CS220

HDL (Hardware Description Language)

1. Verilog → C-like
2. VHDL ⇒ VHSIC Description language
   ↓
   Very High speed Integrated Circuit
3. BSV ⇒ Bluespace System Verilog

Simulate: Isim Xilinx ISE 14.7

Synthesize FPGA lab.

Verilog programme ⇒ Collection of modules.

Inside "always" '=' is blocking assignment & '⇐' is non-blocking.
Outside "always" we have to write
assign a = b otherwise it is an error.

Take care while writing combinational logic inside "always"
1. All RHS must appear in sensitivity list
2. All LHS must be assigned/altered in each possible case.
3.

Basic Logic Design:

Decoder: Latency of decoder → Time taken to decode
↳ Depends on slowest of operations ∝ no. of bits.

3 bit decoder $b_0 = \bar{a}_2 \bar{a}_1 \bar{a}_0$

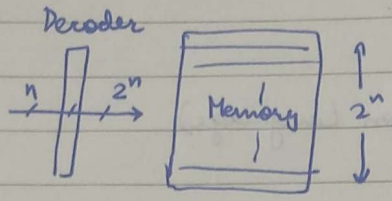
$b_1 = \bar{a}_2 \bar{a}_1 a_0$

⋮

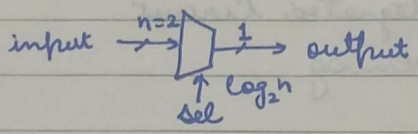
$b_7 =$

So latency of decoder ∝ no. of inputs.

Latency → dictates the fastness of logic circuit

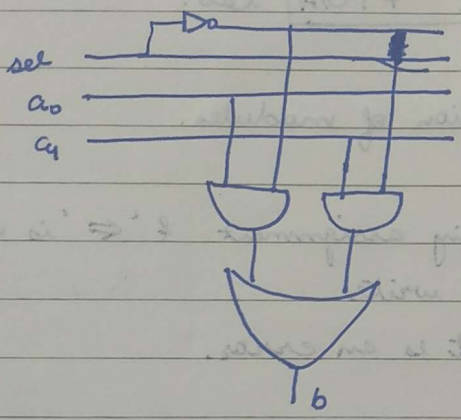


Multiplexor :



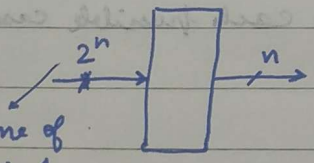
a_0	a_1	sel	b
x	x	0	a_0
x	x	1	a_1

$b = a_0 \cdot \overline{sel} + a_1 \cdot sel$



Increasing no of input in a Mux increases latency.

Encoder:



only one of these is 1.
Others are 0.

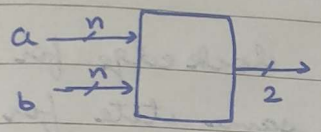
Priority Encoder:

When multipe bits are 1 as input in an encoder, priority is set (inside) for among diff. bits.

E.g. $a_7 = 1, a_6 = 0, a_5 = 0, a_4 = 1, a_3 = 0, a_2 = 1, a_1 = 0, a_0 = 0$

Suppose it is programmed/fixed for 4 as higher priority than 2 and 7, then output is $b_0 = 0, b_1 = 0, b_2 = 1$.

Comparator:



Any Digital logic:

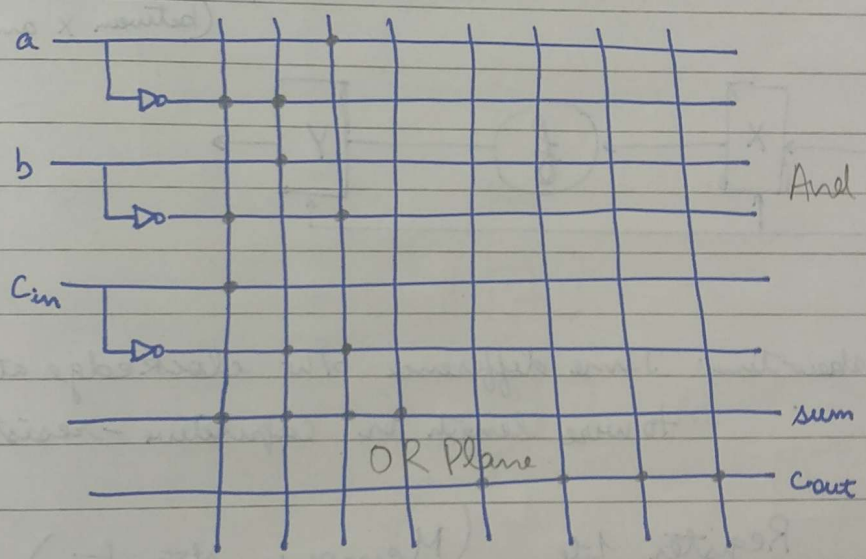
Performance ∝ Latency ∝ Longest Path.

PLA: Programmable logic Array.

