Von - Neumann Architecture (1945): Princeton Architecture

Fetch - Decode - Execute cycle:

a) Program counter points to the present instruction to be fetched.
b) Bits in the register "control" the subsequent actions.
c) Fetch the next instruction & continue.

Stored program concept.

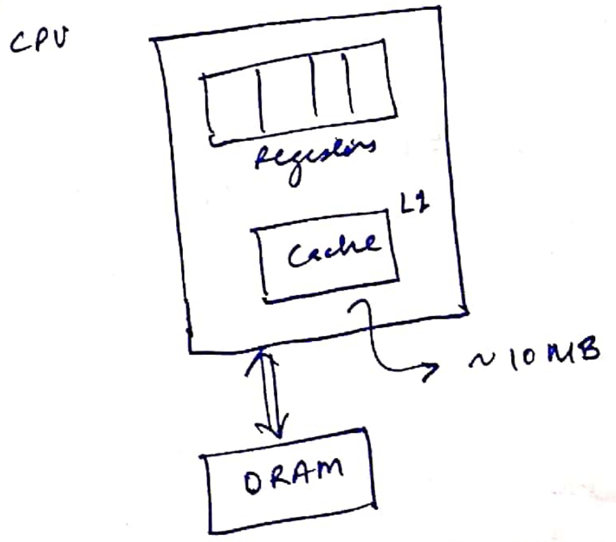
a) Same physical memory to save instructions and data
b) Instruction fetch & data transfer cannot be done simultaneously need 2 clock cycles.

NOTE: Hard disk is treated as an I/O device & NOT memory.
as there are computers without hard disk.

A basic block of code has usually no jump or branch instructions
average size = 6-7 instructions
in other words every 6th or 7th instruction is a branch or jump.

Von - Neumann Bottleneck:

Time spent in memory access limits performance.
(as same memory is used for program & data)



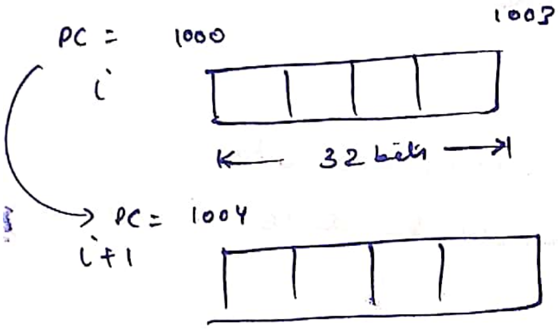
note an 18Mb Cache is not prefered over 2, 10 cache and 8 cache as the more the cache size, the more the time to search

To avoid the bottleneck, later architectures restrict operands to registers

MIPS processor has 32-bit registers.

MIPS does not have cache memory

MIPS every instruction is 4 bytes.
∴ program counter increases by 4.



by default pc (program counter) advance by steps of 4. but if it encounters a jump, the control unit takes it to 5000.

$$1000 \xrightarrow{+4} 1004 \xrightarrow{\text{Control Unit}} 5000$$

Branching does not take an extra clock cycle

instructions on signed integers DIFF from unsigned integer
2 sources one destination

add a, b, c # a gets b + c

All arithmetic operations have this form

Compiled MIPS code:

```
add t0, g, h        # temp t0 = g + h
add t1, i, j        # temp t1 = i + j
sub f, t0, t1       # temp f = t0 - t1
```

C code:

$f = (g+h) - (i+j)$

operations occur on data already present in registers. ∴ we need to do register binding

MIPS has 32 × 32-bit register file
Numbered 0 to 31
32-bit data called a "word"

Assembler names :

1) \$t0 , \$t1 ..., \$t9 for temporary values
2) \$s0 , \$s1 ... \$s7 for saved variables

Processor → memory LOAD
memory → Processor STORE

Direct transfer of data from memory to memory (NOT allowed)
operation on data in memory (NOT allowed)

Register numbers :

\$t0 — \$t7 are reg's 8 – 15
\$t8 , \$t9 are reg's 24 – 25
\$s0 – \$s7 are reg's 16 – 23

Immediate Operands :

Constant data specified in an instruction

$$\underbrace{\text{addi } \underbrace{\$s3}_{5\text{ bit}},\ \underbrace{\$s3}_{5\text{ bit}},\ 4}_{32\text{ bit}}$$

} entire instruction must be encoded in 32 bits.

MIPS I - format instruction

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

MIPS R - format instruction

op	rs	rt	rd	shamt	funct
6	5	5	5	5	6

op : operation code
rs : first source register number
rt : second source register number
rd : destination register number
shamt : shift amount (00000 for now)
funct : function code (extends opcode)

Barrel Shifter

Special shift register !!

```
Loop — A1 (address)
  :
  :
A2 — bne   $t1, $zero, Loop.
```

this contains $A_2 + 4 - A_1$ and **not** the entire address of Loo

b/c **Loop** address = 32 bits

space in beq for **Loop** ~ 16 bits

NOTE: shamt (shift amt.) has 5 bits as we can shift by maximum 0-31 positions

shift right might cause problem in **signs**!

j L1

unconditional jump to instruction labelled L1

C code:

```
while ( save[i] ::k)  i+= 1;
```

(checks whether all elements of array "save" are equal to k)

i = 0
$s3 = 0

i in $S8
k in $s5
address of save in $s6

compiled MIPS code:

```
Loop:   sll    $t1, $s3, 2         $t1 = 0x4 = 0
        add    $t1, $t1, $s6       $t1 = $t1 + $s6
                                       = & save[0]
        lw     $t0, 0($t1)         $t0 = Mem[$t1 + 0]
        bne    $t0, $s5, Exit
        addi   $s3, $s3, 1         $t1 = $s3 x 4 = 4
        j      Loop                $t1 = & save[i]
                                        :
Exit.                                and so on
```

To check whether 2 32-bit no.s are equal,

bitwise XOR the 2 numbers :

Take NOR of all bits in the 32 bit XOR value.

if NOR = 1 then ~~it~~ XOR value = 0
and both no.s are equal .

