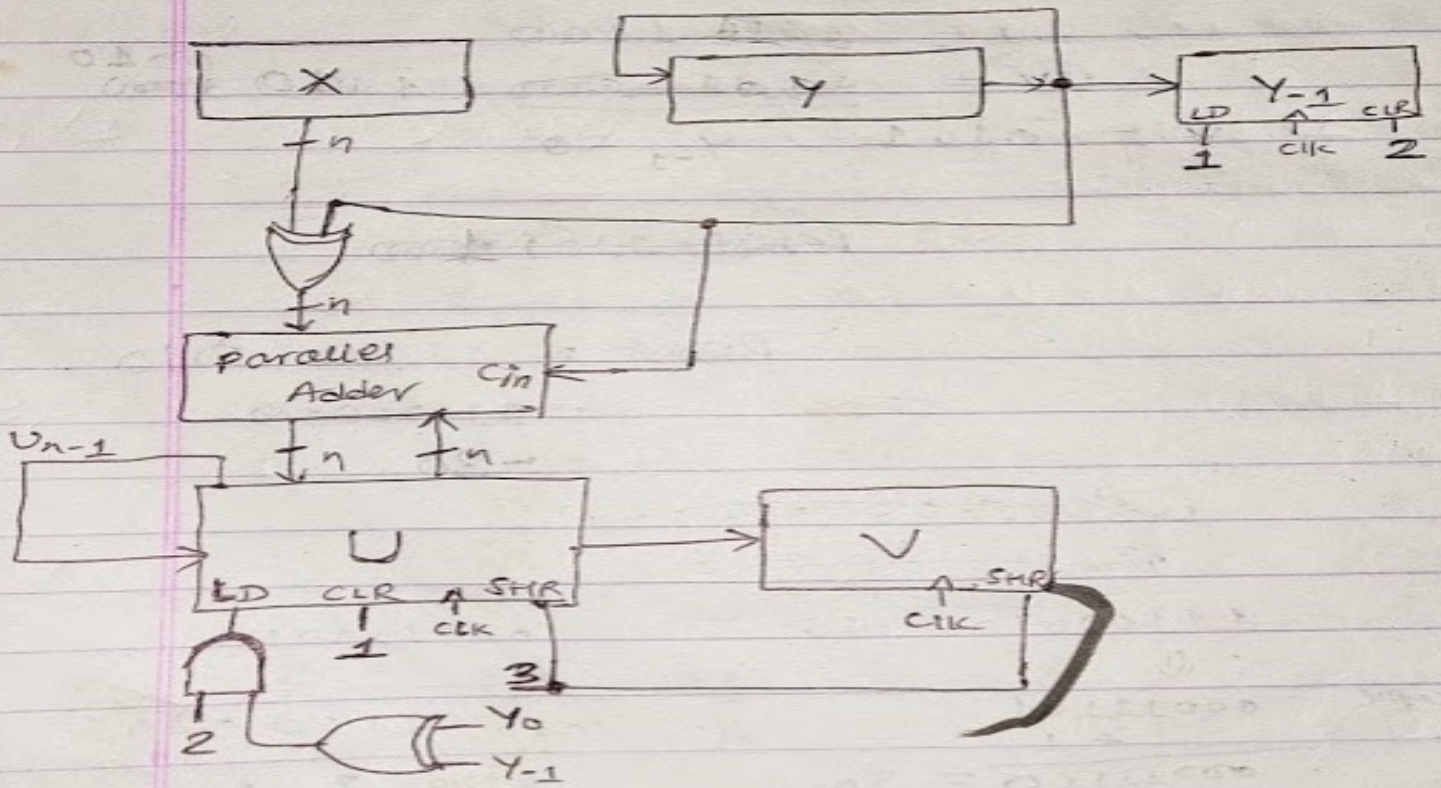
$Y_0' Y_{-1}$ 2: $U \leftarrow U + X$ } $(Y_0 \oplus Y_{-1})$ 2: $U \leftarrow U + (X \oplus Y_0) + Y_0$

2: $i \leftarrow i - 1$

3: ashr (UV), cir (Y), $Y_{-1} \leftarrow Y_0$

z' 3: GOTO 2

z3: FINISH $\leftarrow 1$



2|147
2|73 1
2|36 1
2|18 0
2|9 0
2|4 1
2|2 0
1 0

10010011

* Division.

shift subtract division algorithm

The dividend is initially stored in UV. where U & V are n bits register. The divisor is stored in X register. The quotient in Y register. The remainder will be stored in U. C is a 1-bit register to store extra bit shifted out of U.