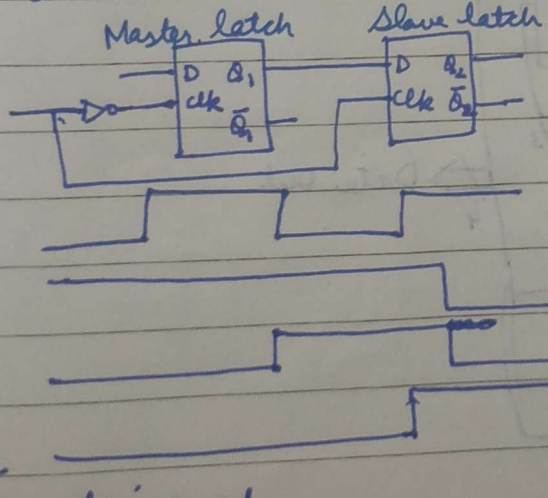
Implementing full adder

$sum = \bar{a}\bar{b}c_{in} + \bar{a}b\bar{c}_{in} + a\bar{b}\bar{c}_{in} + abc_{in}$

$c_{out} = \bar{a}bc_{in} + a\bar{b}c_{in} + ab\bar{c}_{in} + abc_{in}$



Positive level-sensitive latches

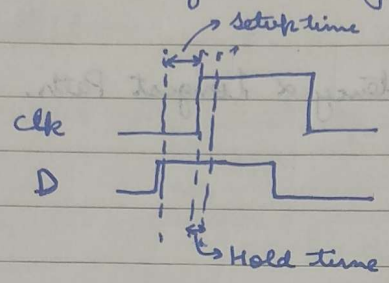


Positive edge trigerred

Synchronous → ~~or~~ occuring at same time (in sync)
Ex. when 2 flip-flops have same clock, they are said to be synchronous.

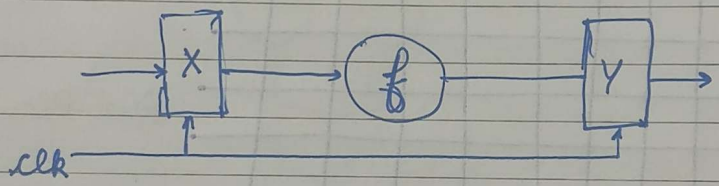
Setup time: D must be in a the required state a short time before clock edge

Hold time: A minimum time after clock edge for which D should remain in ~~same~~ same state for correct functioning



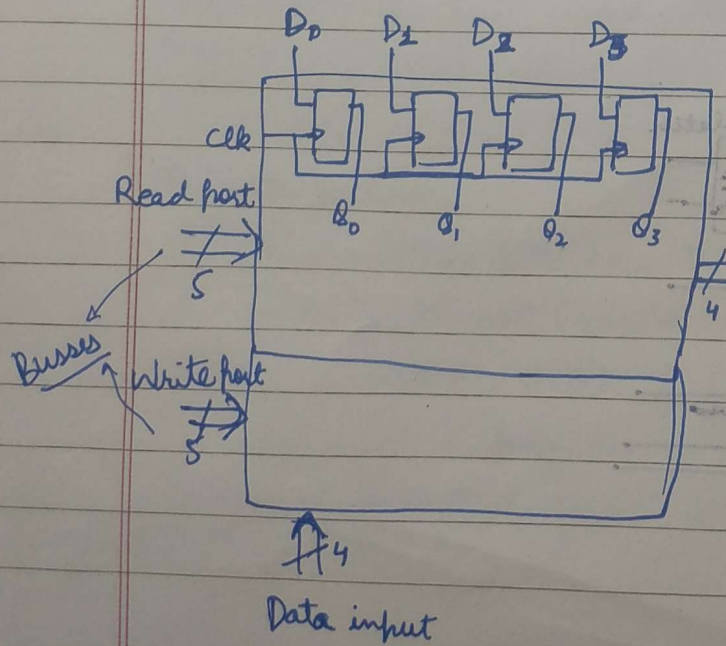
Propagation delay: Clock to Q delay (in Flip-flop)

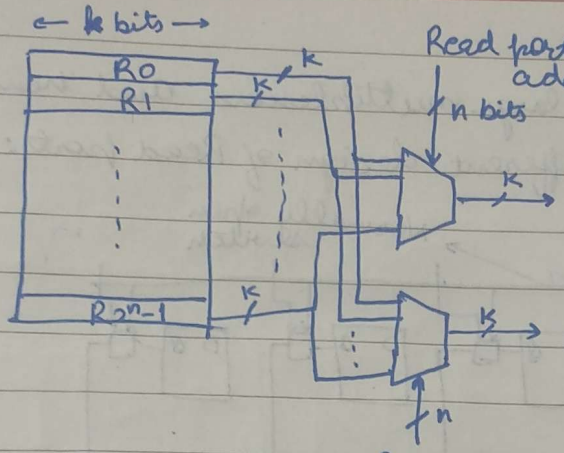
✓ Cycle time ≥ setup time of Y + latency of f + propagation delay of X + clock skew time (between X and Y)



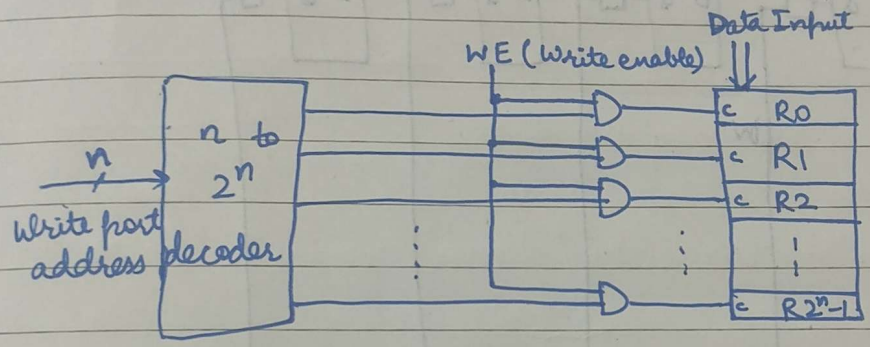
Clock skew time: Time difference b/w clock edge at X and Y due to wire length or capitative-resistive delays etc.