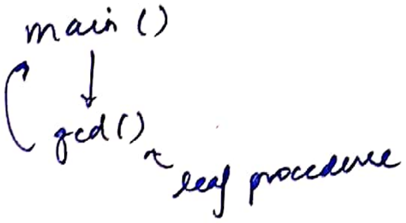
Note : hardware for <, ≥ is slower than =, ≠
that is why no blt, bgt etc.

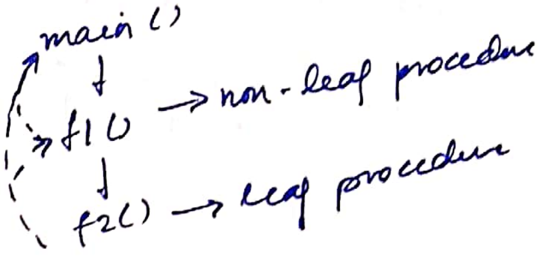
→ Small negative numbers in 2's complement look like large numbers in unsigned.

let array bound = 15

index = 16

$\underbrace{\text{bound} - \text{index}}_{\text{signed subtraction}} = -1$





jal f2
↑
jump and link

jr \$ra stores address of instruction next to the one it has to jump to in \$ra (return address)

instruction next to jal f2

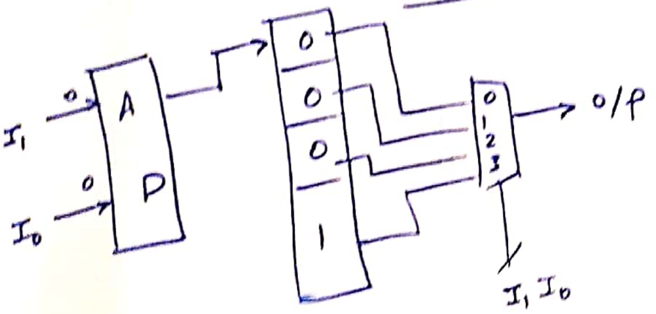
```
fact:
        addi  $sp, $sp, -8
        sw    $ra, 4($sp)
        sw    $a0, 0($sp)
        slti  $t0, $a0, 1
        beq   $t0, $zero, L1
        addi  $v0, $zero, 1
        addi  $sp, $sp, 8
        jr    $ra

L1:     addi  $a0, $a0, -1
        jal   fact
        lw    $a0, 0($sp)
        lw    $ra, 4($sp)
        addi  $sp, $sp, 8
        mul   $v0, $a0, $v0
        jr    $ra
```

```
void strcpy(char *dest, char *src)
{   while (*dest++ = *src++);
}
```

to use a register, first save its previous value somewhere and then after all computations are done, restore its value.

After end of a function, when stack pointer returns to original position, the memory stored in the registers is NOT destroyed.

Lookup table (LUT)



Electrically, all registers are exactly the same, except $zero

ori $t1, $t2, 0x2121

Branch Addressing :

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

NOTE: mips label addresses all end with 00.

∴ all address are divisible by 4

1000

1004 → L1 jumps to P.C here.

2000 → beq $t1, $t2 L1

2004 → P.C position

∴ offset = ~~1004 − 2004~~ 1000 − 2004

= ~~−996/4~~ −224

Jump addressing : ??

```
Loop: sll  $t1, $s3, 2          80000
      add  $t1, $t1, $s6        80004
      lw   $t0, 0($t1)          80008
      bne  $t0, $s5, Exit       80012
      addi $s3, $s3, 1          80016
      j    Loop                 80020
Exit:                           80024
```

2	2000

Addressing mode

a) Immediate addressing

b) Register addressing

c) lw , sw, lh , sh ; Base addressing

d) PC relative addressing branch conditionals

e). Pseudodirect addressing ; jump ,(unconditional) or jal

26 bit initially → left shifted ,
↓
concatenate with 4 bits of P.C.

MIPS ∈ RISC : Reduced Instruction set computers.

X86 ∈ CISC : Complex Instruction set computers.

Major difference : MIPS has lesser addressing modes

Interestingly CISC came before RISC
↓
(but they were bulky & slow)

- pipelining / superscalar , multicore

NOTE: MIPS has only <u>global</u> registers
∴ for recursion, you need to save on a <u>stack</u>.

