Hardware Modelling Using Verilog

5/9/22

Moore's Law : Every 18 months, no. of transistors in chip will double

VLSI Design Flow → Specification
↘ Synthesis → layout
↗ Simulation

We need to use Computer Aided Design (CAD)

⇒ Based on Hardware Descriptive Language

⇒ HDLs provide formats for rep. outputs of various design steps

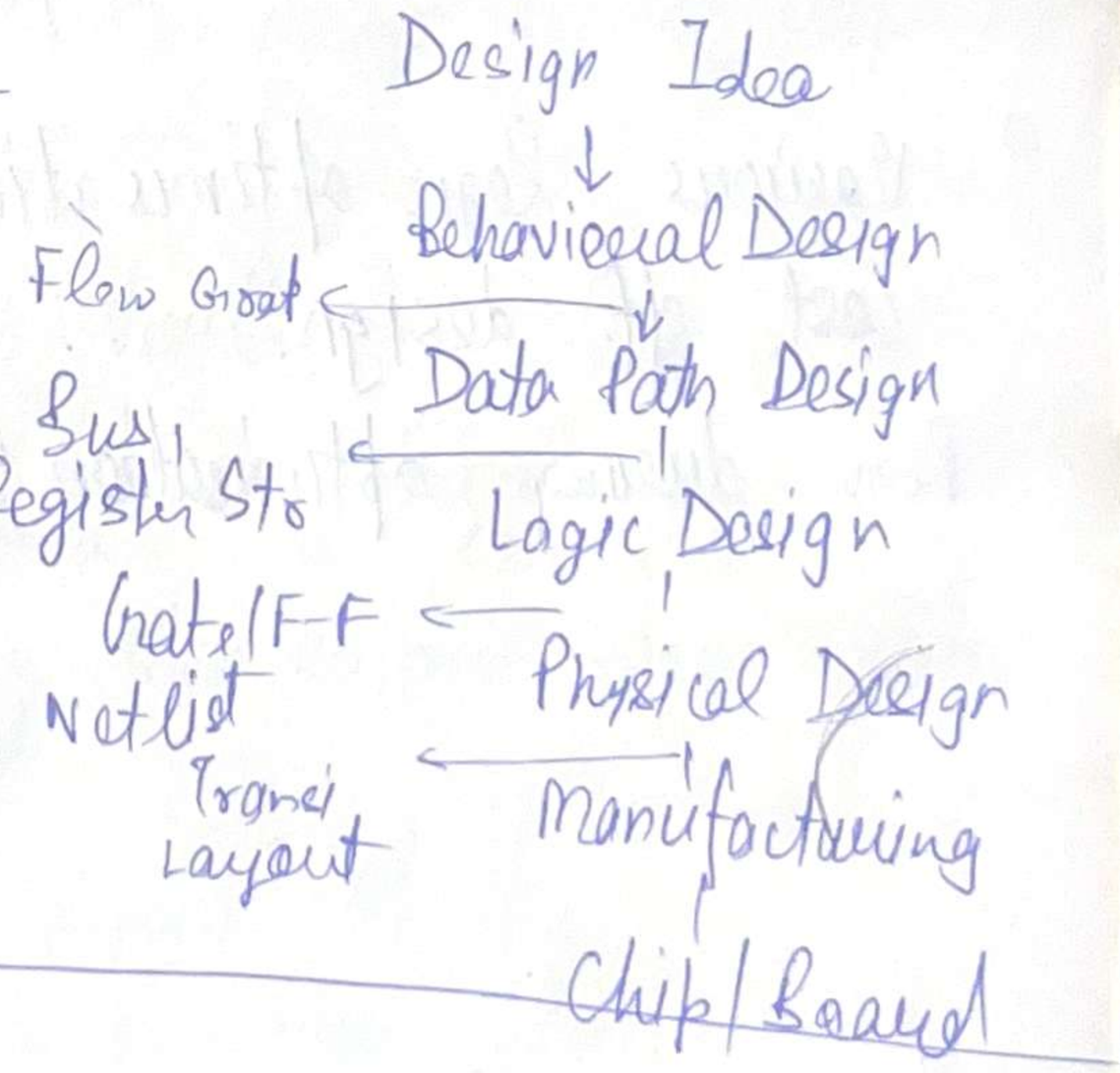
⇛ CAD tool transforms this HDL input into HDL output that contains more detailed info about hardware.

- Behavioural level to Register Transfer level (MUX, Adders)
- RT level to Gate level
- Gate level to Transistor Level
- Tsm to layout level.

Steps in Design Flow

- **Behavioural Design**
  → specify functionality of design. (Boolean expression, Finite state machine behaviour, high-level algorithm)
  → Needs to be synth. into more detailed specifications for hardware realisations.

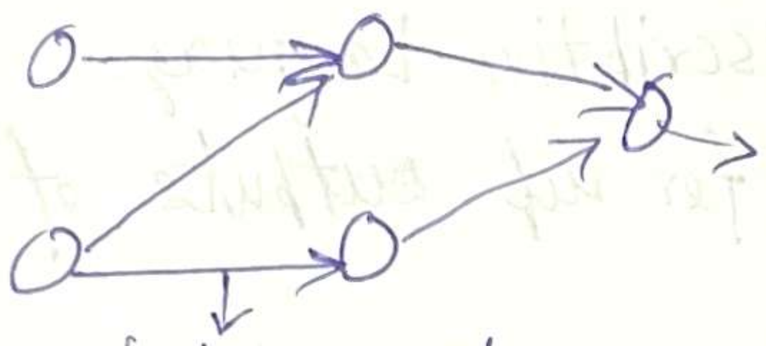
- **Data path design**
  Generate a netlist of register transfer level components, like registers, adders, multipliers, multiplexers, decoders,



~~Data~~ ◦ A netlist is directed graph, where vertices indicate components and edges indicate interconnections

◦ Netlist specification ≡ Structural Design
  ◦ may be specified at various levels, comp. may be functional modules, gates or transistors.
  ◦ Syn. transformed from one level to next.

Netlist



can be Adders, Gate, Transistors

◦ Logic Design → Generate a netlist of gates / flip-flop or standard cells.
  ◦ Standard cell → pre designed circuit module (gates, flip-flops, multiplexers etc) at layout level
  ◦ Various logic optimisation tech. are used to obtain cost eff. design.

Logic minimisation optimisation :→ (i) minimum no. of gates
(ii) mini. no. of gate delays / levels (delay)
(iii) minimise signal transmission activities (ie dynamic power)

◦ Physical Design ⇒ Generate final layout that can be sent for fabrication
(Adders) Final target may be Field Programmable Gate Array (FGPA) where technology mapping from gate level netlist is used

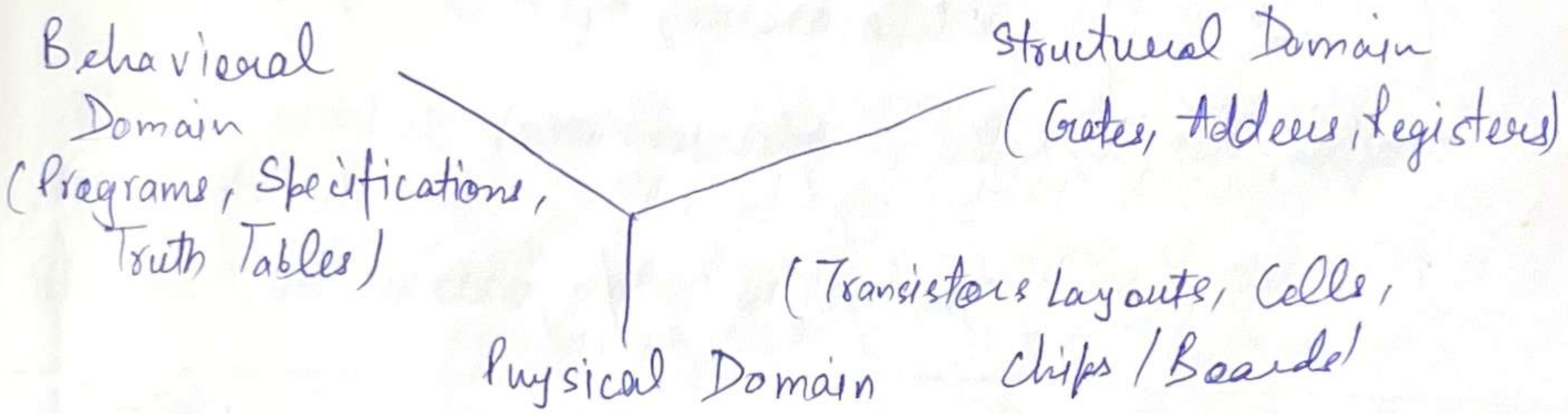
Arduino

Know your Arduino

Lecture - 2 Design Representation

3 pts of view → i) Behavioral
ii) Structural
iii) Physical

conveniently expressed by Y diagrams



Behavioral → Specifies how a particular design should respond to a given set of inputs
Specified by → i) Boolean Equations
ii) Algos written in Verilog or VHDL

Structural spec → Specifies how components are interconnected
List of modules and their ~~specifications~~ interconnections
- called netlist

Physical rep → lowest level of physical specific.
(photo-mask info req. by various processing step
- At module level, physical layout for 4-bit adder may be defined by rectangle or polygon, and collection of parts
- lays number of rectangle or polygon