Steps:-

i) Start
ii) if U ≥ X then exit with overflow.
iii) Initialize Y = 0 & C = 0;
iv) Perform for i = 1 to n (= no. of bits)
{ a) linear shift left (C,U,V);
b) linear shift left (Y);
c) If (U ≥ X) Then {Y₀ = 1, U = U - X}
}
(True)

Example:

147 ÷ 13

10010011 ÷ 1101

dividend. (U V)

divisor (X.)

2|147
7

$$\begin{array}{r} 10010 \\ -\ 1101 \\ \hline 0101 \end{array}$$

shl ← 0

$$\begin{array}{r} 10010 \\ -\ 1101 \\ \hline 0101 \end{array}$$

Trace of shift subtract division algorithm

i	Function	C	U	V	Y	Comments
	Initial	0	1001	0011	0000	
	if U ≥ X			←	←	U < X, No exit
				left	0	
1	shl(CUV)	1	0010	0110		
	shl(Y)				0000 → Y_0	
	if CU ≥ X		0101		0001	10010 ≥ 1101, $Y_0 = 1$, U = CU − X, 10010 − 1101 = 0101
2	shl(CUV)	0	1010	1100		
	shl(Y)				0010	
	if CU ≥ X		(no operation)			CU < X. 01010 < 1101
3	shl(CUV)	1	0101	1000		
	shl(Y)				0100	
	if CU ≥ X		1000		0101	10101 ≥ 1101, U = 10101 − 1101, U = 1000
4	shl(CUV)	1	0001	0000		
	shl(Y)				1010	
	if CU ≥ X		0100		1011	10001 ≥ 1101

0100 ← Remainder

1011 ← Quotient

$$\begin{array}{r} 10001 \\ -\ 1101 \\ \hline 0100 \end{array}$$

RTL code for shift subtract division algorithm

Here X, U, V and Y are n-bit values

C, & OVERFLOW are 1 bit values

1, 2, 3, 4 are sequence of states

1 : FINISH ← 1, OVERFLOW ← 1

G=1 if U≥X

2 : Y ← 0, C ← 0, OVERFLOW ← 0, i ← n

3 : shl (CUV), shl (Y), i ← i-1.

(C+G) 4 : $Y_0 \leftarrow 1$, $U \leftarrow U + \bar{X} + 1$

$\bar{Z}$4 : GOTO 3

Z4 : FINISH ← 1

The RTL code for the trace of the operation of $\overset{UV}{147} \div \overset{X}{13}$

U → 1001

V → 0011

X → 1101

n = 4

when i=0 then Z becomes 1.

* Signed Notations:

In signed notations, most significant bit represent sign of the numbers (0 for positive number & 1 for negative number) and others bit represent the magnitude.

+3 = 0 0011 (5 bit representation)

-3 = 1 0011

* Addition & subtraction of signed magnitude numbers:

BCD:

Binary coded decimal.

It is the most popular format to represent decimal data.

In BCD, each 4 bits represent one decimal digit. 0001 → 1, 1001 → 9

From values 0 to 9, their representations are same as in binary. 4 bit values above 9 are not used in BCD since they do not correspond to any decimal digit

Multidigit decimal numbers are stored as multiple groups of 4 bits per digit

e.g:- 11 23 (Decimal)

0001 0001 0010 0011 (BCD)

8 4 2 1

0101 = 8×0 + 4×1 + 2×0 + 1×1

= 4 + 1 = 5.

In BCD, $+14 = \underset{\text{Sign bit}}{0}\ 00010100$

$-14 = \underset{\text{Sign bit}}{1}\ 00010100$

Addition & Subtraction using BCD

As long as sum of two digits is no more than 9, a binary parallel adder will give correct result.

There are two situations when there will be invalid BCD result :-

i) $8 + 2 = 1000 + 0010 = 1010$

which is not a valid BCD o/p.

ii) $8 + 9 = \underset{\text{carry}}{1}\ 0001$ $\quad 1000 + 1001$

$= 10001$

not a correct result although it is a BCD value.