NOTE:
arguments : $a0 .. $a3
if more arguments, use an array to store & retrieve from memory.
DO NOT use $t
(it will work but is not recommended)

Q. Implement the following C code in MIPS:

```
int fib (int n) {
    if (n <= 1)
        return n;

    else
        return fib(n-1) + fib(n-2) ;

}
```

$a1 = array
$a2 = argument
$a3 = 1

→

```
fib :
  addi $sp , $sp , -12
  sw $ra ,  0($sp)
  sw $a2 ,  4($sp)
  bgt $a2 , $a3 , recurse
  move $v0 , $a2
  jr $ra

recurse :
  lw $t0, 4($sp)
  addi $t0 , $t0 , -1
  move $a2, $t0
  jal fib
  sw $v0 ,  8($sp)
  lw $t0,  4($sp)
  add $t0, $t0, -2
  move $a2, $t0
  jal fib
```

```
lw $t1 , 8($sp)
add $v0, $v0, $t1
lw $ra, 0($sp)
addi $sp, $sp, 12
jr $ra
```

(2) :

```
func:
        addi $t0 , $zero, 1
        addi $v0, $zero, 1

Loop:
        sle   $t1, $t0, $a0        → set less than or equal to
        beq  $t1 , $zero, Exit
        mul  $v0 , $v0, $t0
        addi $t0, $t0 , 1
        j  Loop

Exit:
        jr $ra
```

```
int func (int a0)
{ int  t0 = 1
  int  v0 = 1

  for ( t0 =1 ; t0 ≤ a0 , ++t0)
     {     v0 = v0 * t0 ;
     }
  return v0;
}
```

Instructions per Clock Cycle (IPC)

IPC average number of enstruclions executed per clock cycle.

For processors with a single execution unit, $IPC|_{max} = 1$

In practice, IPC is always < 1

$CPI = \frac{1}{IPC}$; $CPI|_{ideal} = 1$, $CPI|_{actual} > 1$

↓

clock cycles per instruction

$IPC = 0.8$
$\Downarrow$ $CPI = 1.25$

Goal of COA:

increase IPC (and thus decrease CPI)

If we have super-scalar processors with $IPC|_{max} > 1$
we will get better processors !!

Execution time of a program (P) :

a) P has machine level instructions (IC - instruction count)

b) CPI : average clock cycles per enstruclion

c) CCT : clock cycle time.

$\therefore$ CPU-time = $\boxed{IC \times CPI \times CCT}$ (units (equal to CCT's unit)

humans can distinguish b/w only events which are atleast 10 milliseconds apart.

$Ex|_{CPU}$: $IC \times CPI \times CCT$

CCT → completely dependent on Technology

IC → better Algorithm / compiler

CPI → reduce clock cycles

In CMOS IC Tech :

Pmer = Capacitive load × $Voltage^2$ × Frequency

MIPS (Millions of Instructions per second)

$$= \frac{\# \text{ of instructions executed}}{Ex|_{CPU} * 10^6}$$

Rejected as $Ex|_{CPU}$ → 5 times

# of instructions → 5 time

$\Big\}$ MIPS stays same

Each instruction of i^{th} class : CPI_i # of clock cycles to execute

Note: Just total no. of instructions does not help, as some instructions may take up more time than others.

Amdahl's law : ("Law of diminishing Returns") (as speedup is not $\propto$ number of CPUs)

Time taken to execute a program using one CPU = T_1

Time taken to execute the program using 'n' CPUs = T_n

$\therefore$ Speedup = $\frac{T_1}{T_n}$; Time to execute parallelizable part = P

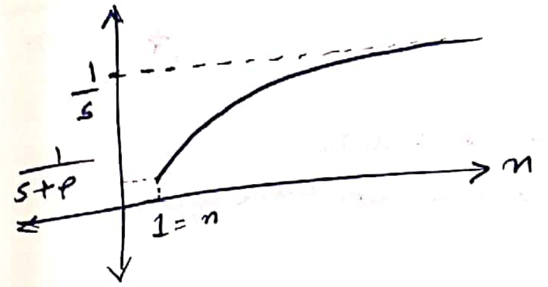
Time to execute non parallelizable part = S

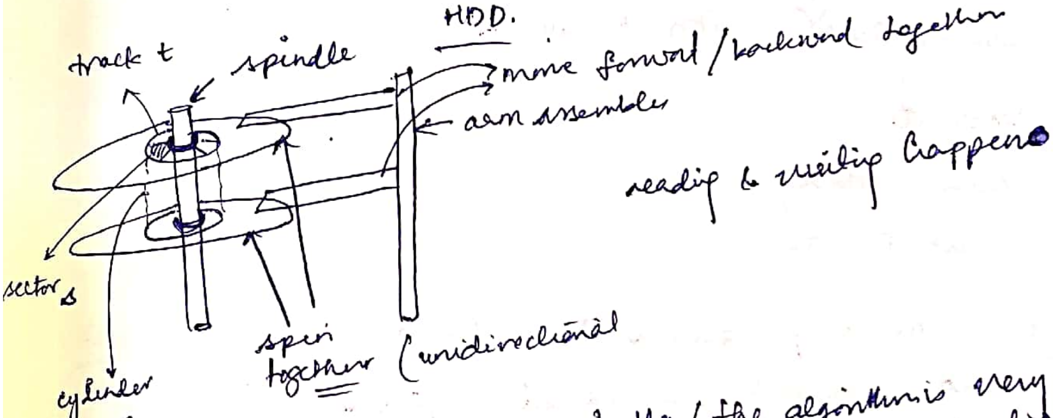
for one processor situation, let (normalized) $S + P = 1$

$$T_1 = P + S \; ; \quad T_n = \frac{P}{n} + S$$

$$\therefore \text{Speedup}_n : \frac{P+S}{\frac{P}{n} + S} = \frac{1}{S + \frac{P}{n}} \quad \text{if } (S + P = 1 \text{ (normalized}$$

$$\boxed{\therefore \lim_{n \to \infty} \text{Speedup}_n = \frac{1}{S}}$$





HDD.
track t
spindle
move forward / backward together
arm assembles
sector s
cylinder c
spin together (unidirectional

reading & writing happens simultaneously

HDD is expensive b/c of the controller (the algorithm is very complex & never divulged)