H M V Continued 29 Sept

Lecture 3 ASIC → Application Specific Integrated Circuit

Used when high performance and high packing density is required

② When it has to be mass produced

FPGA → Field programmable Gate Array

- Fast turnaround time is req. to validate design
- Trade off in performance

~~Code Example~~ ~~module~~

Lecture 4 FPGA [Read on web]

- programmable
- Array of logic cells connected via routing channels
  - Diff types of cells
    - input IO cells.
    - Logic cells
  - Interconnect. b/w cells ⇒ SRAM, anti fuse

Fundamental Building block inside of FPGA are flip flops and lookup table. (LUT)

CLB → ~~Combinatorial~~ Configurational Logic Blocks

Lecture 5 Gate Array

done with metal mask design and processing

Eg. 2 step manufac. process

Standard Cell Based Design

- also called semi custom design style
- Requires developing full custom mask set
- Commonly used logic cells are developed and stored in standard cell library.
- Typical library may contain a few hundered cells

Characteristics

① made of fixed height

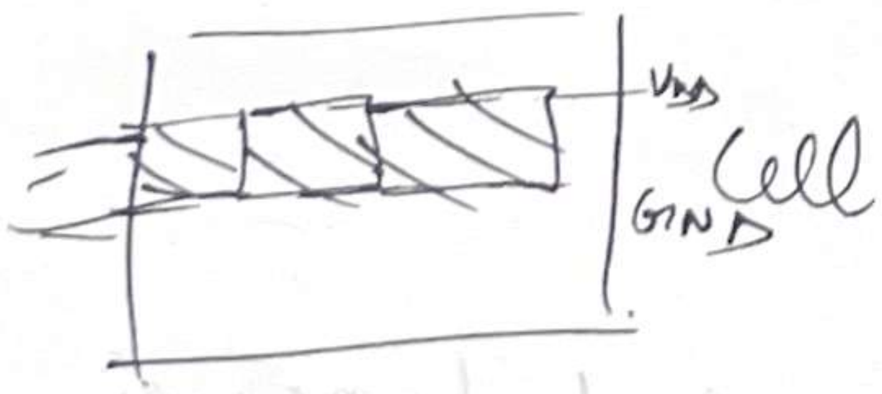
To enable automated placement of cells, and routing of inter-cell connections

* number of cells can be abutted side-by side to form rows.

- Power and ground rails ~~run~~ typically run horizontal to upper and lower boundaries of cell

Neighbouring cells share a common power and ground bus.

The input and output pins are located on upper and lower boundaries of cell.



Custom Design

Entire mask design is done from ~~st~~ scratch

Design reuse is economical

Full custom design is used for memory cell

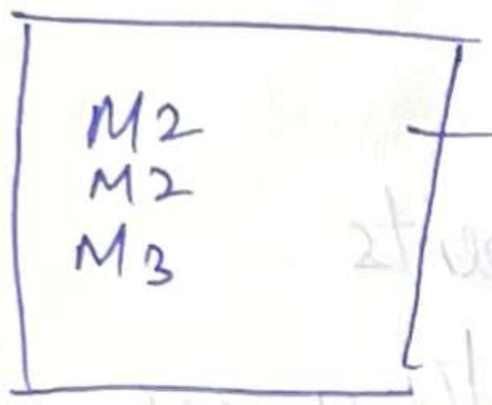
Week 2

Lecture 6 Verilog Language Features Part - 1

```
module name( ports) ;
  input/output declarations
  local net declaration
  parallel statements
end module
```

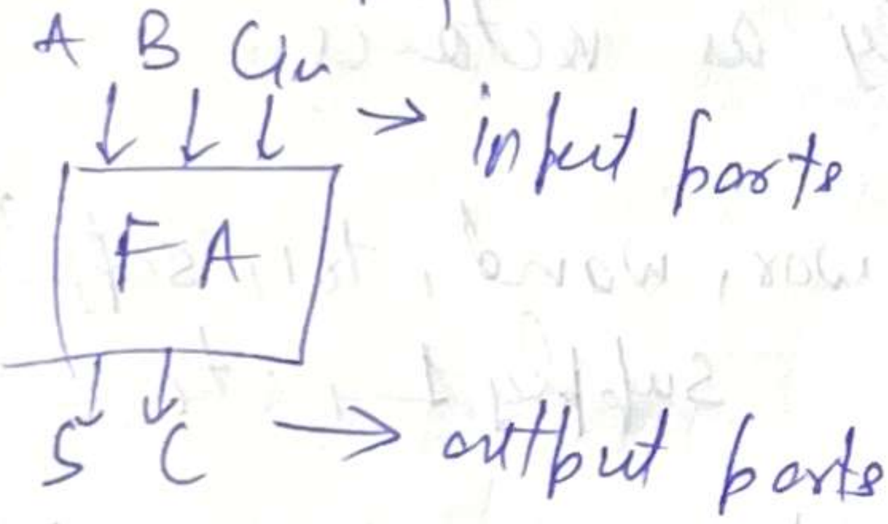
module cannot contain def. of other modules
(Same as functions in CPP)
Can be instantiated within another module
Inst → hierarchy forms

Module



→ 2 copies of M2
1 copy of M1
→ Embedded

Instantiation



assign statement → represents continuous assignment, Variable gets updates as RHS changes.

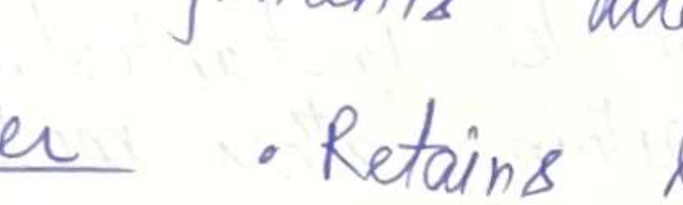
assign variable = expression ;

LHS must be "net" variable ("wire")

RHS can contain both "register" & "net" type variables

assign → models behavioral design style → used to model combinational circuits

Ex assign $t_1 = a \xrightarrow{\&} b$; (and)



→ continuesly deriven

Data Types in Verilog

Net
- must be continuesly deriven
- cannot be used to store value
- Used to model connections blw continuous assignments and instantiations.

Register
- Retains last value assigned to it.
- Used to represent storage elements

↓

connection blw hardware elements

1 bit values unless they are declared explicitly as vectors

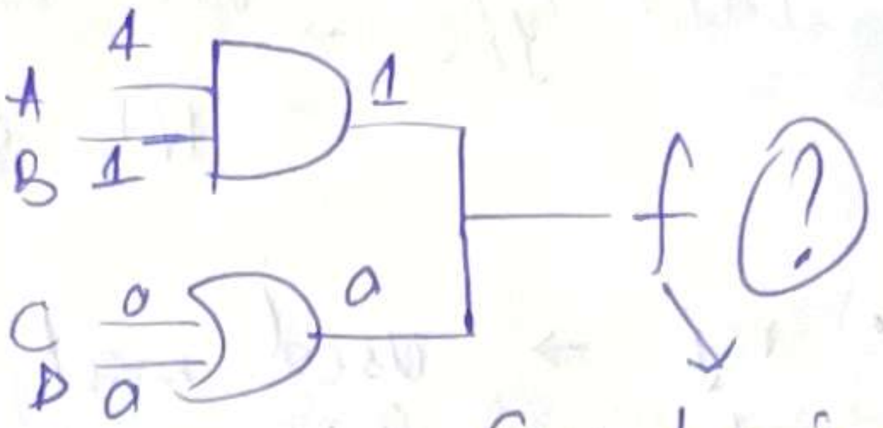
Net data types → wire, wor, wand, tri, supply0, supply1, etc.

(wire, tri) → equivalent. When there are multiple deriveers deriving them, deriver outputs are shorted together ?

wor, wand → OR, AND at junction

Supply0, Supply1 → power connections

```
module use-wire(f, A,B,C,D);
input A, B, C, D;
output f;
wire f;
assign f = A & B;
assign f = C | D;
end
```



if wand f; f = (A & B) & (C | D)

(?)

---

supply 0 , supply 1 → greatest signal strength

Eg
```
supply0 gnd;
supply1 vdd;
```

Data values and Strengths of Signals

4 value levels , 8 strength levels
→ used to resolve conflicts between signal drivers of diff strengths in real circuits

value level	
0	logic 0
1	logic 1
x	unknown
z	high impedance

All unconnected net → z
Register variables → x

Strength	Type
~~Supply~~ supply	Driving
strong	Driving
pull	Driving
large	Storage
weak	Driving
medium	Storage
small	Sto
high z	High Impedance

Strength inc ↑

If two signals of unequal strengths get driven on a wire, stronger signal will prevail

Register Data Type → Var that can hold a value
→ Not same as "Register" in IRL

Types
- reg → used most
  integer → loop counting
  real : store floating point numbers
  time : keep track of simulation time (not used in synthesis)

"reg" → default val = "x"
can be assigned a value in synchronism with clock or otherwise.
- declaration specifies size ~~itself~~ explic (default - 1 bit)

```
reg x,y;          // 1 bit
reg [15:0] bus;   // 16 bit bus
```

→ Treated as unsigned number in arithmetic expressions
→ for used for sequential hardware (counters) etc)
(clock count in code ??)