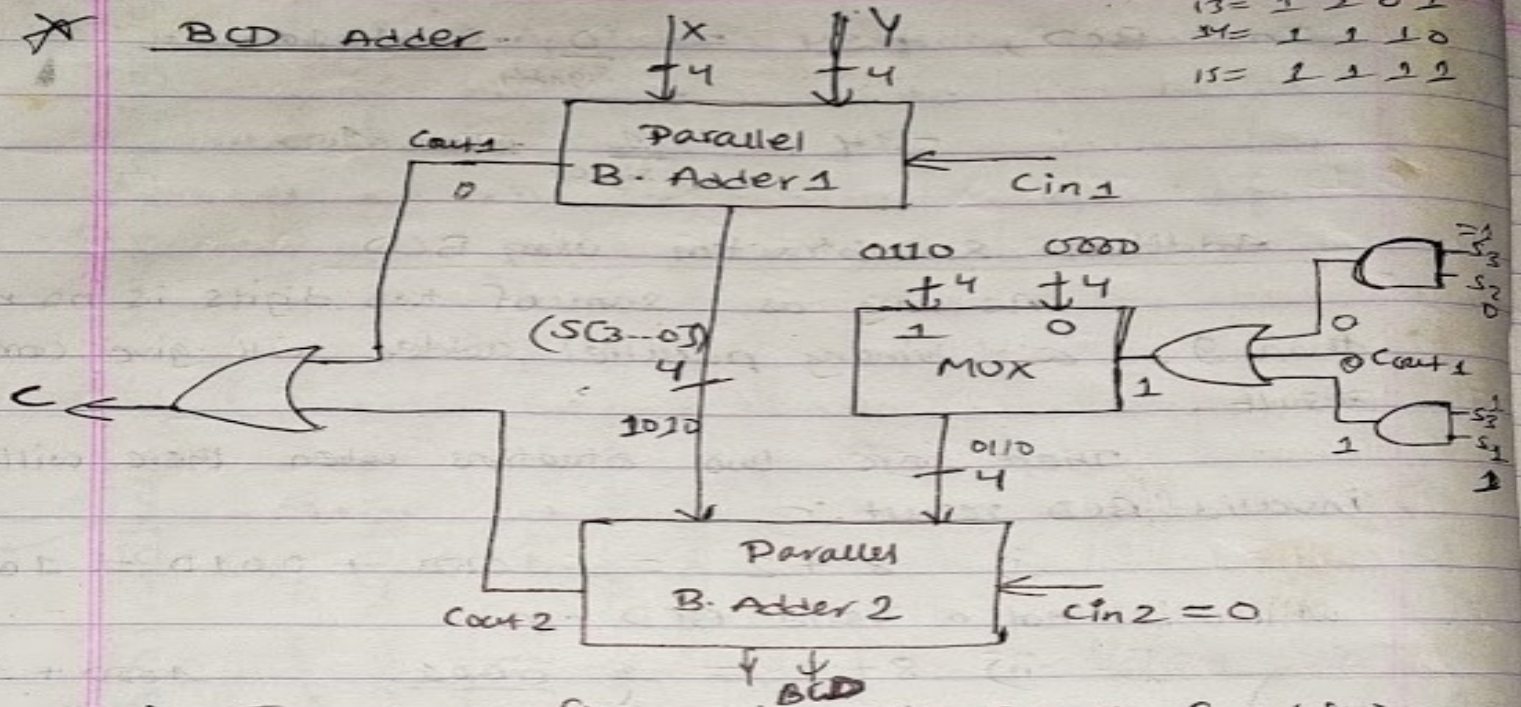
In both cases, adding 6 to the result generated by parallel binary adder produces correct BCD result.

i) $1010 + \underset{(6)}{0110} = \underset{\text{carry}}{1}\ 0000 \Rightarrow$ BCD (10)

ii) $0001 + 0110 = 0111$ BCD (7)