SATA : (Serial) Advanced Technology Attachment

SCSI : Small computer system Interface.

skuzy → IScazee (Pronunciation !( )

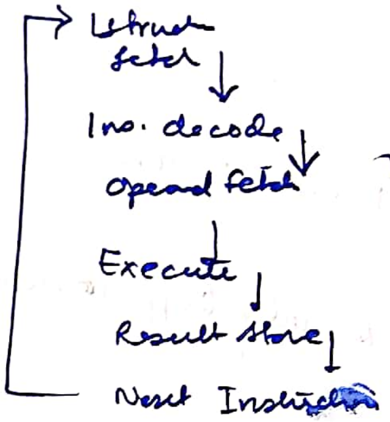
On an average; half the length of a track to be rotated to access data.

On an average; arm assembly moves half the radius of disk

Von Neumann

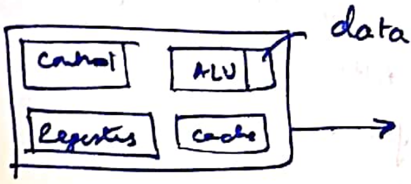
Physical Memory to save instructions & data

Instruction fetch & data cycle, 2 sep. clock cycle



Von neuman

Bottleneck, memory access takes times more, operands stored in registers



CPU

SRAM

DRAM

sec. Memory — Floppy, CD, HDD

Tertiary memory — RAID, Tap

RAID ↓ Redundant Array of Independent disks

Harvard Architecture

separate bus for data & instruction transfer

if not physically separate, then also

Performance affected by :

Algo:— no. of operation (IC & possibly CPI)

~~No. of~~ Language / Compiler / Architecture:

No. of instructions executed per operation.

Processor & memory system

How fast instruction executed

I/O system (including OS)

How fast I/O opns executed

Response time, time for one operation incl. I/O, O.S, CPU time

$\downarrow$

Throughput: work done / unit time

performance $\propto \frac{1}{\text{CPU time}}$

M/C Level program P.

$$\text{Exec. time (of CPU time)} = \underset{\text{Instruction count}}{IC} \times \underset{\text{avg. Clock cycles per instr}}{CPI} \times \underset{\text{clock cycle time}}{CCT}$$

in Hz

CPU time = CPU clock cycles × CCT

Performance:

Reduce no. of clock cycles
Increase clock rate
Hardware designer must often trade off clock rate against cycle count

$$\text{CPU time} = \frac{\text{CPU Clock Cycle}}{\text{Clock Rate}} \quad \text{— how fast clock c..}$$

10 s : 2 ~~GHz~~ $\frac{}{2\ \text{GHz}}$

6 s : $\frac{1.2 \times 2 \times C.T.'}{}$

$\frac{6}{2.4}$ $\quad \frac{\cancel{60}\ 5}{\cancel{24}\ 2} = 2.5$

/ instruction count

No Clock cycles = IC × Cycles per Instruction

$$\therefore \text{CPU time} = \frac{\text{I.C.} \times \text{Cycles per Instruction}}{\text{clock rate}}$$

Avg. cycles P.I.

$$\text{clock cycle} = \sum_{i=1}^{n} (CPI_i \times I.C._i) \qquad i = i^{th} \text{ class of Instruction}$$

$$CPI = \sum_{i=1}^{n} \left(CPI_i \times \frac{IC_i}{IC_{total}} \right)$$

CPU time = IC × CPI × CCT

Algo : IC & CPI

Programming Language : IC & CPI

Compile : IC & CPI

Instruction set architecture : IC, CPI, CCT

CPI : memory hierarchy, pipeline

CCT : logic design, technology

MIPS : Millions of Instructions per second

$$= \frac{\text{IC of program P}}{\text{Exec. time of P in sec} \times 10^6}$$

~~Exec time~~

Power = Capacitive Load × Voltage^2 × Freq

Exec. time best performance measure

use parallelism to improve performance as power is limiting

Power wall : min. voltage limit

heat removal means limited

$$\text{ExTime}_{new} = \text{ExTime}_{old} \left[\left(1 - \text{fraction}_{enhanced}\right) + \frac{\text{fraction}_{enhanced}}{\text{speedup}_{enhanced}} \right]$$

$$\text{speedup (overall)} = \frac{\text{ExTime}_{old}}{\text{ExTime}_{new}} = \frac{\text{Perf.}_{new}}{\text{Perf.}_{old}}$$