"integer" : general purpose for manipulating quantities
- more convenient to use in situations like loop counte
- treated as 2's compliment signed integer in arithm expressions
- default size → 32 bits

Synthesis tool tries to determine size using data flow analysis

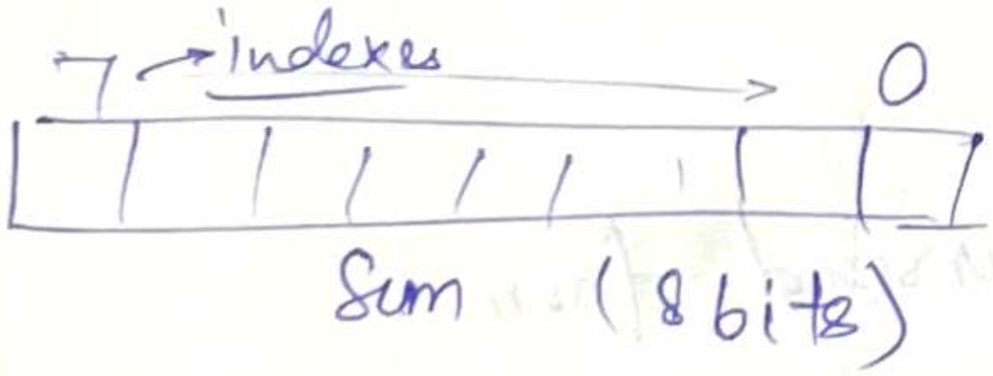
"real" store floating point numbers
When real val assigned to integer, real num rounded off to nearest integer.

time → Sim. is carried out using logical clock called simulation time.
"time" used to store simulation time.

System function "$time" "$time" gives current simulation time

Vectors "net", "reg" types can be declared as vectors, of multiple bit widths
- Declared by specifying range [range 1 : range 2]
  where range 1 most significant bit
  range 2 least ——

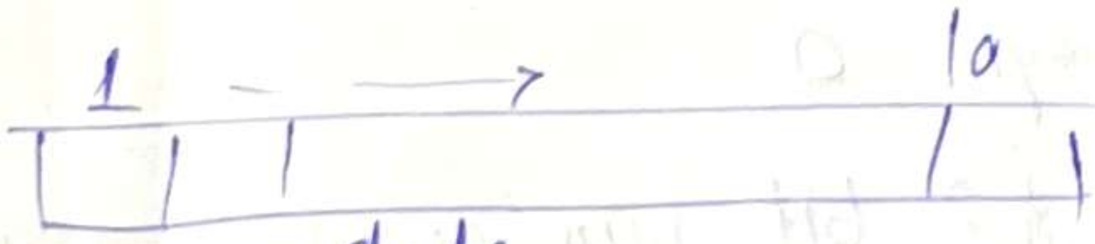
wire [7:0] sum; // MSB = sum[7], LSB is sum[0]



individual bits can be accessed
[⇒ sum[1] →

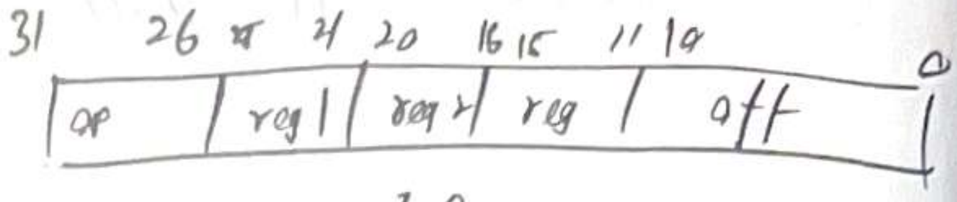
(Array type)

reg [1:10] data;



parts of a vector can be addressed and used in an expression.

Eg
```
reg[31:0] IR;
reg [5:0] op code;
reg [4:0] reg1, reg2, reg3;
reg [15:0] offset;
```



Can also used
```
sum = IR[25:21] + IR[20:16]
// Adding 2-4 bits and storing in sum var
```

Multi Dimensional Arrays and Memories

- multi dimensional arrays of any size can be done in Verilog.

Example `reg [31:0] register_bank [15:0]; // 16 32-bit registe`

- Memories can be modelled in Verilog as 1D array of registe
  Each element of array addressed by single array index.

Specifying Constant Values

Specified in either sized or unsized form

Syntax: `<size>'<base><number>`

```
4'b 0101    // 4 bit = 0101
1'b 0       // Logic 0
12'h B3C    // 12 bit number 1011 0011 1
25          // signed number in 32 bits
```

Parameters

constant with a given name

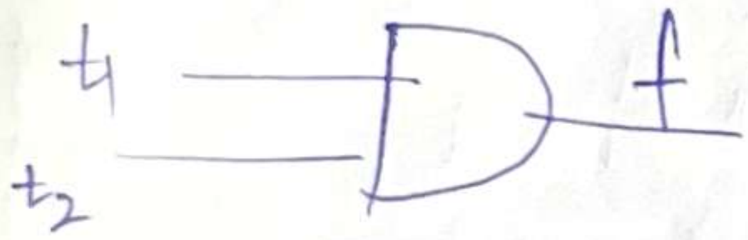
- cannot specify size of parameter. Size decided from const. value itself.

parameter HI = 25, LO = 0;

Verilog Language Features (Part 3)

Predefined Logic Gates in Verilog

- can be instantiated within a module to create a structured design
- Gates respond to logic values (0, 1, x, or z) in logical way



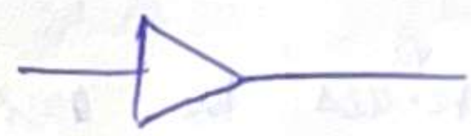
$f = t_1 + t_2$

z → High impedance state

Primitive Gates

buffers

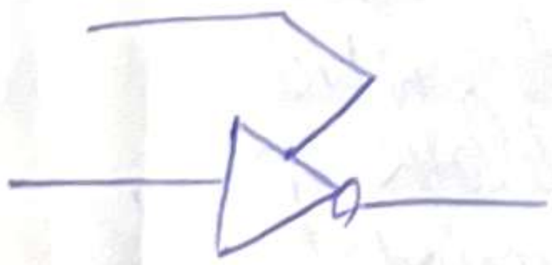
buf (out, in)



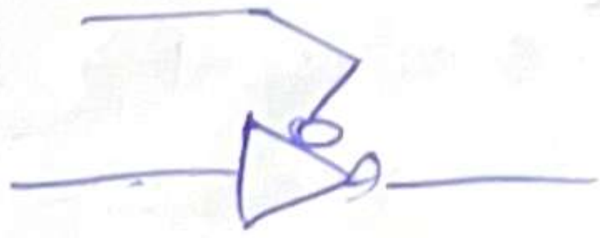
buf if 0 / buf if 1

if ctrl = 0, out = in
= 1, out = z

if ctrl 1, out = in
~~else~~ = 0, out = z



not if 1



not if 0

Some restrictions when instantiating primitive gates:

→ output port must be connected to net (eg wire)
(wire by default)

→ Inputs ports may be net or register type

• Single output but can have any number of i/p (except NOT + BUF)

→ When instantiating a gate, an optional delay may be specified.

• Used for simulation
• Logic synthesis tools ignore the time delays

Eg `and #5 G1(F, A, B);`
↓
delay

Time Scale Directive

Delay values in one module need to be specified in terms of some time unit, while those in some other other modules need to be specified in terms of other time unit.

`timescale <reference time unit> / <time precision>`

reference time unit → unit of measurement for time

time precision → precision to which delays are rounded off during simulation

Valid values → 1, 10, 100

For: 10ns

#5 → 50ns

units: s, ms, us, ps, fs
↓
microsecond

Specifying Connectivity during Instantiation

- When module is instantiated within another module, 2 ways to specify connectivity of signal lines b/w 2 modules.

a) Positional association ⇒ Parameters of module being instantiated are listed in same ~~and~~ order as in original module description.

b) Explicit association (in testbench .OUT(Y)) (Review after testbench)
. parameter of module instant in arbitrary order
: less chance of errors

Hardware Modelling Issues

- In terms of hardware realization, value computed can be assigned to :-
  - A "wire", - + "flip-flop" (edge-triggered storage cell)
  - A "latch" (level triggered storage cell)

"Net" data type always map to "wire" during synthesis
"Register" maps either to "wire" or "storage cell" depending upon the context under which value is assigned.

Lecture-9 Verilog operators

Arithmetic operator

Unary
+ A
- B
-(A+B)

Binary
A + B
A - B
A * B

* → multiply
/ divide
% modulus
** expentiation

Logical Operators (return either TRUE or FALSE)

! - logical negation
&& logical AND
|| logical OR

Value 0 treated FALSE, Any non-zero value TRUE

Relational operators (operate on numbers, and return a Boolean val.)

!=
==
>=
<=
>
<

Bitwise operators: (operate on Bits, return bits)

~ Bitwise NOT
& Bitwise AND
| OR
^ XOR
~^ XNOR

Reduction operators → accept a single word operand and produce a single bit as output. Operates on all the bits within the word.