	S_3	S_2	S_1	S_0
10 =	1	0	1	0
11 =	1	0	1	1
12 =	1	1	0	0
13 =	1	1	0	1
14 =	1	1	1	0
15 =	1	1	1	1

BCD Adder



⇒ In above figure X & Y (each of 4 bits) are added together. Then this result is added to either 0 (0000) or 6 (0110) to produce the correct BCD sum. If the result $S_3S_2S_1S_0$ is not a valid BCD digit then either

$S_3 \wedge S_2 = 1$ or $S_3 \wedge S_1 = 1$. (This condition is true when invalid bits are from 1010 to 1111). In this case, multiplexer select bit is set to 1, which causes 6 to be added to the result, correcting its decimal representation

The multiplexer select bit is also set to 1 if addition of X & Y generates carry out as when X = 8 & Y = 9. This also causes the circuit to add 6 to original sum.

if either parallel adder generates a carry out of 1, the carry out of BCD adder should also be 1.

* Disadvantages of using BCD notations

1) By representing numbers in decimal it is a waste of considerable amount of storage space

17 → 00010111 – BCD

Since the no. of bits needed to store decimal number in binary code is greater than the no. of bits needed for its equivalent binary representation.

2) The circuit requires to perform decimal arithmetic are complex.

Despite of these limitations there are some merits of decimal representations.

Advantage of BCD

Computer inputs & outputs data are generated by people who use the decimal system Some applications such as business data processing require small amount of arithmetic computations compared to the amount required for input & output for decimal data.

For this reason, some computers & all electronic calculators perform arithmetic operations directly with the decimal data (in binary code) and thus eliminate the need for conversion to binary & back to decimal. Some computer systems have hardware for arithmetic calculating with both binary & decimal data.

* Specialized Arithmetic Hardware

⇒ The different special arithmetic hardware are designed to speed up arithmetic computations

Coprocessors were formerly used to speed up computations

The coprocessor chip had specialized hardware that performed arithmetic calculations much more quickly than microprocessor. The coprocessor monitored instructions on system data bus. When the microprocessor fetched an instruction that the coprocessor could execute, the coprocessor sent a signal to microprocessor indicating that it would perform computation. The coprocessor then calculated the result & sent it to microprocessor. If the coprocessor was not present, the microprocessor executed the instruction in slow way.

* 3 techniques to improve speed of arithmetic calculation:

→ 1) Pipelining
2) Look up tables
3) Wallace tree multipliers

e.g Coca cola manufacturing process.

1) Pipelining : (Arithmetic Pipelining) consists of number of stages. Firstly, data enters a first stage which performs some operations on data. Then results are passed to next stage which performs its operations. and so on until the final computations has been performed.)

For arithmetic pipeline, an individual stage might be an adder or a multiplier. A pipeline does not speed up individual computation It improves performance by overlapping computations. For e.g:- In a 3 stage pipeline, the first stage process data set 1 while second stage process data set 2 & last stage process data set 3. The net effect is that results are output more quickly than in non-pipelined arithmetic.

For i = 1 TO 100 DO {A[i] ← (B[i] . C[i] + D[i])}

Each operation ~~x~~ multiplication & addition takes 10 ns to complete.