MIPS : big endian

MSB at least address of word

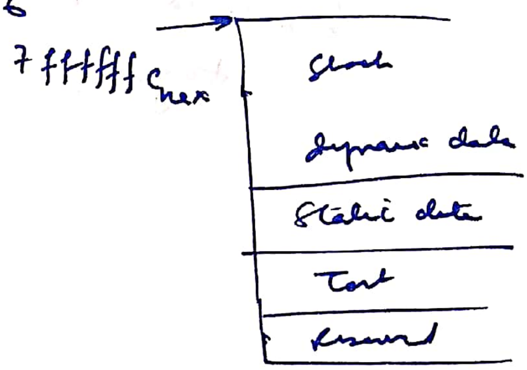
R-format

op	rs	rt	rd	shamt	func
6	5	5	5	5	6 (extends opcode)

I-format

op	rs	rt	const / address
6	5	5	16

→ -2^{15} / $2^{15}-1$



sll : shift left logical
srl : shift right logical

eg. and $t0, $t1, $t2 → mask bits

t2 | 0 0 11 | 0 1 1 1 0 0
t1 | 11 1 1 | 0 0 0 0 0 0
t0 0 0 1 1

or → include bits

nor $t0, $t1, $zero = not

basic block:

no branch (except at end)
no branch target (except at beginning)

slt rd, rs, rt
rd = 1 if rs < rt
else rd = 0

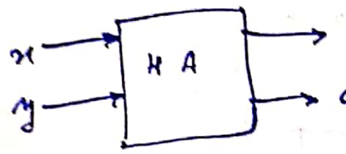
lui
ori

slti rd, rs, constant

slt $t0, $s1, $s2 if s1<s2
bne $t0, zero, L branch to L

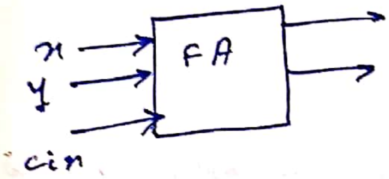
(better than blt)

Binary (Integer) Addition :



sum = $x \oplus y$

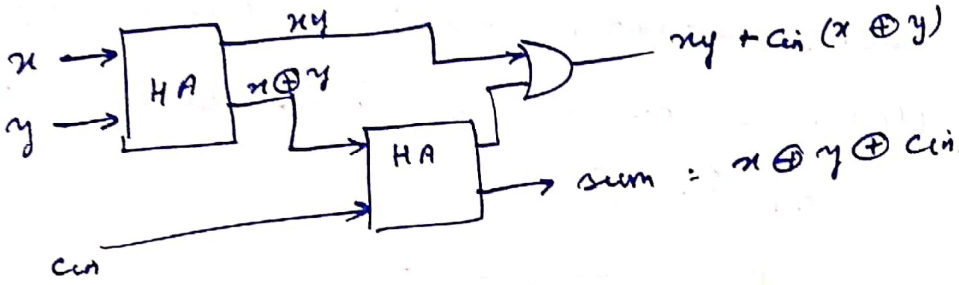
carry = xy



sum = $x \oplus y \oplus cin$

carry = $xy + cin(x+y)$

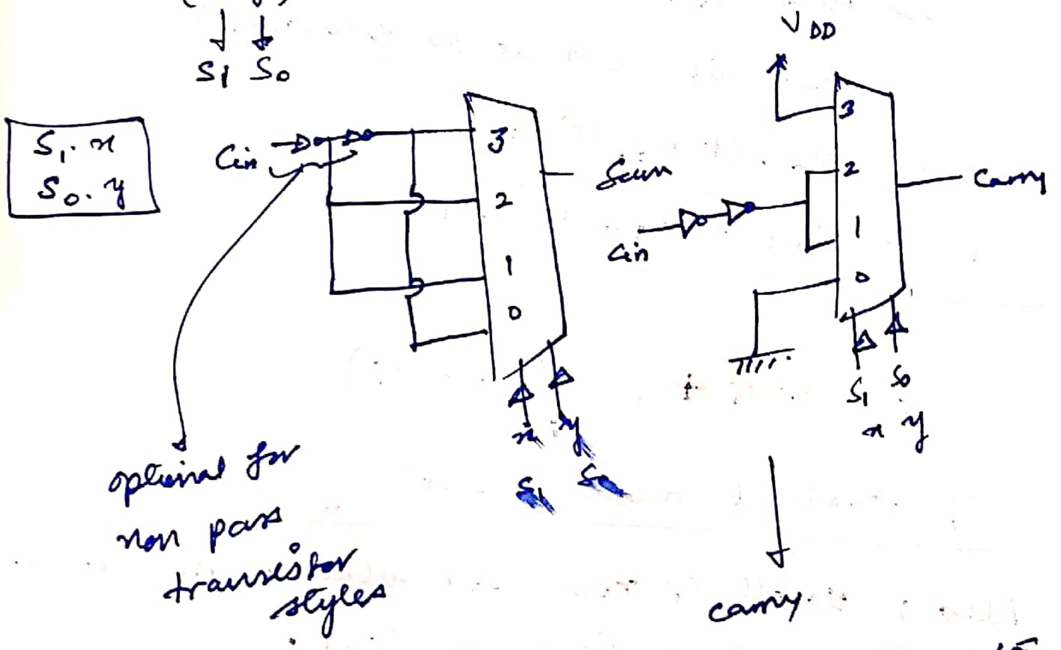
$= xy + cin(x \oplus y)$



FA with MUXes :

○ Muxes can be implemented very efficiently using pass transistor logic CMOS circuits.