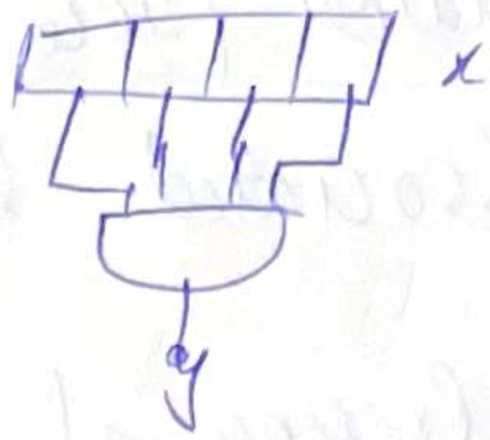
~& NAND
~| NOR

Ex wire [3:0] x ; wire y ;

assign y = &x ;

↓

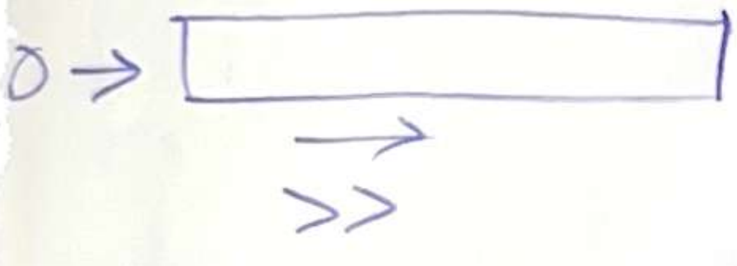
Red. operator



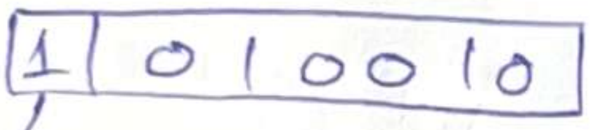
Shift operators:

>> shift right >>> arithmetic shift right

<< shift left



>>> : 2s compliment num sys.



1 (-ve)

0 (+ve)

shift right : /. 2

shift left : * 2

-ve no. : shift in 1

+ve no. : shift in 0

Conditional operator :

cond_expr ? true_expr : false_expr

↓

operation

if is true

else

Concatenation operator : Joins together bits from 2 or more comma separated expression.

{ ... , ... , ... }

Replication operator { n { m } }? Joins n copies of expression m.

Operator precedence ⇒ check web

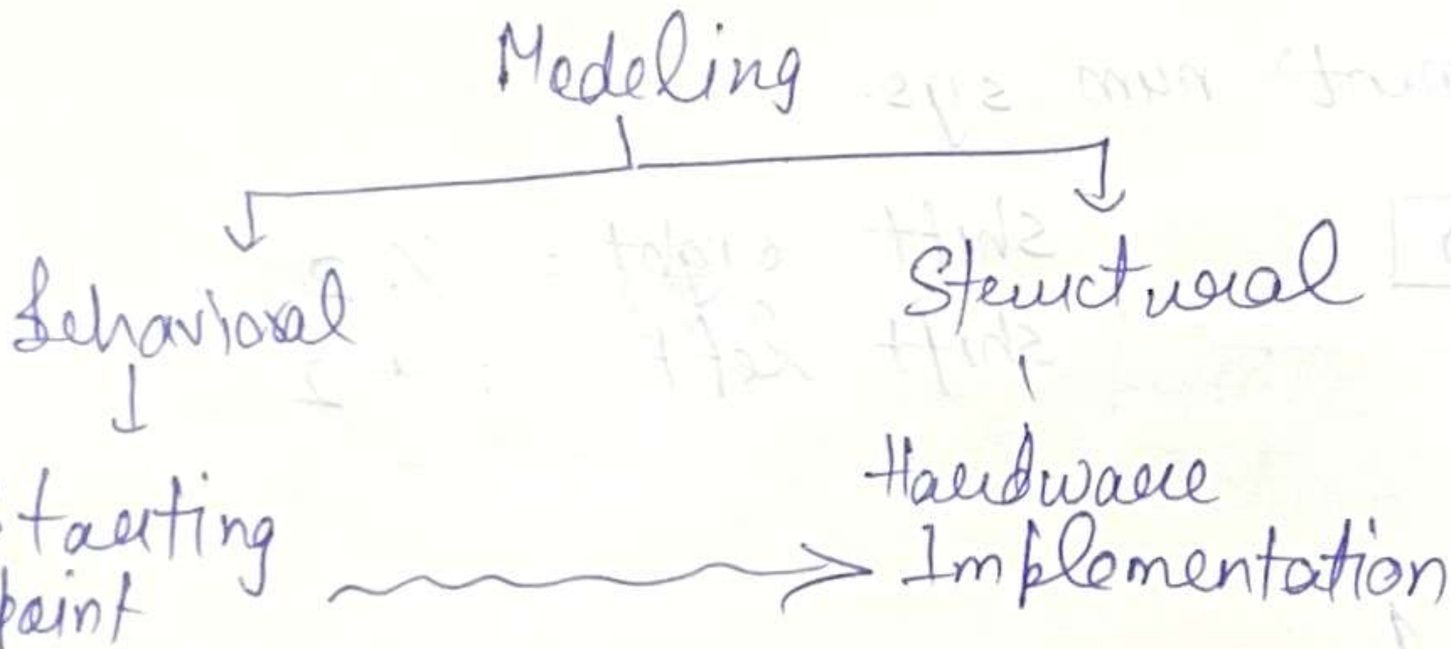
↓
associated left to right

* Presence of 'z or 'x' in reg or wire being used in an arithmetic expression results in whole expression being unknown (x)

Boolean false = 1'b0

True : 1'b1

Lectures 10 Verilog Modelling Examples



Example structural hierarchical description of 16 to 1 MUX.

a) pure behavioral modelling

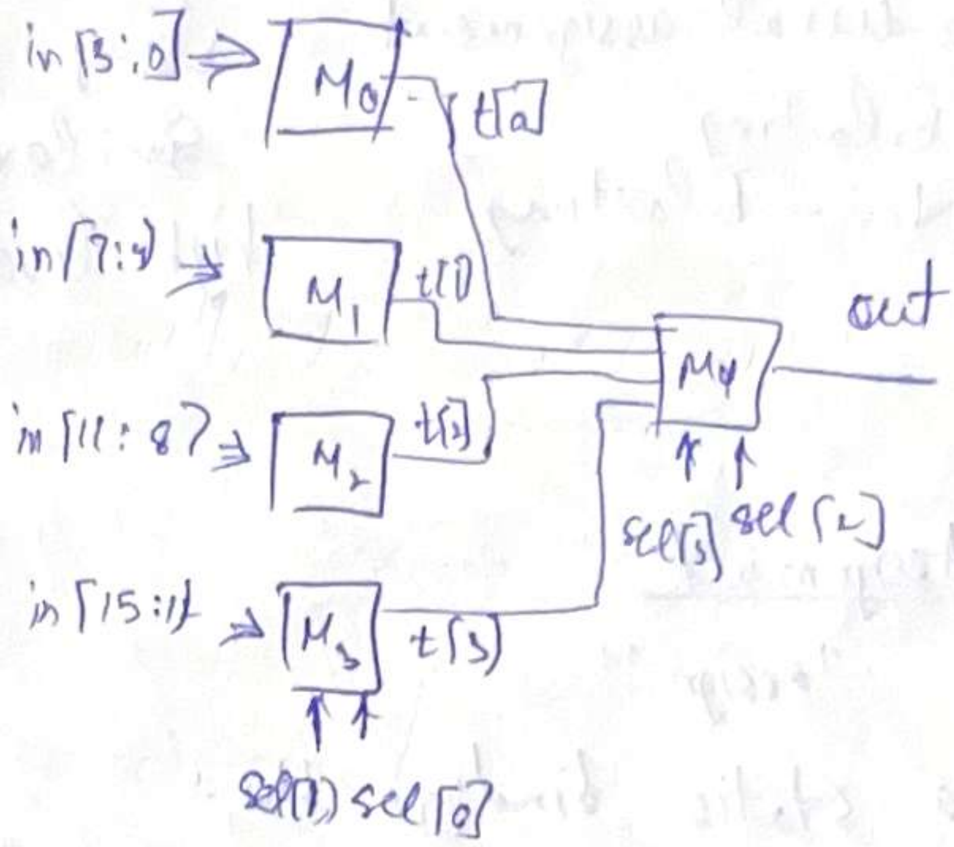
(a) Structural modelling of 4-1 multiplexer specified using behavioral model

(b) Make structural model of 4-1 bus multiplexer using behavioral modelling of 2-to-1 multiplexer

(c) Make structural gate level modelling of 2-1 multiplexer to have complete structural hierarchical descr

Example Behavioral modelling of 4 to 1 MUX + Structural Modelling of 16 to 1 MUX

```
module  mux4to1 (in, sel, out);
input [3:0] in;
input [1:0] sel;
output out;
assign out = in[sel];
endmodule
```

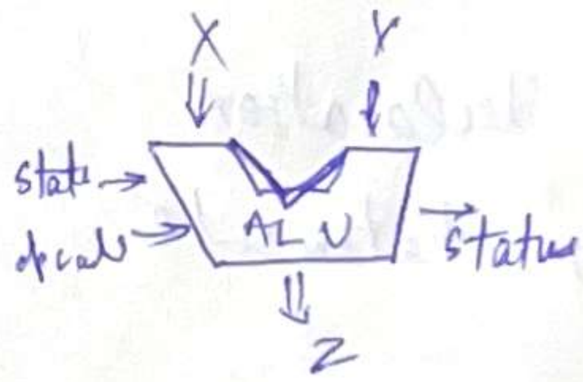



```
module  mux16to1 (in, sel, out);
input[15:0] in;
input[3:0] sel;
output out;
wire[3:0] t;

mux4to1 M0 ( in[3:0], sel[1:0], t[0]);
mux4to1 M1 (in[7:4], sel[1:0], t[1]);
mux4to1 M2 ( in[11:8], sel[1:0], t[2]);
mux4to1 M3 ( in[15:12], sel[1:0], t[3]);
mux4to1 M4 ( t, sel[3:2], out);
endmodule
```