Register file (Memory structure)

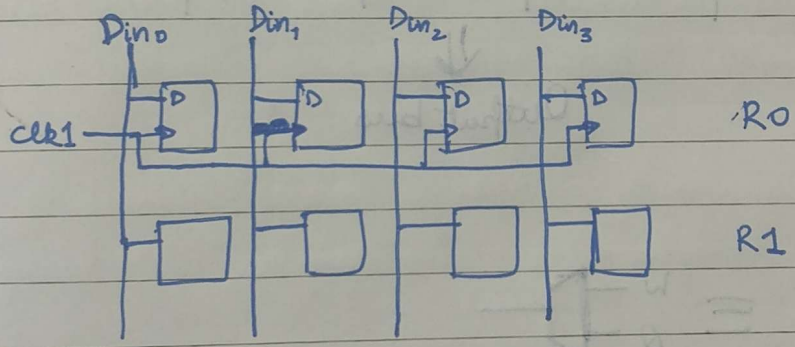




2 Read ports.

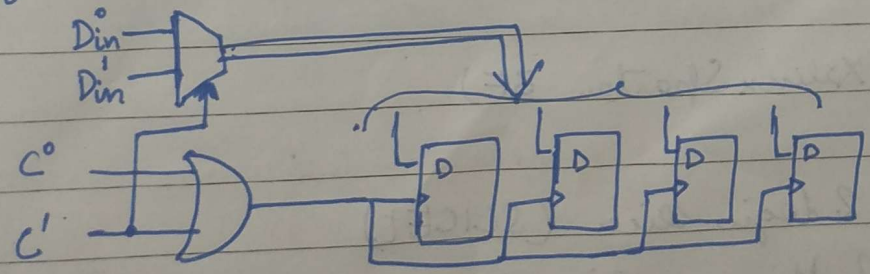


c → clk.

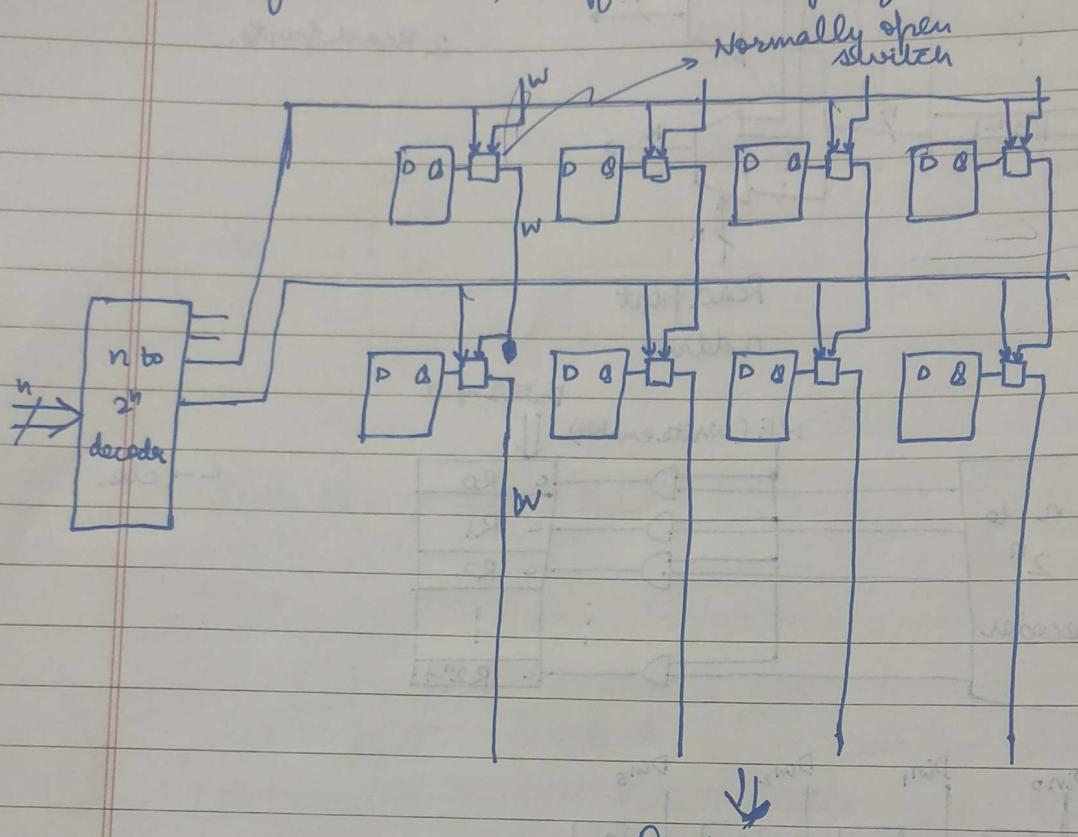


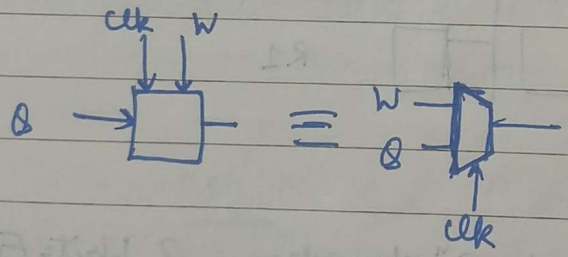
2 Write ports : Two n to 2^n decoders, 2 Write Enable, 2 Data Inputs

c^0 for clock 0 and c^1 for clock 1.