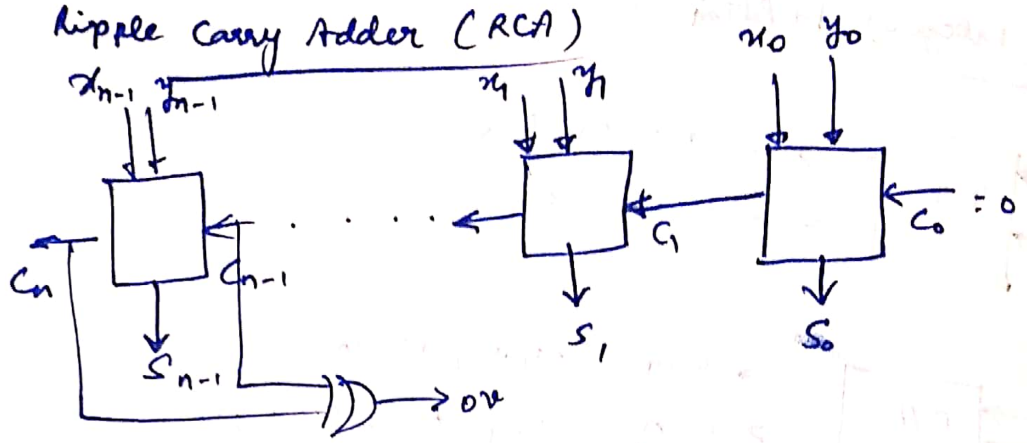
Sum = $(xy)\,Cin + (x\bar{y})\,\overline{Cin} + (\bar{x}y)\,\overline{Cin} + (\bar{x}\bar{y})\,Cin$



carry

$= (xy)1 + (x\bar{y})\,cin + (\bar{x}y)\,cin + (\bar{x}\bar{y})0$

Ripple Carry Adder (RCA)



Overflow Bit :

for 2s complement number addition, usually Cout is discarded. However, Cout is useful in detecting overflo

overflow : 2 no.s of same sign are added but the result of opposite sign.

$ov = x_{n-1} y_{n-1} \overline{s_{n-1}} + \overline{x_{n-1}}\, \overline{y_{n-1}}\, s_{n-1}$

↓

to detect overflow

Prove this

Text ~~book~~ definition $\boxed{ov = C_n \oplus C_{n-1}}$

Sometime C_{n-1} is not available outside the chip then we can do the following :

$S_{n-1} = x_{n-1} \oplus y_{n-1} \oplus C_{n-1}$

$$\boxed{C_{n-1} = S_{n-1} \oplus x_{n-1} \oplus y_{n-1}}$$

$C_i = x_i y_i + C_{i-1} (x_i + y_i)$

Carry Lookahead Adder (CLA) ..

Idea ; unroll C_i recurrence relation to remove dependency on $C_1, C_2, \ldots, C_{i-1}$.

Ideally leads to $O(1)$ delay addition of two. n-bit numbers. But not feasible ~~[illegible]~~ practically.

Carry Generate $g_i \overset{def}{=} x_i y_i$ $g_i = 1 \Rightarrow$ i-th stage generated a carry of its own

carry propagate $P_i = x_i \oplus y_i$

$c_{i+1} : g_i + c_i P_i$

$P_i = 1 \Rightarrow$ ith stage will propagate the i/p carry c_i to c_{i+1}

Absorb. $a_i = \overline{x_i}\,\overline{y_i}$

On an average carry propagation occurs in length 2,

doesn't occur when $x_i, y_i = 0, 0$
$x_i, y_i = 1, 1$

carry select Adders / conditional sum Adders.

Motivation $O(\lg(n))$ delay for n-bit addition

Brent Kung (early 1980's)] very old
Kogge stone (1970s)]

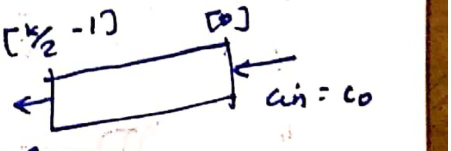
$$S = a \oplus b \oplus c_{in}$$

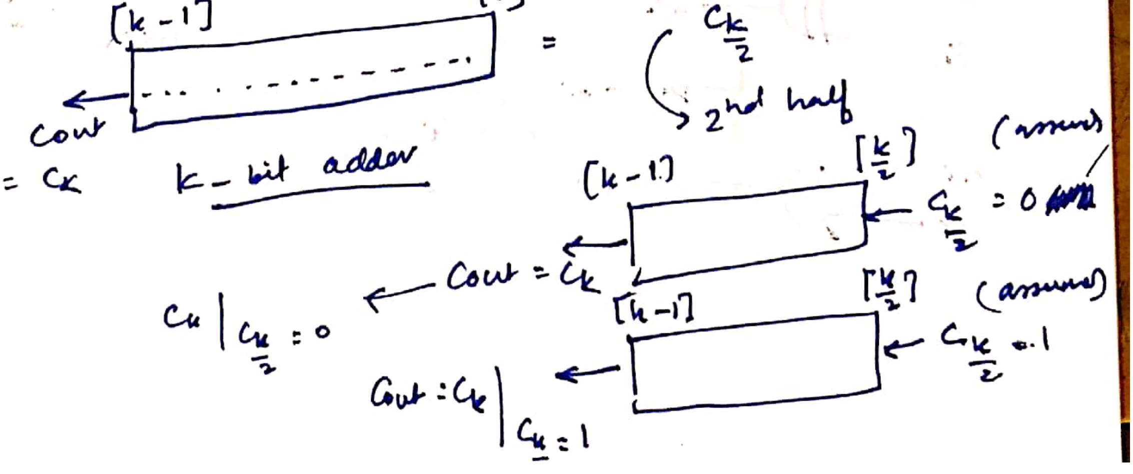
$$C_{out} = \underbrace{ab}_{} + C_{in}(a \oplus b)$$