Example 2

Version 1: Behavioral desc. of 16 bit adder



ALU ⇒ Arithmetic & Logic Unit

- Generation of status flags:
  - Sign : whether sum is +ve or -ve
  - Zero sum ⇒ zero
  - Carry : whether there is carry out of last step
  - Parity number of 1s in sum are even or odd
  - Overflow whether sum cannot fit 16 bits

Lec 12 # Description Styles in Verilog

- Data Flow
  - continuous assignment — Using assignment statements
- Behavioral
  - procedural assignment — Using procedural statements
    - Blocking — Similar to a program in
    - Non-Blocking — high level language

Continuous Assignment

- keyword "assign"
- Forms a static binding b/w:
  - "net" being LHS (wire)
  - RHS can be "net" or "register" type var

```
assign a = b + c;
```

Assignment → continuously active
↳ used for combinational circuits
↳ Also for some sequ. circuits

```
module
[ signal declaration
[ assign statements
{ =
  =
  -
  :
endmodule
```

NOTE Whenever there is array reference on RHS with variable index, a MUX is generated by synthesis tool

```
assign = data out = data [sel] ;
```

→ If index constant, wire will be created

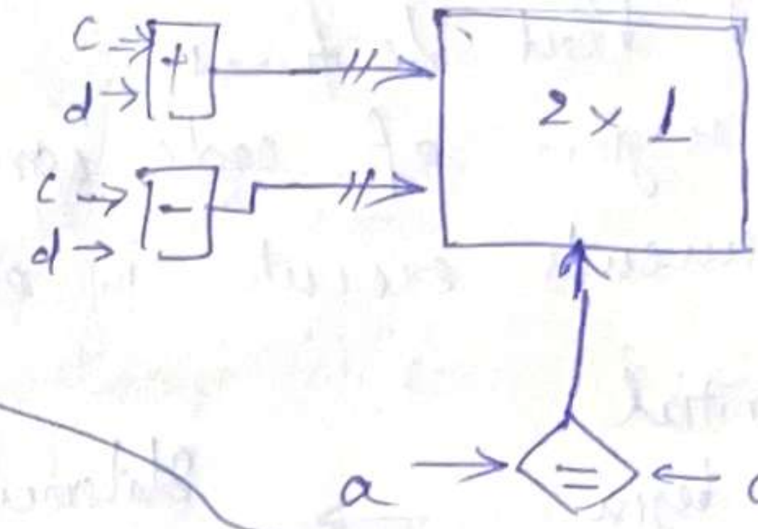
Conditional operator also generates a MUX

assign f = sel ? a : b ; ⟹ For a bus, an array of 2×1 Mux will be generated

if sel == 1, choose a
else choose b

Example assign f = (a == 0) ? (c+d) : (c-d);

if a == 0 ⇒ c+d
else ⇒ c-d




```
module generate_decoder(out, in, select);
 input in;
input [0:1] select;
output [0:3] out;
assign out[select] = in;
endmodule
```

⟶ Non constant index in LHS generates decoder

Whenever a synthesis tool detects a variable index in LHS, a decoder is generated

MUX out = in[sel]

Lecture 13 : Procedural Assignment

- Two kinds supported
  - The "initial" block
    - Executed once at begin
    - Used only in testbenches
  - "Always" block
    - Continuous loop that never terminates

Procedural block defines
→ A region of code containing sequential statements
→ Statements execute in order they are written

initial block
```
initial
  begin
    :
    :
  end
```
→ Statements start at time 0, execute only once
multiple "initial" blocks will start concurrent at time 0..

Used in test benches for simulation :—
- Specifies stimulus to be applied to design-under-test (DUT)
- Specifies how DUT outputs are to be displayed / handled
- Specifies file where waveform info is to be dumped

Always block

Starts at $t = 0$, executes statements inside block repeatedly and never stops.

block
```
always ( condition )
  begin
    =
    =
    =
  end
```
↑ inf loop

Used to model a block of activity that is repeated indefinitely in digital circuit.

Clock signal ⇒ example.

Example

```
module generating-clock;
output reg clk;
initial
  clk = 1'b0; // initialised to 0 at time 0
always
#5 clk = ~ clk; // Toggle after time 5 units
initial
# 500 $finish; // Finish at time 500 unit
endmodule
```

(Now what is "net" and "reg"?)

All these blocks will execute concurrently

Basic syntax of always block:

```
always @(event_expression);
begin
=
=
end
```

@(event_expression) facet req. for both combinational and sequential circuit descriptions.

→ This block will be executed only if event_exp is triggered

Only "reg" can be ~~used~~ assigned within an "initial" or "always" block.

Reason :-

Sequential ~~block~~ "always" block executes only when event expression triggers

At other times, block is doing nothing.

Object being assigned to must remember the last value assigned (not continuously driven).

Any kind of variable may appear in event expression (reg, wire, etc)

reg → hardware register

→ combinational circuit

Sequential statements in Verilog

one or more sequential statements can be present inside an "initial" or "always" block.

Statements executed sequentially

Multiple assign statements inside begin -- end may either execute sequentially or concurrently depending upon type of assign.

Types of assign → blocking (a = b + c;)
→ non blocking (a <= b + c;)

Sequential suits.

(a) begin -- end

```
begin
seq. stat 1;
  "   "  2;
  .
  .
  .
  n;
end
```

If n = 1, begin -- end not req.

(b) if.. else

```
if (<exp 1>)
  seq-stat;
else if (<exp 2>)
  seq stat;
  .
  .
else default state-m;
```

also these can be grouped with begin -- end

(i) case

case (<expression>)
expr 1 : seq - statement ;
expr 2 : seq - — — ;
|
|
default : default. stat. ;
endcase

⟹ Can replace complex if-- else statement for multiway branching

Expression compared to alternative in order they are written.
if name of alternative matched, default is executed

two variations : "case z" and "case x" ↗ imp

↙ treats all 'z' values as dont cares

↓ treats all "x" and "z" as don't cares

(d) 'while' loop

while (<expression>) → executes until <exp> not true
sequential-statement ; → can be single or grp within "begin .. end"

e) 'for' loop

for (expr 1 ; expr 2 ; expr 3)
sequential-statement ; → can be single or grp.

executes as long as expr 2 is true

a) initial condition expr 1

b) terminal condition (expr 2)

c) procedural assignment to change the value of control variable (expr 3)

"for" can be conveniently used to initialise an array or memory

(f) "repeat" loop → execute fixed number of times

```
repeat ( <expression> )
    Sequential statement ;
```

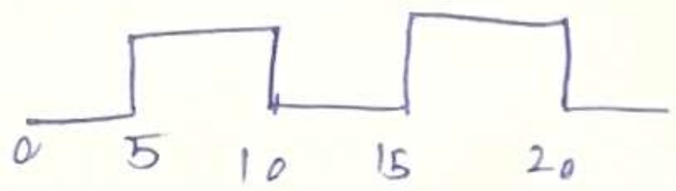
cannot be used to loop on a general logical expression like "while"

exp can be const, var, or signal val

- if it is variable or signal val, it is evaluated only when loop starts and not during execution of loop.

Example 100 clk pulses generated.

```
reg clock;
initial
  begin
    clock = 1'b0;
    repeat (100) ⟶ exactly 100 clk pulses generated
      #5 clock = ~clock;
  end
```



(g) 'forever' loop

```
forever
  sequential-statement;
```

forever construct does not use any expression and executes forever until $finish is encountered in test bench.

Used with timing specified

- If delay not specified, simulator would execute this indefinitely without advancing $time.
- Rest of design will never be executed (if time not

// clock generation using "forever" construct

```
reg clk;
initial
  begin
    clk = 1'b0;
    forever #5 clk = ~clk; // 1a unit clk period
  end
```

Other constructs available.

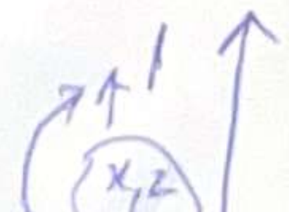
#(time-value) → makes a block suspend for 'time-value' units of time
→ time unit can be specified using 'timescale command

@(event-expression) → makes a block suspend until "event-expression" triggers
⊛ keywords associated with e-e ahead

② (event. expression)
specifies the event that is required to resume execution of the procedural block.