Read ports use very large multiplexors and hence can be very slow. So, Different design of Read ports:





For 2 ~~writing~~ read-write ports, we need 2 decoders, 2 switches at each flip-flop.

FPGA (Xilinx Spartan 3E)

CLB: 2 Logic slices (SLICEL)
2 Memory slices (SLICEM)

Shift Registers:

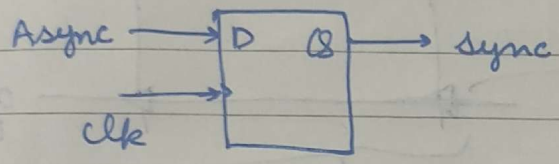
Registers which shift the stored ~~ata~~ data in forward/reverse direction.

Inputs: [Shift amount, Shift direction]

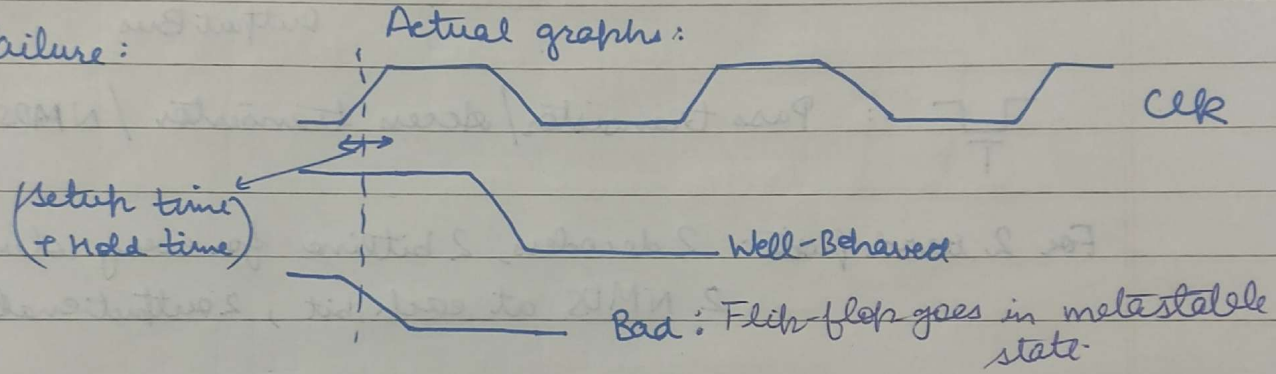
~~and sometimes~~

Asynchronous input:

Use D flip-flop as synchronizer



Failure:



Solution: Chains of 2 flip-flops

Works if metastability period < Clock period.

Memory Structures:

Register file (discussed)

SRAM (Static RAM) → Volatile memory (Holds memory as long as given power)

Fast memory

Gigantic Register file

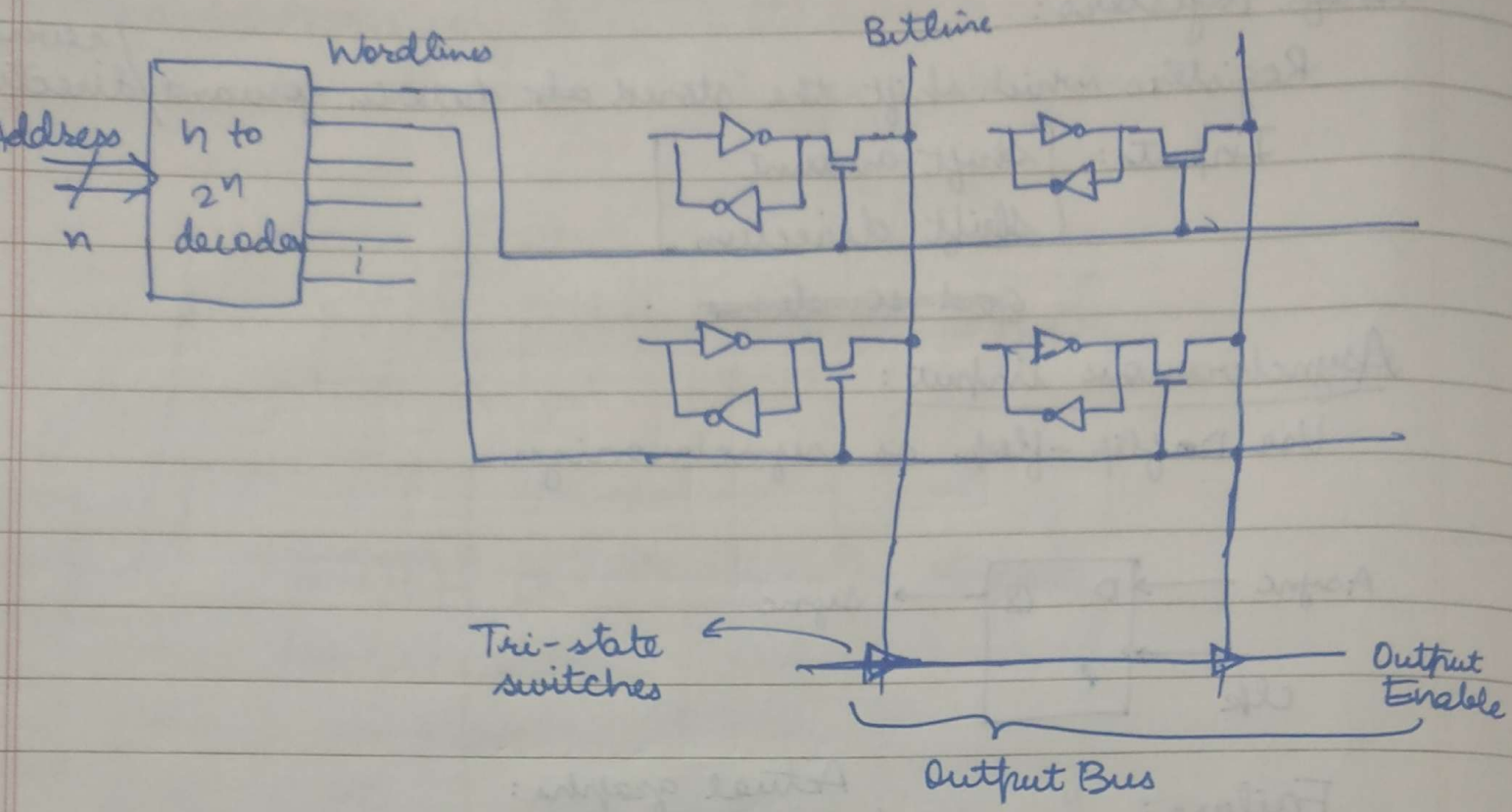
Chip select → to enable specific chip out of a bunch of chips