re-express in terms of $a \oplus b$ to enable logic sharing with 's'.

$$ab = (a'b' + ab) \cdot a$$
$$= (\overline{a \oplus b}) \cdot a$$

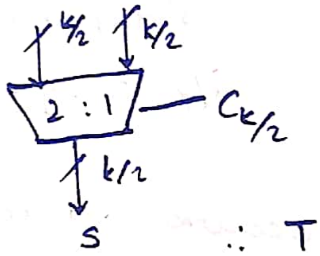
$$C_{OUT} = (\overline{a \oplus b})\, a + c_{in}(a \oplus b)$$



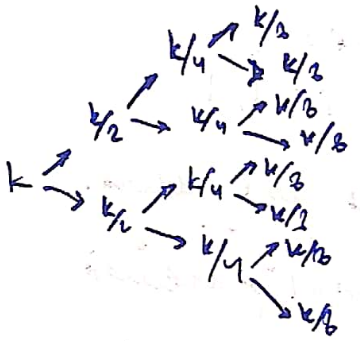


Generate C_k for both $C_{\frac{k}{2}} = 0$ & $C_{\frac{k}{2}} = 1$

finally select the correct o/p using a 2 : 1 mux and $C_{\frac{k}{2}}$ as select line.



$\therefore T(k) = T(k/2) + c$



. . . till n leaf nodes, we'll find specialised full adders

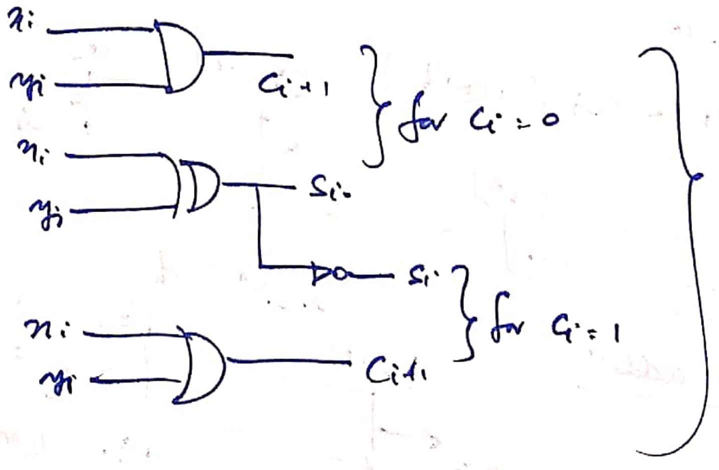
now $S_i = x_i \oplus y_i \oplus c_i$

$\Rightarrow S_i|_{c_i=0} = x_i \oplus y_i$

$\Rightarrow S_i|_{c_i=1} = \overline{x_i \oplus y_i}$

$c_{i+1} = x_i y_i$ if $c_i = 0$

$= x_i + y_i$ if $c_i = 1$



Hardware cost

$O(\underbrace{k \lg k})$ ·(for mux)

it dominates the total time taken

exercise

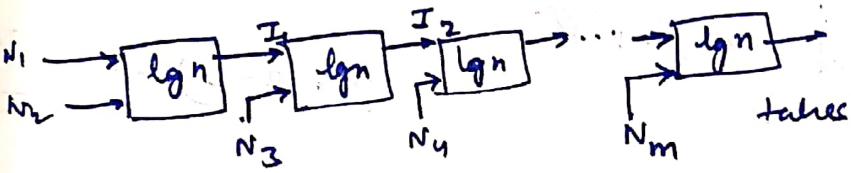
perform 2 level unrollip wip
4 $\frac{k}{4}$-bit adders

NOTE: all this is done as ripple carry takes a lot of time

Carry save adders:

Goal: Assuming I have a fast carry-Lookahead adder with logarithmic delay which I call "Logarithmic CLA" (LCA) ← (gives $O(\lg n)$ delay for n-bit addition)

I will add m numbers each of n-bits in time $O(\lg m \lg n)$

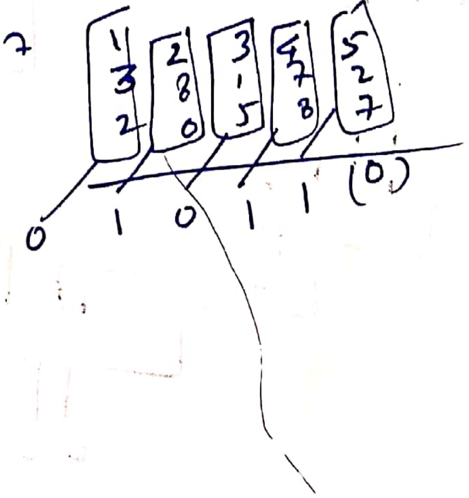


takes $O(m\lg n)$

$S = (x+y+z) \% 10$

$C_{i+1} = (x_i+y_i+z_i)/10$ ↑ integer division

e.g. x = 12345
y = 38172
z = 20587



c =

3-to-2 encoding:

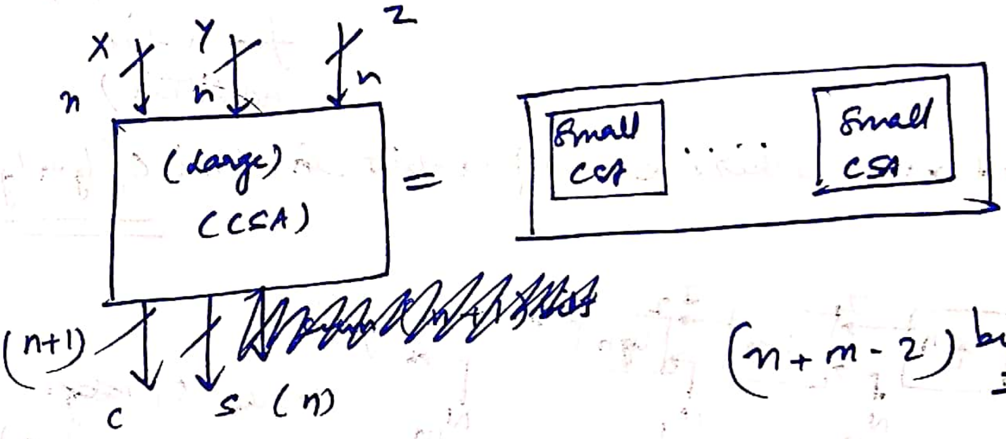
note:

i) {c, s} generation for 3 binary numbers with any no. of bits can be done in O(1) time.

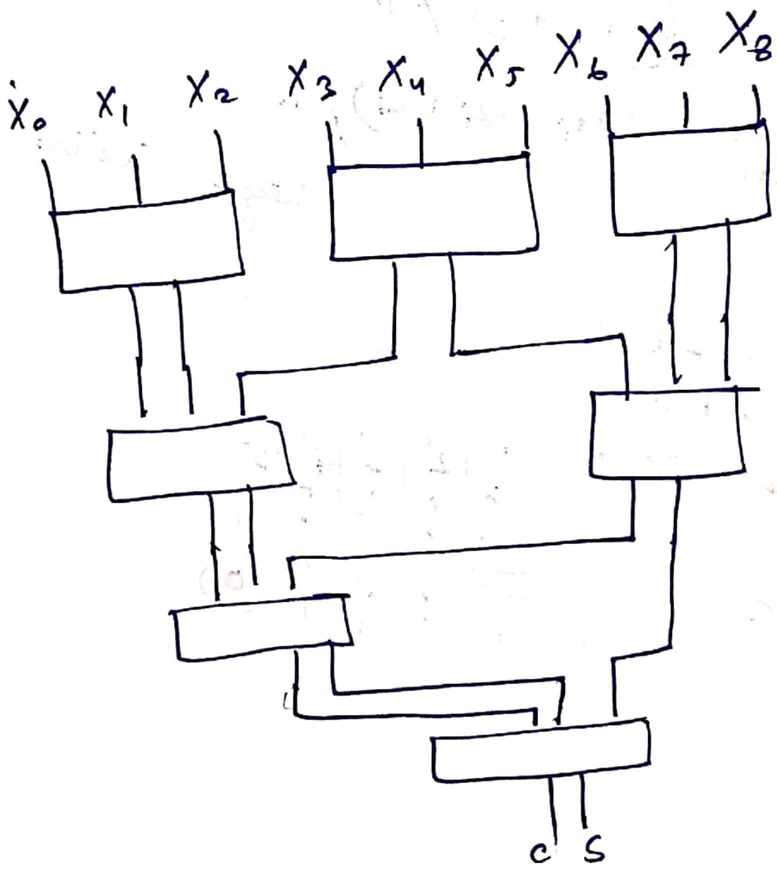
ii) c+s = sum can happen. in O(lg n) time, where n = no. of bits in each number.

iii) 3 numbers each n-bit long can be added in O(lg n) time.

Now to add 'm' binary numbers each "n" bit long.



(n+m-2) bit finally



Quiz 3rd september

Booth's multiplication :

↕ see theory later

example :

4-bit representation X = −5 = 1011

Y = 0111 = 7

2 registers required Q (n+1 bit)

A accumulator (n bit)

Accumulator Q

0000 1011 0 (appended) = {X, 1'b0}

step

0 (init)

1 A − Y 1001 → 11011 0
R.S. R.S. 1100
sign extension

2. R.S 1100 → 01101 X
1110

0101 → 10110 X

3. A + Y 0010
R.S. 10110

4. A − Y 1011 → 1101 X
R.S 1101 1101

product !!

John Hayes

X = 2 = 0010

Y = 6 = 0110

−Y = 1010

1

2

3

4