- The event can be any one of following:-
  a) change of signal value
  b) positive or neg. edge occuring on signal (posedge or negedge)
  c) List of above mentioned events separated by "or" or comma

posedge → transition from {0,x,z} to 1 and from {0,z,x} to 1



negedge → 

→ negedge

Example

@(in) // "in" changes

@(a or b or c)

- @(posedge clk) // positive edge of 'clk'

- @(*) ⇒ // any variable changes

// D flip flop with synchronous set and reset

```
module dff ( q, qbar, d, set, reset, clk);
  input  d, set, reset , clk;
  output reg q; output qbar;
  assign qbar = ~q;
  always @(posedge clk)
  begin
  if (reset==0)  q<=0;
  else if (set==0)  q<=1;
  else q<=d;
  end
endmodule
```

for ~~ass~~ asynchronous

modify always @(posedge clk or negedge set or negedge reset)

Lecture - 15

Procedural Assignment (Examples)

①
```
module max21 ( in1, in0, s, f);
input in1, in0, s;
output reg f;

always @( in1 or in0 or s)  → comma & can be used also
  if(s)
    f = in1;
  else
    f = in0;
endmodule
```

② 4 D flip flop

```
module dff_negedge( D, clock, Q, Qbar );
input D, clock;
output reg Q, Qbar;
always @(negedge clock)
  begin
    Q = D;
    Qbar = ~D;
  end
endmodule
```

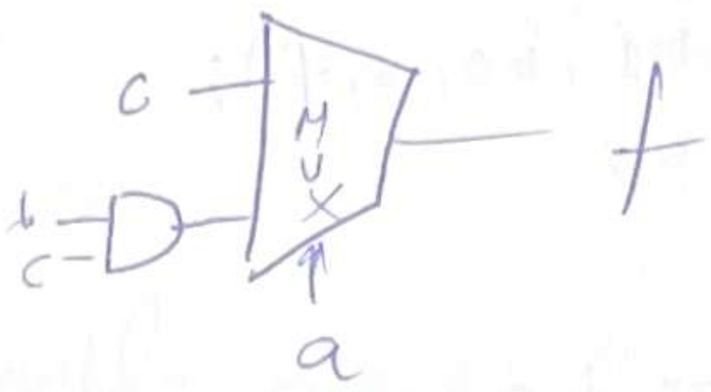
When a "case" statement is incompletly decoded (specified), the synthesis tool will infer the need for a latch to hold the residual output when the select bits take the unspecified value

③
```
module xyz( f, a, b, c);
input a, b, c;
output reg f;
always @ (*)
  if( a == 1) f = b & c;
end module
```


→ for a = 0, value of f is unspecified

```
module xyz( f, a, b, c);
 input a, b, c;
 output reg f;
 always @(*)
  begin
    f = c;
    if(a==1) f = b & c;
  end
end module
```

a = 0



Working of Peiority Encoder ???

1) 2 bit comparator

```
module compare ( A1, A0, B1, B0, lt, gt, eq);
input A1, A0, B1, B0;
output reg lt, gt, eq;
always @( A1, A0, B1, B0)
  begin
    lt = ( {A1, A0} < {B1, B0});
    gt = ( {A1, A0} > {B1, B0});
    eq = ( {A1, A0} == {B1, B0});
  end
end module
```

→ continuous assignment

Lecture 17

Procedural Assignment

- used to update variables of types "reg", "integer", "real", "time"
- value assigned to variable remains unchanged until another procedural assign statement assigns new value to variable

(Different from continuous assignment (using "assign") that results in expression on RHS to continuously drive "net" type variable on left.

Types of procedural assignment statements:

a) Blocking (denoted by "=")

b) Non-blocking (denoted by "<=")

Left hand side of a procedural assignment statement can be one of:

a) register type var ("reg", "integer", "real" or "time")

b) bit select of these variables (a[15])

c) part sellct (IR[31:26])

d) concatenation

RHS can be any express. consisting of "net" and "register" that evaluate to value

Procedural assignment statement can only appear within procedural blocks. ("initial" or "always")

{a[5], b[3:2], c} = x + y;

a, b, c provided these are reg type var

x + y can be net or register

Blocking Assignment (=)

variable_name = [delay_or_event_control] expression;
//

blocking assignment

Executed in order they are specified in procedural block.
- target of assignments get updated before next seq. statement in procedural block is executed.
- they do not block execution of statements in other procedural blocks

Recommended style for ~~mod~~ combinational logic

Non Blocking Assignment (<=)

variable_name <= [delay_or_eventexpression] expression;

Allows scheduling of assignments without blocking execution of statements that follow within procedural block.

- Assignment to target gets scheduled for end of simulation cycle (at end of procedural block)
- Statements subsequent to instruction under consideration are not blocked by assignment
- Allows concurrent procedural assignment, suitable for sequential logic

Recommended style for modelling sequential logic.

- ~~Several~~ "reg" type variables can be assigned synchronously under the control of common clock.

integer a, b, c;
initial
begin
a=10; b=20; c=~~20~~ 15;
end
initial
begin
a <= #5 b + c;
b <= #5 a + 5;
c <= #5 a - b;
end

(Good for synch seq-circuit)

→ they will be evaluated togetn

initially a = 10, b = 20, c = 15;
a = 35 at t = 5
b = 15 at t = 5
c = -10 at t = 5

Example Swapping "a" and "b"

always @ (posedge clk)
a <= b;
always @ (posedge clk
b <= a;

Some Rules to be followed

- blocking and non-blocking not mixed in same "always" block.
- Verilog synthesiser ignores the delays specified in procedural assignment statement.
→ may lead to fuctional mismatch.

Not permissible → x = x + 5;
x <= y;

(Lect 17 Part-2)

More examples // up-down counter (synchronous clear)

```
module counter (mode, clr, ld, d_in, clk, count);
                                  └→ load input
  input mode, clr, ld, clk;
  input [0:7] d_in;
  output reg [0:7] count;

  always @ (posedge clk)
    if (ld)          count <= d_in;   // if ld = 1, Then —
    else if (clr)    count <= 0;      // if clr = 1, Then —
    else if (mode)   count <= count + 1;
    else             count <= count - 1;
endmodule
```

// Parameterised design : N bit counter

```
module counter (clear, clock, count);
  parameter N = 7;
  input clear, clock
  output reg [0:N] count;

  always @ (negedge clock)
    if (clear)
      count <= 0;
    else
      count <= count + 1;
endmodule
```

// a ring counter

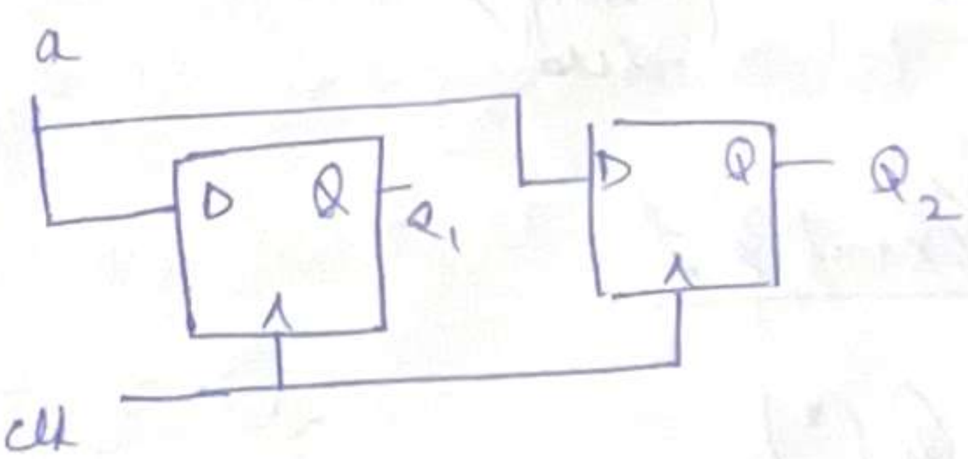
```
module ring_counter (clk, init, count);
 input clk, init;
 output reg [7:0] count;
 always @(posedge clk)
  begin
     if (init) count = 8'b10000000;
     else begin
          count <= count << 1;
          count[0] <= count[7];
     end
  end
endmodule
```

shift left by 1 → both will work simultaneously

or use `count = {count[6:0], count[7]};`

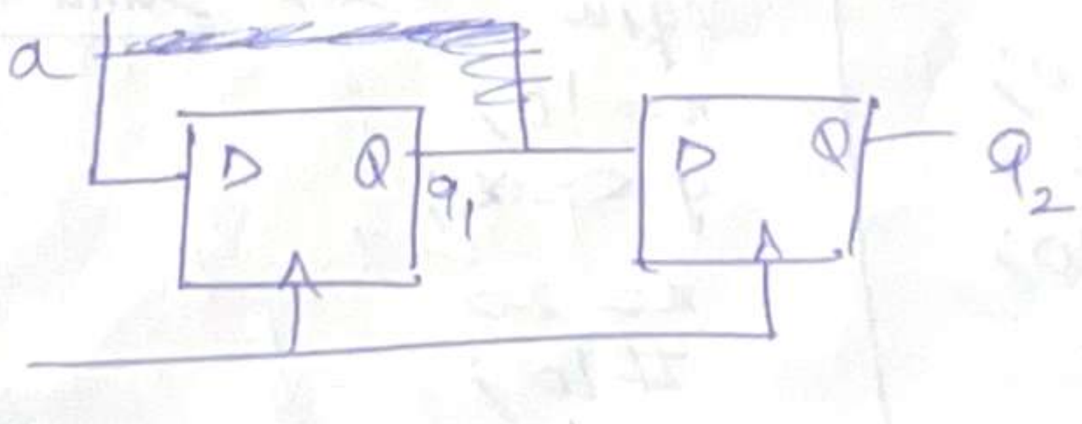
Ques 18

```
always @(posedge clk)
  begin
    q1 = a;
    q2 = q1;
  end
```



Now

```
q1 <= a;
q2 <= q1;
```



Lec 19 part-4 Mixing blocking / nonblocking not recommended.