↳ 1 bit

Rows are called vaults

Each bit → 2 cross-coupled inverters.

All bits in column are connected to a common wire called ~~bites~~ bitline

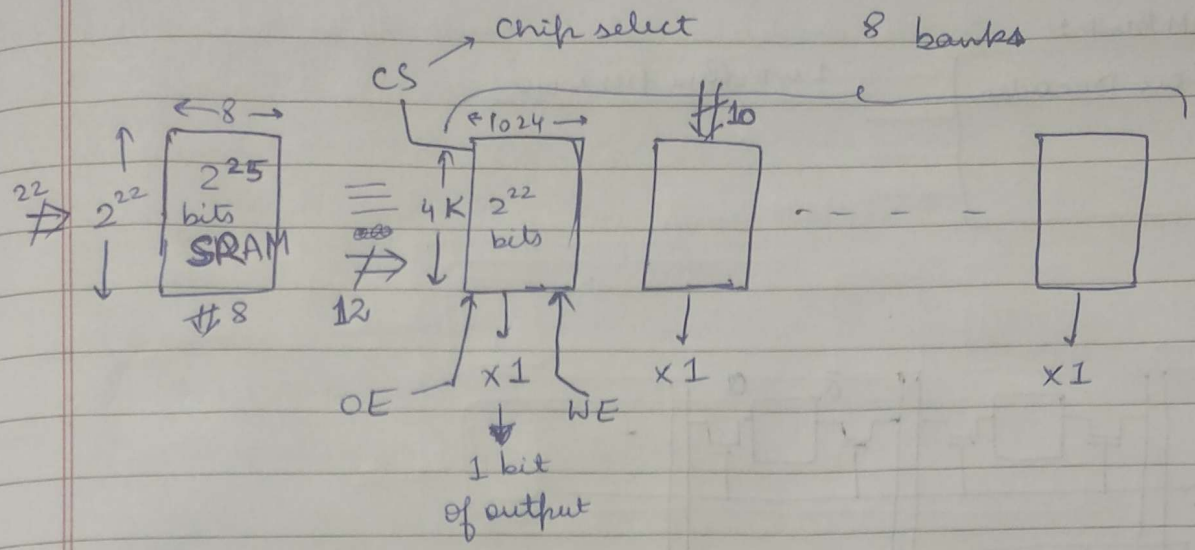


: Pass transistor / Access transistor / NMOS

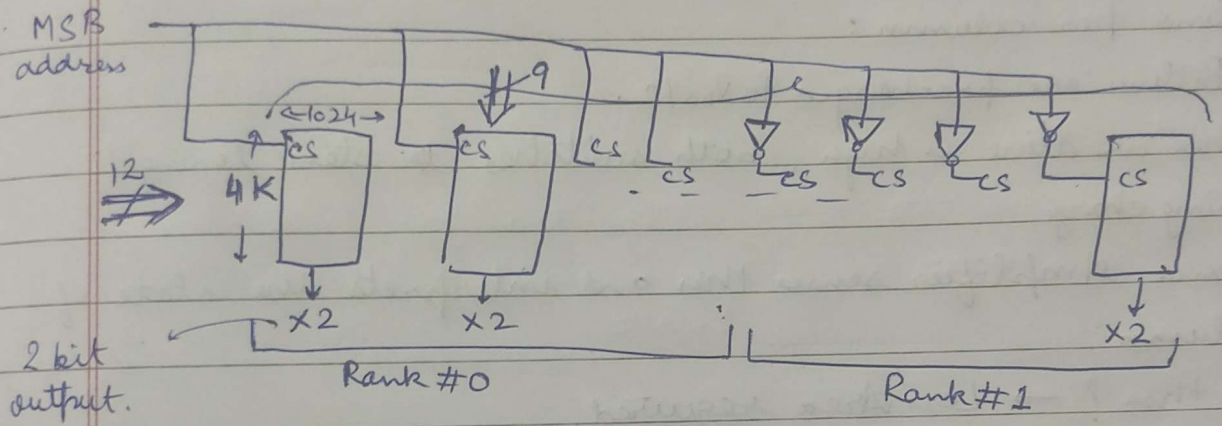
For 2 read ports: 2 decoders, 2 bitlines for every column, 2 NMOS at each bit, 2 output enable.