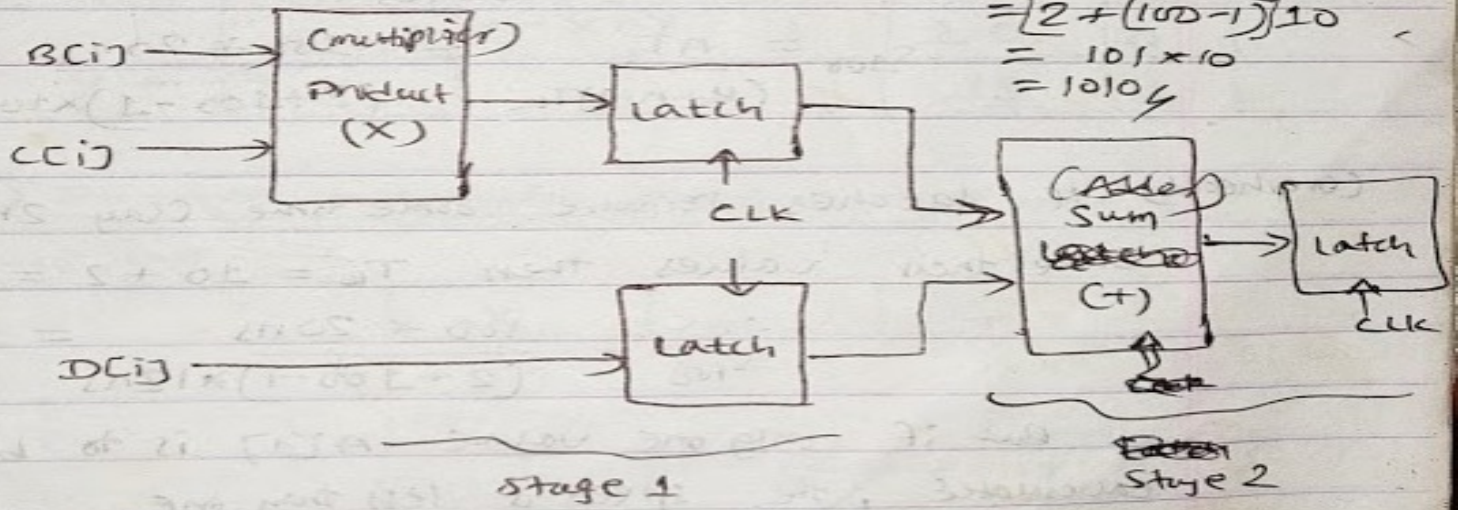
A non pipelined processor now takes 20ns (10+10ns) to calculate A[i] & total (100 * 20ns) = 2000 ns to execute code.

B[1] * C[1] + D[1]
1st stage 10ns
2nd stage 10ns
B[2] * C[2] + D[2]

But A pipelined can perform above operation in two stages - The first stage perform multiplication & second perform addition.

In pipelining, during first 10 ns, the first stage calculates B[1] * C[1]. In next 10ns, second stage adds this value to D[1] & store result in A[1]. At the same time when second stage is performing, stage 1 multiplies B[2] & C[2] then after 10 ns stage 1 performs B[3]·C[3] & stage 2 calculates final value A[2]. So, the pipeline execute above code in 1010 ns instead of 2000 ns

$$(1C+(n-1))t_k$$
$$=[2+(100-1)]10$$
$$=101\times10$$
$$=1010$$



- should be with

The speedup factor of pipeline is given as

$$S_n = \frac{nT_1}{(K+n-1)T_K}$$

where S_n is amt of time needed to process 'n' pieces of data. K = stages. T_1 is amt of time to process one piece of data T_K is clock of K-stage pipeline.

In above example

$T_1 = 20ns$

$T_K = 10ns$

$n = 100$

$K = 2$ stages

$$S_{100} = \frac{nT_1}{(K+n-1)T_K} = \frac{100 * 20ns}{(2+100-1)\times 10ns} = 1.98$$

overhead) if latches require some time (say 2ns) to store their values then $T_K = 10 + 2 = 12ns$

$$\therefore S_{100} = \frac{100 * 20ns}{(2+100-1)*12ns} = 1.65$$

But if only one value A[1] is to be calculated, the speed is less than one

$$S_n = \frac{1 * 20ns}{(2+1-1)*12ns} = 0.83 < 1$$

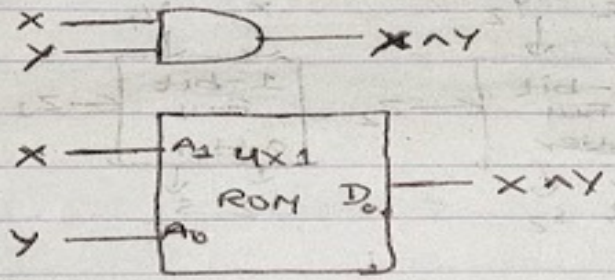
$\Rightarrow$ slower than non-pipelined

2) Lookup Tables :-

A combinational circuit can be implemented by a ROM configured as lookup table. The inputs to the combinational circuit serve as address inputs of ROM. The data outputs of ROM corresponds to output of combinational circuit.