Basic Idea

- possible to combine both blocking & nonblocking assignments in same procedural block (viz. "always")
  - Simulator or synthesis tool supports this.
- Interpretation not straightforward - we not recc for designers

Example 1

```
always@(*)
  begin
  x = 10;
  x = 20;
  y = x;
  #10;
end
```

```
begin
x = 10;
x = 20;
y <= x;
#10;
end
```

→ Same result, no delay in blocking assigns
so x = 20 at t = 0
Then y = 20

Example 2

```
always @(*)
  begin
    x = 10;
    y = x;
    x = 20;
  #10;
end
```

```
begin
x = 10;
y <= x;
x = 20
#10;
end
```

→ Same result

Generate Blocks "generate" "endgenerate"

generate statement allows Verilog code to be generated dynamically before the simulation or synthesis begins.

— convenient for parameterized module description

Generate instantiations can be used for various Verilog blocks:

- modules, user-defined primitives, gates, continuous assignments, 'initial', 'always' blocks etc
- Generated instances have unique identifier names and can be referenced hierarchically.

Special "genvar" variables:

- used to ~~generate~~ ~~var~~ declare variables that are used only in evaluation of generate block.
- These variables do not exist during simulation or synthesis
- value of 'genvar' can be defined only in generate loop.
- Every generate loop is assigned a name, so that variables inside the generate loop can be referenced hierarchically

Example Bit wise XOR (f, a, b);

```
module xor_bitwise (f, a, b);
  parameter N = 16;
  input [N-1:0] a, b;
  output [N-1:0] f;
  genvar p;
  generate for (p=0; p<N; p=p+1)
    begin xorlp
      xor XOR (f[p], a[p], b[p]);
    end
  endgenerate
endmodule
```

this name was given to generate loop

(?)

→ Relative hierarchical names will be

xorlp[0].XOR, — — xorlp[15].XOR

Lec20 User Defined Primitives (UDP)

used to define custom Verilog primitive by use of lookup table

They can specify:
- Truth table for combinational functions
- State table for seq. func.
- Don't care, rising and falling edges can be specified.

For combinational func, TT entries of the form

<input 1> <input 2> . . - <input N> : <output>;

For sequential state table entries

<input 1> <input 2> - - <input N> : <present_state : <next_s

Rules

Input terminals to UDP can only be scalar variables
- multiple i/p terminals can be used
- input terminals are declared as "input",
- input entries in table must be in same order as "input" terminal list.

only one scalar output terminal must be used
- output terminal must appear in beginning of terminal list
- For combinational UDPs, o/p terminal declared as "output"
- For sequential UDPs, o/p terminal declared as "reg".

* For sequential UDPs, state can be initialised with "initial" statement (optional)

```
input a, b, c;
a b c : <output>
```

Guideline
- UDPs model functionality only
  — Do not model timing or process tech.
- Functional block can be modeled as UDP only if it has exactly one output
  - If block has more than one o/p, it has to be modeled as a module.
  - Alternative, multiple UDPs can be used, one per output.
- Inside simulator, a UDP is typically implemented as a lookup table in memory
- UDP state tables should be specified as completely as possible.
  - For unspecified cases, output set to 'x'.

Example // Full adder sum generation using UDP

```
primitive udp-sum ( sum, a, b, c);
  input a, b, c;
  output sum;
  table
  // a  b  c    sum
     0  0  0 :  0;
     0  0  1 :  1;
     '  '  ' '  '
     '  '  ' '  '
     '  '  ' '  '
  end table
end primitive
```

We can also specify don't care input combinations as "?"

```
// Instantiating UDPs.
module full_adderu (sum, cout, a, b, c);
   input  a, b, c;
   output sum, cout;

   udp_sum SUM (sum, a, b, c);
   udp_cy  CARRY (cout, a, b, c);
endmodule

// 4 to 1 multiplexer
primitive udp_mux41 (f, s0, s1, s2 i0, i1, i2, i3);
input s0, s1, i0, i1, i2, i3;
output f;
table
//   s0 s1   i0 i1 i2 i3 : f
     0  0    1  ?  ?  ?  : 1
     0  0    0  ?  ?  ?  : 0
     1  0    ?  1  ?  ?  : 1
     1  0    ?  0  ?  ?  : 0
     0  1    ?  ?  1  ?  : 1
     0  1    ?  ?  0  ?  : 0
     1  1    ?  ?  ?  1  : 1
     1  1    ?  ?  ?  0  : 0
endtable
endprimitive
```

// A T flip-flop

primitive TFF(q, clk, clr);
  input clk, clr;
  output reg q;
  table
  // clk clr   q   q_new
      ?   1   : ? : 0;
      ?   (10) : ? : -; // ignore -ve edge of "clr"
     (10) 0   : 1 : 0;
     (10) 0   : 0 : 1;
     (0?) 0   : ? : -; // ignore +ve edge of "clk"
  endtable
endprimitive

(-) ↳ keep the prev state

x → invalid

Some Rules to follow

- "?" cannot be specified in o/p field (Dont care)
- "-" → No change only in o/p
- "r" Same as (01) → rising edge
- "f" Same as (10) → falling edge
- "*" → any value change in signal

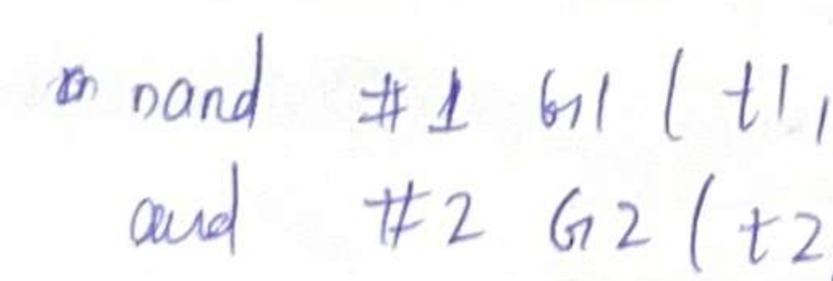
Lecture 21 Verilog Test Bench

- procedural block that executes only once
- used for simulation.
- generates clk, reset, req. test vectors for a given design-under-test (DUT)
- can monitor DUT outputs and present them in a way as specified by creator
  - Print values of single lines
  - Dump values in file from where waveforms can be viewed

- inputs & o/p s of DUT must be connected to test bench.

- Test bench uses "initial" procedural block that executes only once
- "always" to generate test inputs, like clk signal

Test Bench



Description.

```
module example (A,B,C,D,E,F,Y);
    input  A,B,C,D,E,F;
    output Y;
    wire t1,t2,t3;

    nand  #1 G1 (t1,A,B);
    and   #2 G2 (t2,C,~B,D);
    nor   #1 G3(t3,E,F);
    nand  #1 G4 (Y,t1,t2,t3);
endmodule
```

```
module testbench;
  reg A,B,C,D,E,F; wire Y;
  example DUT (A,B,C,D,E,F,Y);

initial
  begin
    $monitor ( $time, " A = %b, B = %b, C= %b, D= %b, E= %b,
                F= %b, Y= %b", A,B,C,D,E,F,Y);
    #5  A = 1; B=0; C=0; D=1; E=0; F=0;
    #5  A=0; B=0; C=1; D=1; E=0; F=0;
    #5  A=1; C=0; (// other var not changing)
    #5 $finish;
  end
endmodule
```

%b → binary

How to write
- create dummy template.
- declare inputs to design-under-test (DUT) as "reg" and outputs as "wire".

Because we have to initialise DUT inputs inside procedural blocks typically "initial", where only "reg" can be assigned.

- Instantiate DUT
- Assign known val to DUT inputs
- Monitor DUT o/ps for functional verification

For Synchronous seq. circuits → need clock generation logic,

Simulator directives: $display, $monitor, $dumpfile, $dumpvar, $finish etc.

→ Dont need test bench for synthesis

- $display ("<format>" , expr1 , expr2 , ---) ;
  — print immediate values of text or variables to stdout
  format specifiers "b" (binary), "h" (hexa)

- $monitor ("<format>", var1, var2, ---) ;
  only print values whenever value of some variable in given list changes.
  (functionality of event-driven input)

$finish → terminate the simulation

#100 $finish // After $time = 100 then stop simulation.

- $dumpfile (<filename>);
  ↳ file for storing values of selected vars
  extension .vcd (value change Dump)
  contains info about any value changes on selected variable.

$dumpoff → stop dumping of variables
All variable dumped with "x" and next change of variables will not be dumped.

$dumpon;
↳ starts previously stopped dumping of variables

$dumpvars (level , list_of_vars);

— specifies variables to be dumped

- var level = 0 → all var from models will be dumped
all module instances which have variables will also be dumped.

- level = 1 → only listed variables and variables of listed module will be dumped

$dumpall
↳ current values of all variables will be written.
irrespective of change.