* Increasing height of SRAM increases length of bitline and size of decoder also, leading to SRAM being slowed down.
* Increasing width of SRAM increases length of Wordline leading to SRAM being slowed down.

Building large SRAMs :



Another way: (Can be Power effective)

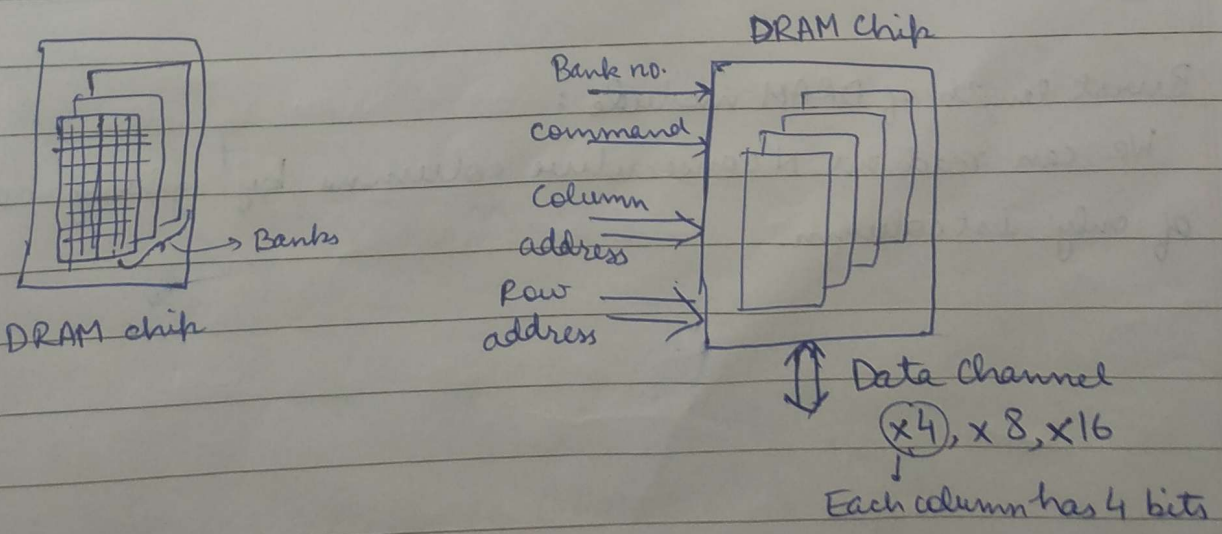


Dynamic RAM:

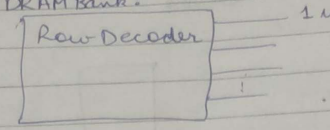
Each cell contains only a capacitor, content leaks away.

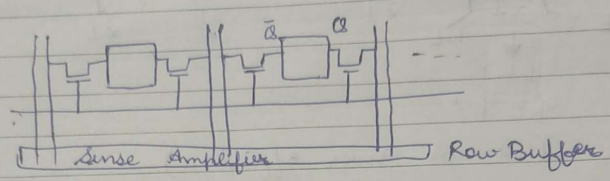
Why?

We require high density storage.



DRAM Bank:





2 Bitlines per column:

Bitline are precharged to $V_{DD}/2$.

When wordline is high, both switches start losing / gaining charge.

Sense Amplifier senses this and interprets the state of column.

Why this? → Less Area required.

RAS signal is sent along with row address (RAS is basically read enable).

Row Address Strobe

Row Address & Column address are never sent simultaneously So, a single decoder can be used.

Burst length of DRAM module:

We can read out N consecutive columns by giving address of only 1st column.

Channel: 64 bit, let output width be k
if x k chips so no. of chips in a rank = $\frac{64}{k}$ E.g. x4 chips ⇒ 16 chips in a rank for 64-bit Channel

Steps:
1. Precharge the bank (Close existing row)
2. Activating a row
3. Read Column (CAS)

Row Hit (Only step 3), Row miss (Step 2 & 3), Row conflict (All steps)

Example:

Lets assume each step above requires x cycles.
Read from the foll addresses:
(r0, c0), (r2, c3), (r3, c1), (r0, c2), (r3, c3)
If DRAM reads in the given order, it takes 15x cycles.
If DRAM intelligently reads, it first reads all columns from a single row and then move onto next row rather than changing row & column each time (∵ changing a row requires more cycles).
So, it takes $3x + x + 3x + 3x + x = 11x$ cycles.

DDR ~~RA~~ DRAM
↳ Double Data Rate

Can transfer the data on both clock edges: Two data transfers in a cycle.
We can't increase clock frequency ∵ power consumption ∝ frequency and it increases.

The first burst has variable latency and after that each half cycle brings a ~~burst~~ (column width of data × no. of chips) i.e (no. of bits in channel) amount of data

Let chips be x8 output chip

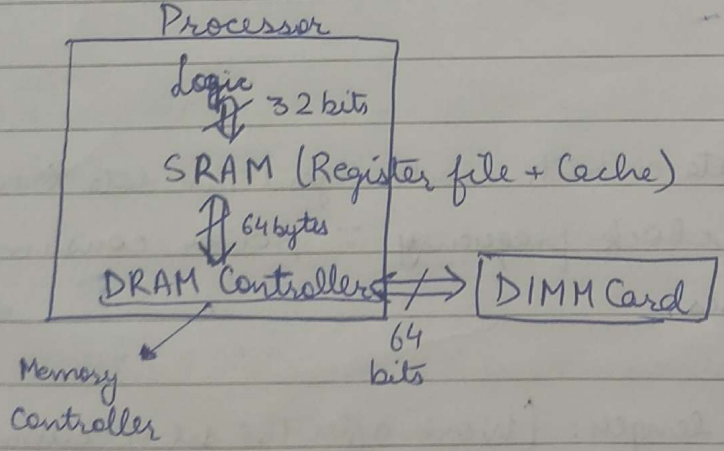
DDR3 - 2133 ⇒ 1.067 GHz

DDR3 - 2400

DDR4 - 2400 ⇒ 2400 million transfers per sec.