Example :-

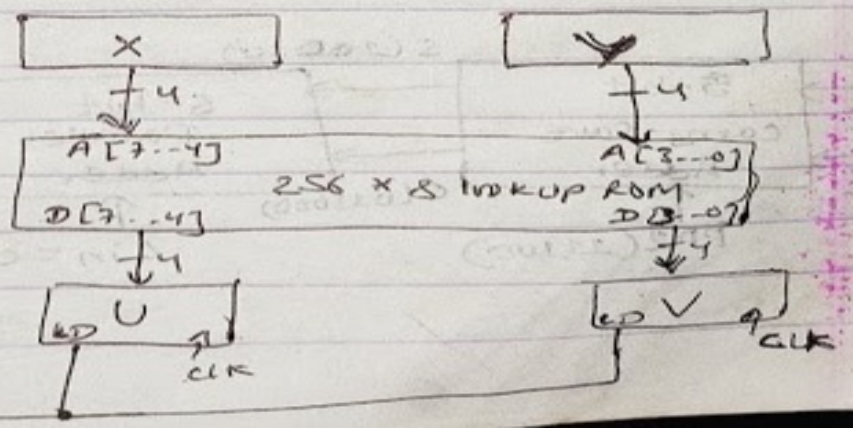
1) Let us consider 4×1 ROM programmed as two input AND gate



Address	Data
0 0	0
0 1	0
1 0	0
1 1	1

$\Rightarrow 2$ ie doubles the speed

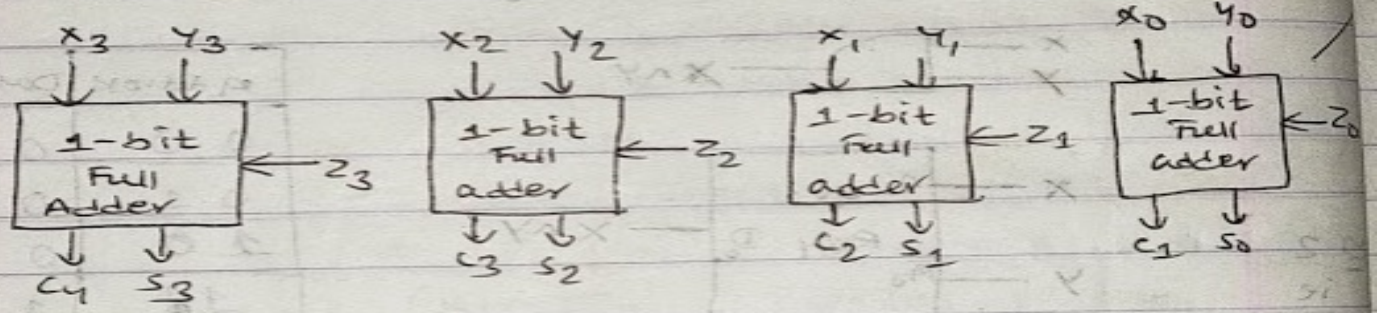
2) Shift-add multiplication algorithm implemented by lookup ROM