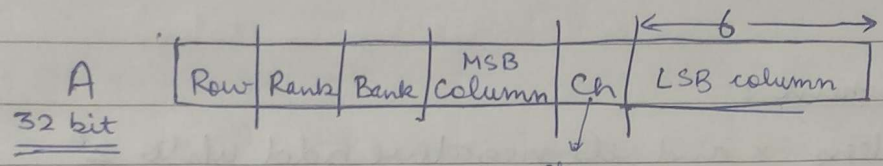
⇓

1.2 GHz



SRAM gives an address A (32 bit to DRAM controller).
& DRAM Controller has to send 64 bytes from address A to address A+63 back to SRAM.

Address Decoding:



Error correction:

Each DRAM address: 9 bits of Data + Ecc (Error correction code)
(8 bits + Parity)

SECDED: Single error correct Double error detect codes.

Finite State Machines

State elements in any sequential logic.

⇒ Finite no. of states

, State Diag

Output f^n depends only on current state: Moore machine

on current state & Input: Mealy machine.

Finite State Machines

Two's Compliment :

Rule: an integer x and its negative add upto 2^n.

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

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

$(2^n - 1) - x$ is bitwise inversion of x.

So, to find $-x$,

1. Bitwise invert x
2. Add one to it.

- To convert 2's complement binary no. to decimal, use -2^{n-1} as coeff of for MSB. and rest is same as regular binary number.

Ex: $7 + (-6)$

$+6 = 0110$

$-6 = 1010$ (In 2's compliment)

$\Rightarrow$

7	0111
-6	1010
	10001

$(-7) + 6$

$-7 = 1001$

$6 = 0110$

$1111 = -2^3 + 2^2 + 2 + 1 = -1$

$(b_{n-1}\, b_{n-2} \cdots b_0)$ In 2's complement $= b_{n-1} \times (-2^{n-1}) + \underbrace{\sum_{i=n-2}^{0} b_i 2^i}_{\alpha}$

$-Y = (1 b_{n-2} \ldots b_0) = -2^{n-1} + \alpha$

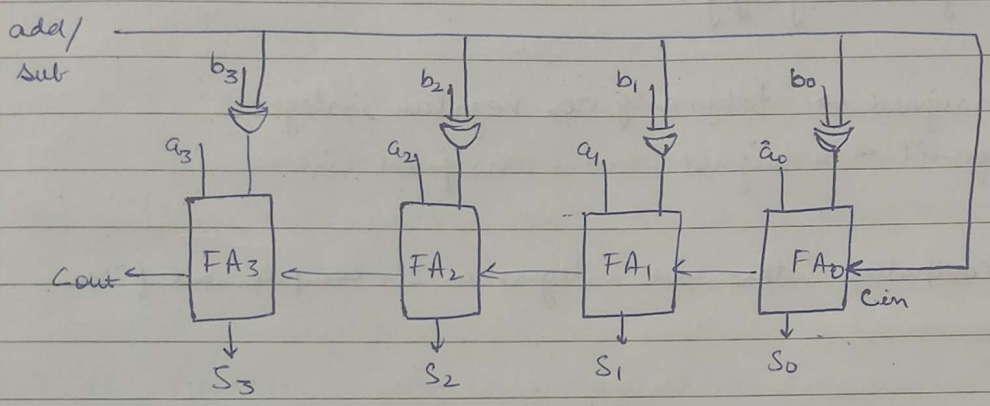
$Y + (-Y) = 2^n$

$Y = 2^n - (-Y)$

$= 2^n - 2^{n-1} - \alpha = 2^{n-1} - \alpha$

$\Rightarrow \quad -Y = -2^{n-1} + \alpha.$

?!



A+B
A-B

Here A & B must be supplied in 2's complement form.

Imbalanced ranges on positive and negative sides.

One's Complement:

+7-6

$$\begin{array}{r} 0111 \\ 1001 \\ \hline 10000 \end{array} \quad \times$$

There are different cases that require different treatment.

Biased Representation

n=4, bias = 8.

-8 to +7

Imbalanced ranges

-8 ⇒ 0000

-7 ⇒ 0001

⋮

+7 ⇒ 1111

6 ways to represent integer in C :

short, unsigned short } n=16 (16 bit representation)

int, unsigned int } n=32

long long, unsigned long long } n=64

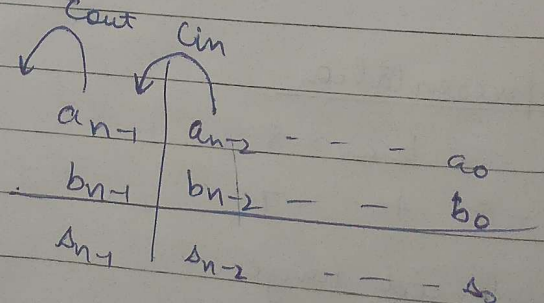
unsigned → takes only non-negative integer.
by default ∞ unsigned means unsigned int.