$2^n = 2^8 = 256$

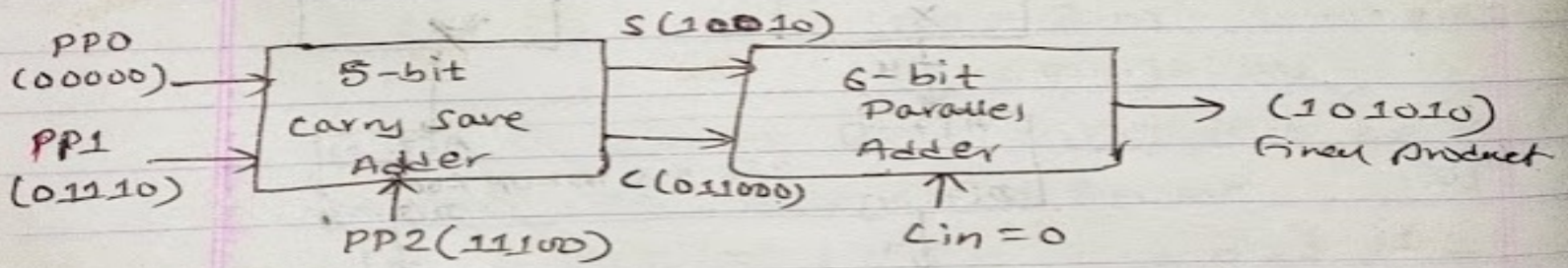
3) WALLACE TREES

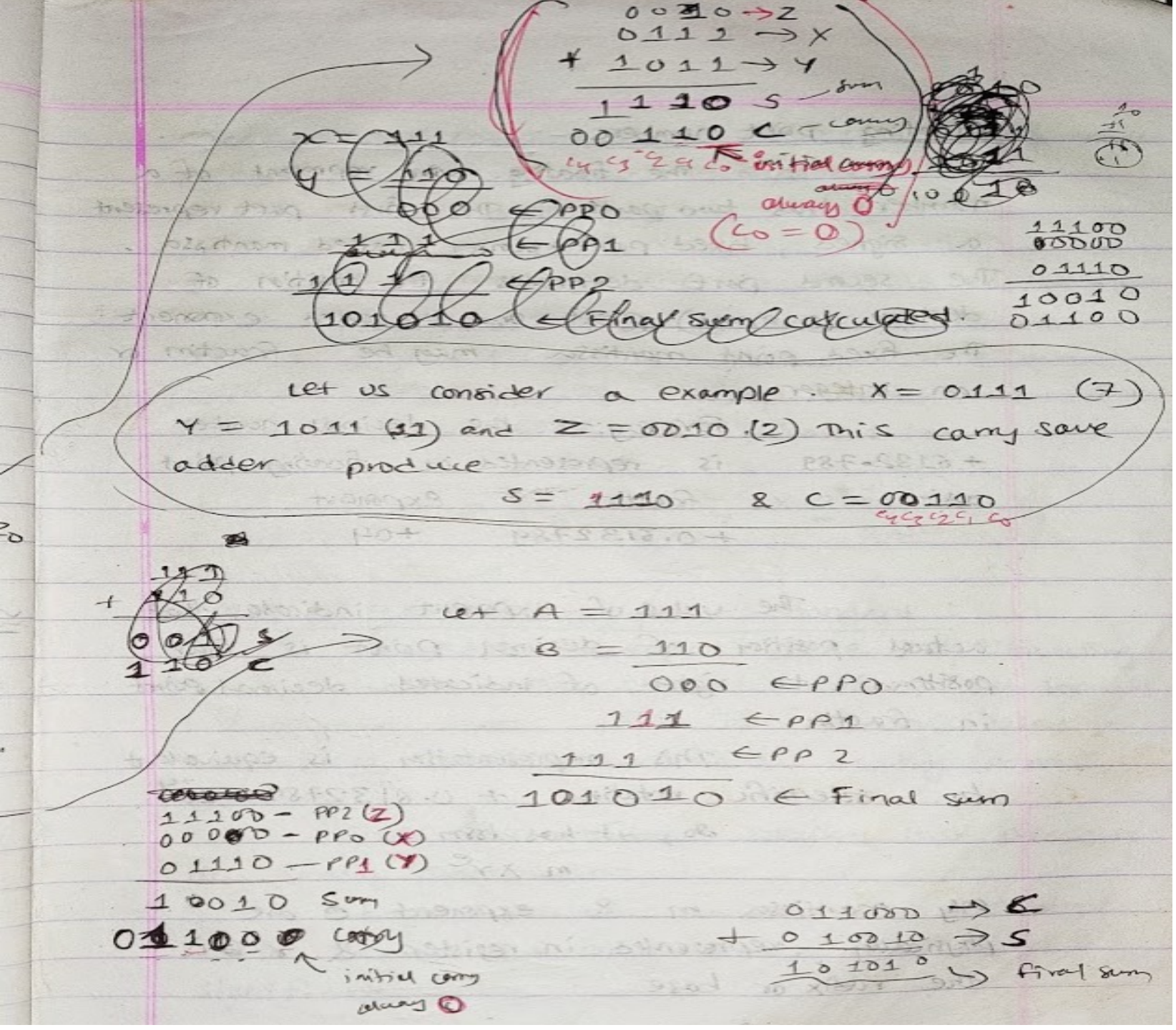
A wallace tree is a combinatorial circuit used to multiply two numbers. It requires more hardware but can produces a product in less time. It requires carry save adders & only one parallel adder. A carry-save adders can add 3 values simultaneously instead of two. It output both sum & a set of carry bits.



(fig : A carry save adder)

A 3X3 multiplier constructed with a carry save adder.




0 0 1 0 → Z
0 1 1 1 → X
+ 1 0 1 1 → Y
1 1 1 0 S
0 0 1 1 0 C
sum
carry
initial carry
always 0
(C0 = 0)
X = 111
Y = 110
000 ← PP0
111 ← PP1
111 ← PP2
101010 ← Final sum calculated.
11100
00000
01110
10010
01100
Let us consider a example. X = 0111 (7)
Y = 1011 (11) and Z = 0010 (2) This carry save
adder produce
S = 1110 & C = 00110
Let A = 111
B = 110
000 ← PP0
111 ← PP1
111 ← PP2
101010 ← Final sum
11100 - PP2 (Z)
00000 - PP0 (X)
01110 - PP1 (Y)
10010 Sum
01100 carry
initial carry always 0
011000 → C
+ 0 10010 → S
10 1010 → Final sum

Floating point numbers

The floating point represent of a number has two parts. The first part represent a signed, fixed point number called mantissa. The second part designates the position of decimal (or binary) point and is called exponent. The fixed point mantissa may be fractim or an integer.

For e.g.— the decimal number +6132.789 is represented in floating point as:—

fractim	Exponent
+0.6132789	+04

The value of exponent indicates that actual position of decimal point is four positions to right of indicated decimal point in fractim.

This representation is equivalent to scientific notation $+0.6132789 \times 10^{+4}$

so, it has form

$$m \times r^e$$

Only mantissa m & exponent e are physically represented in register & r is the radix or base.

For e.g:- a binary number +1001.11 is represented as

	fraction	exponent
	0100111	000102

The fraction has a 0 in leftmost position to denote positive. The binary point of fraction follows sign bit but is not shown. The exponent has equivalent binary number +4. The floating point number is equivalent to

$$m \times r^e$$
$$= (0.1001110)_2 \times 2^{+4}$$

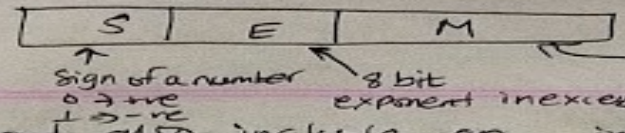
IEEE 754 floating point standard :

⇒ IEEE developed IEEE 754 floating point standard. This standard defines set formats & operation modes. IEEE 754 standard used in virtually all CPUs that have floating point capability

IEEE 754 standard specifies two precision for floating point numbers.

Single precisions numbers have 32 bits : 1 for the sign, 8 for exponent and 23 for the significand.

IEEE standard for floating point numbers for single precision number



Value = $\pm 1.M \times 2^{E-127}$.

The significand also include an implied 1 to the left of its radix point.

Double precision numbers use 64 bits: 1 for the sign, 11 for the exponent and 52 for significand

Example:- (Single precision)

+19.5 Sign = 0

Significand = 0011 1000 0000 0000 0000 000 (23 bit)

Exponent = 1000 0011 (131)

(127 + 4)

bias value

+19.5 = 10011.1

= 1.00111 × 2^4

1 not included.

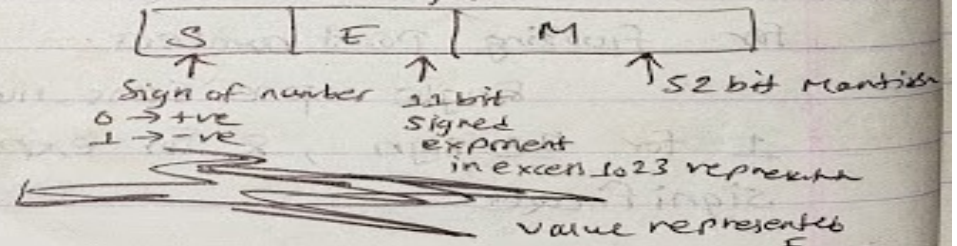
$1 \times 2^1 + 1 \times 2^2 + \ldots$

2 | 19
2 | 9 1
2 | 4 1
2 | 2 0
2 | 1 0
1

0.5 × 2 = 1.0 → 1
0.0 × 2 = 0 → 0

10011.10

IEEE standard for double precision floating point numbers.



value represented

= $\pm 1.M \times 2^{E-1023}$

THE END