All computers store -ve integers in 2's complement form.

Overflow

In 2's complement representation, overflow occurs iff carry in to MSB is different from carry out from MSB.

Cin	Cout
0	0
0	1
1	0
1	1



$C_{in} = 0$, $C_{out} = 1$.

$\Rightarrow a_{n-1} = b_{n-1} = 1$.

In this case $S_{n-1} = 0 \Rightarrow$ +ve $\Rightarrow$ wrong $\Rightarrow$ overflow.

$C_{in} = 1$, $C_{out} = 0$.

$\Rightarrow a_{n-1} = 0$, $b_{n-1} = 0$.

$\Rightarrow S_{n-1} = 1$. $\Rightarrow$ -ve $\Rightarrow$ wrong $\Rightarrow$ overflow.

$C_{in} = 0$, $C_{out} = 0$

atmost one of a_{n-1} & b_{n-1} is 1 and atmost one of a_{n-2} & b_{n-2} is 1.

* Here since carry is zero, so no overflow is obvious.

$C_{in} = 1$, $C_{out} = 1$.

atleast one of a_{n-1} & b_{n-1} is 1 and atleast one of a_{n-2}, b_{n-2} is 1.

(* Lecture - slides *)

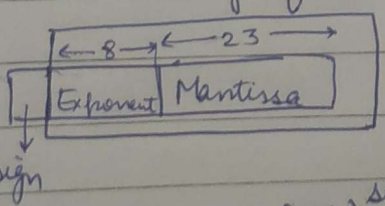
Floating-Point numbers:

Computers → floating-point binary number using normalized scientific notation.

$1.b \times 2^c$ $\quad b, c \rightarrow$ binary numbers.

IEEE 754

1.b → significand



$(-1)^{sign} \cdot 2^{Exponent} \cdot (1.mantissa)$

1.1×10^{-1}

Exponent fields → +ve & -ve exponents.
A larger magnitude floating-point no. should have large 31-bit magnitude
So, can't use 2's complement
∵ 1 at start of -ve exponent makes it larger than +ve exponent.
So, Biased encoding

Bias = 127.

Encoded Exp = 127 + Actual Exp.

So, Biased Encoding preserves monotonicity.

Min +ve number =

Denormalized number:

Largest +ve denormalized number:

0 | 0 - - -0 | 1 1 - - 1

$(1-2^{-23}) \times 2^{-126}$

Smallest +ve denormalized number:

0 | 0 - - -0 | 0 - - 01

$0.0--01 \times 2^{-126} = 2^{-149}$

Representable magnitudes:

0 → (Exponent = -127)

Denormalized: 2^{-149} to $(1-2^{-23}) \times 2^{-126}$

Normalized: 2^{-126} to $(2-2^{-23}) \times 2^{127}$

Rounding :

$x = 1.\underbrace{11\cdots1}_{24}$

$x' = 1.\underbrace{111\cdots10}_{23}$

$x'' = 1.\underbrace{11\cdots1}_{23}$

$x''' = 2.\underbrace{00\cdots0}_{23}$

$x - x' = 0.\underbrace{0\cdots011}_{24}$

$x - x'' = 0.\underbrace{0\cdots001}_{24}$

$x''' - x = 0.\underbrace{00\cdots01}_{24}$

$= \frac{1}{2^{24}}$

Instruction Set Architecture (Assembly Language).

Implementation of ISA should be simple.

Operations : Basic : +, −, *, /

Source operands, Operation ⟶ Destination operand.

MIPS instruction "add a, b, c"

(MIPS: processor family; a: Destination)

For $a = b + c + d + e$

MIPS :

```
add a, b, c
add a, a, d
add a, a, e
```

Compiler : HLL program ⟶ MIPS assembly language program

(HLL: High level lang.)

Note: Eg. diff codes in HLL, lead to diff translations in MIPS assembly lang.

Eg. $d = (b + c) - e$ ≡

```
add d, b, c
sub d, d, e
```

Spill: deallocate and store it in memory.

Operands come from register from register file or memory location (SRAM Cache or DRAM)

HLL program to Assembly language problem:

- Mapping HLL operators to computer instructions
- Mapping HLL operands to computer instruction operands.

In MIPS, many instructions allow only register operands
In x86, both register & memory operands are allowed.

Width of register file dictates width of register operand. (32/64 bit).

Mapping variables to oprands:

- Register allocation of variables is compiler's responsibility
  Goal is to minimize no of fills & spills.

Example: 4 registers : r1, r2, r3, r4.

C program:

```
a = b + c;   // a → r1, b → r2, c → r3
d = e + f;   // d → r4
a = a + d;
b = a + e;
```

In step 2, c is deallocated (without spill)
deallocate b, because that would save a spill
if we deallocate a, spill will be required. (to store updated value of a in memory).

Memory operands:

Direct memory addressing mode (requires 32 bit of address for 4GB memory)

Computer doesn't always know address of all variables.

Displacement mode of memory addressing:

100($2) → (100 + contents of $2) → address of memory operand

12 bits (7 for 100 & 5 for $2).

100 → displacement

$2 → base register

Careful regarding not affecting total access time by addition in this mode.

"load $1, 100($2)" now takes 23 bits instead of 43 bits

```
a = (int *)malloc(size);
x = a[0] + e; ⇒ { load $1, 0($2)
                  add  $1, $1, 1
                  store $1, 0($2)
a[1] ⇒ 1($2).
```