- $dumplimit (filesize); → used to set max size of .vcd file

$dumpvars (0, module1 , module2); ⇒ All vars in 1, 2 will be dumped

(1, a, b, c)

↓ order these

Lecture 22 Writing Test benches

```
module testbench;
  reg a, b, c; wire sum, cout;
 full_adder FA( sum, cout, a, b, c);

 initial
   begin
     $monitor ( $time, " a =%b , b = %b, c = %b, sum = %b, cout=%b",
                a, b, c, sum, cout );

     #5 a = 0; b = 0; c = 1;
     #5 b = 1 ;
     #5 a = 1;
     #5 a = 0; b = 0; c = 0;
     #5 $finish;
   end
endmodule
```

Print all

```
integer i;

initial
begin
  for ( i = 0; i < 8 ; i = i + 1)
    begin
      {a, b, c} = i;  #5;
      $display (" T = %2d, a = %b, b = %b,
                c = %b, d = %b , sum = %b
                cout = %b", $time, a, b, c,
                sum, cout);
    end
  #5 $finish;
end
endmodule
```

Lecture 33 MODELLING FINITE STATE MACHINES

FSM → can be represented in form of state table or in form of state transition diagram.

Mealy and Moore FSM types

A deterministic FSM can be mathematically defines as a 5-tuple

$(\Sigma, \Gamma, S, S_0, \delta, \omega)$

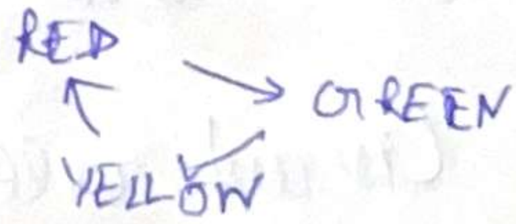
Σ → set of i/p comb. , Γ → set of o/p comb

S → finite set of states , $S_0 \in S$ → initial state, δ state transition function, ω → output function.

Here , $\delta : S \times \Sigma \rightarrow S$

For Mealy machine , $\omega : S \times \Sigma \rightarrow \Gamma$ (output depends on state + & inputs)

Moore , $\omega : S \rightarrow \Gamma$ (depends only on state)




```
module cyclic_lamp ( clock, light);
input clock ;
output reg [0:2] light;
parameter S0 = 0, S1 = 1, S2 = 2;
parameter RED = 3'b100 , GREEN = 3'b010, YELLOW = 3'b001;
reg [0:1] state ;

always @ (posedge clock)
   case (state)
   S0 : begin
        light <= GREEN ; state <= S1 ;
        end

   S1 : begin
        light <= YELLOW ; state <= S2 ;
        end

   S2 : begin
        light <= RED ; state <= S0 ;
        end

   default : begin
             light <= RED;
             state <= S0;
             end
   endcase
endmodule
```

5 ff generated 2 for state 13 for lights
due to non-blocking assignments

But actually we do not need separate ff for outputs as outputs can be directly generated from state.

How to achieve this?
- Make assignment states to light in separate "always" block.
- Use blocking assignment triggered by state change, not by clock

---

Example 2 (L-14)

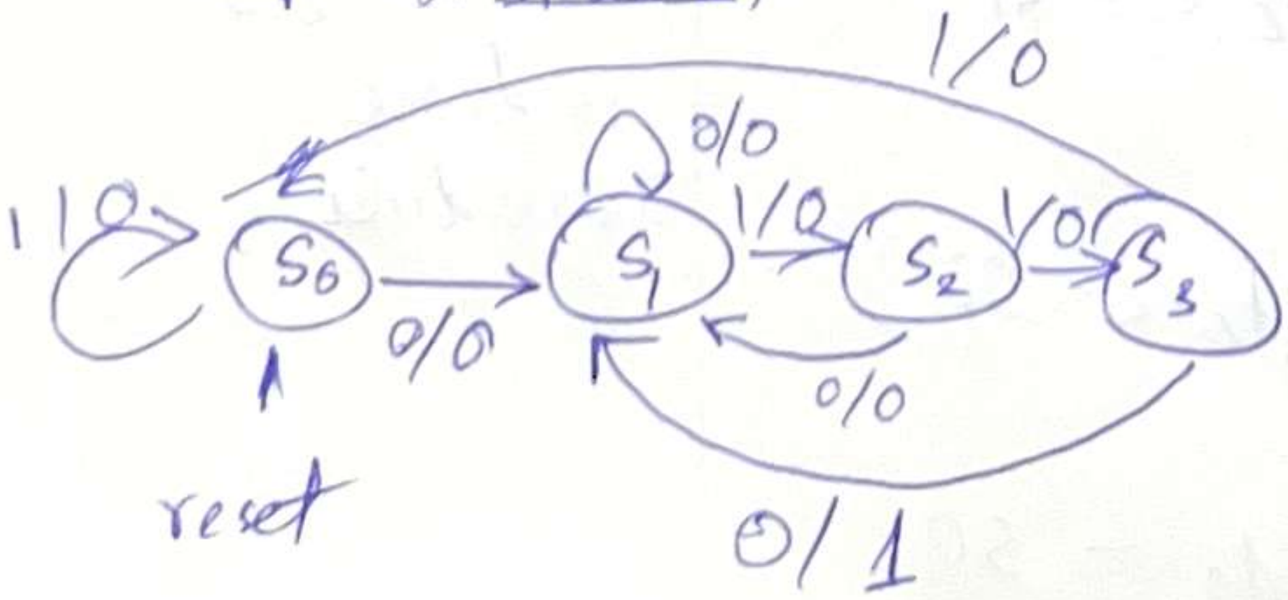
- Design of a serial parity detector
  — A continuous stream of bits is fed to circuit in synchronism with clock.

Circuit will generate bit stream as output, where 0 indicate even no. of ones and as 1 > odd no of 1

write the code yourself

Sequence Detector

Detect pattern 0110 → given output 1 else 0
overlap considered



```
module detect ( clk, x, reset, z);
input x, clk, reset;
output reg z;
parameter S0=0 &, S1 = 1, S2=2, S3=3;
reg [0:1] PS, NS;

always @ (posedge clk or posedge reset)
   if (reset)    PS <= S0;
   else          PS <= NS;

always @ (PS, x)
   case ( PS)
     S0 : begin
          z = x ? 0 : 0;

          NS = x ? S0 : S0;
          end
     S1 : begin
          z = x ? 0 : 0;
          NS = x ? S2 : S1;
          end
     S2 : begin
          z = x ? 0 : 0;
          NS = x ? S3 : S1;
          end
     S3: begin
          z = x ? 0 : 1;
          NS = x ? S0 : S1;
          end
     end case
endmodule
```

L-25 Datapath and Controller Design (Part 1)

In complex digital system, the hardware is typically partitioned into two parts:

a) Data path, which consists of functional units where all computations are carried out [Registers, multiplexers, bus, adders, multipliers, counters]

b) Control path, implements FSM, provides control signals to data path in proper sequence.

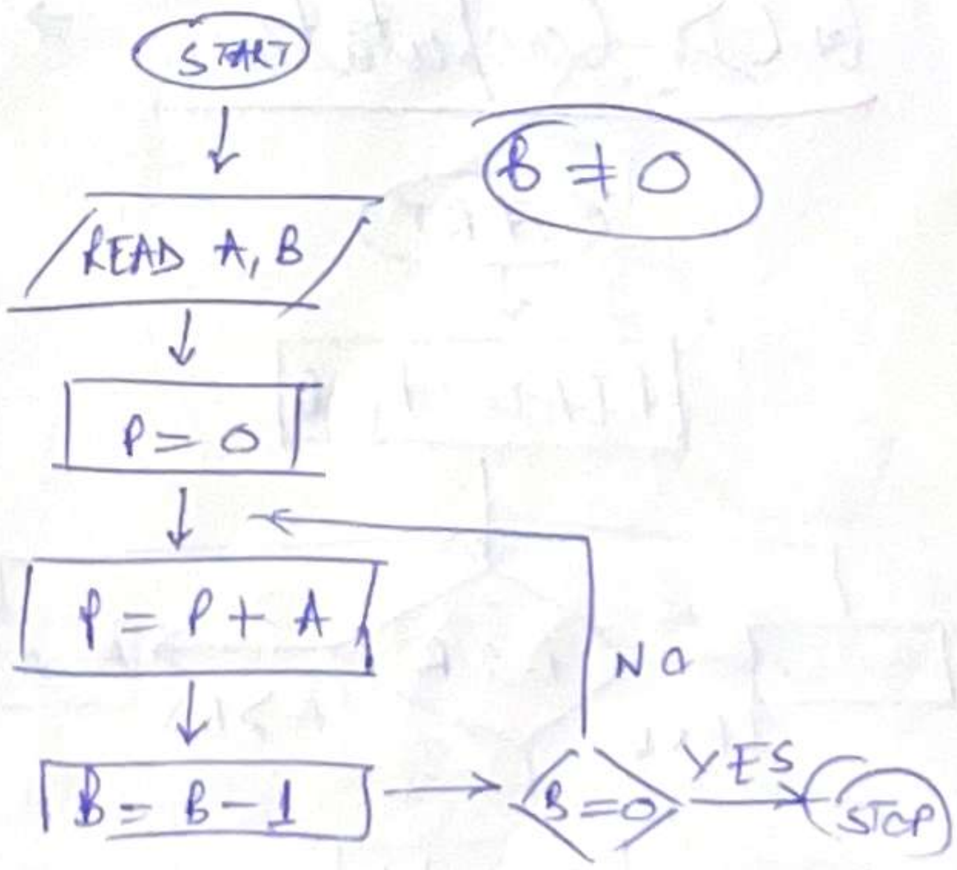
In response to control signals, various operations are carried out by data path.

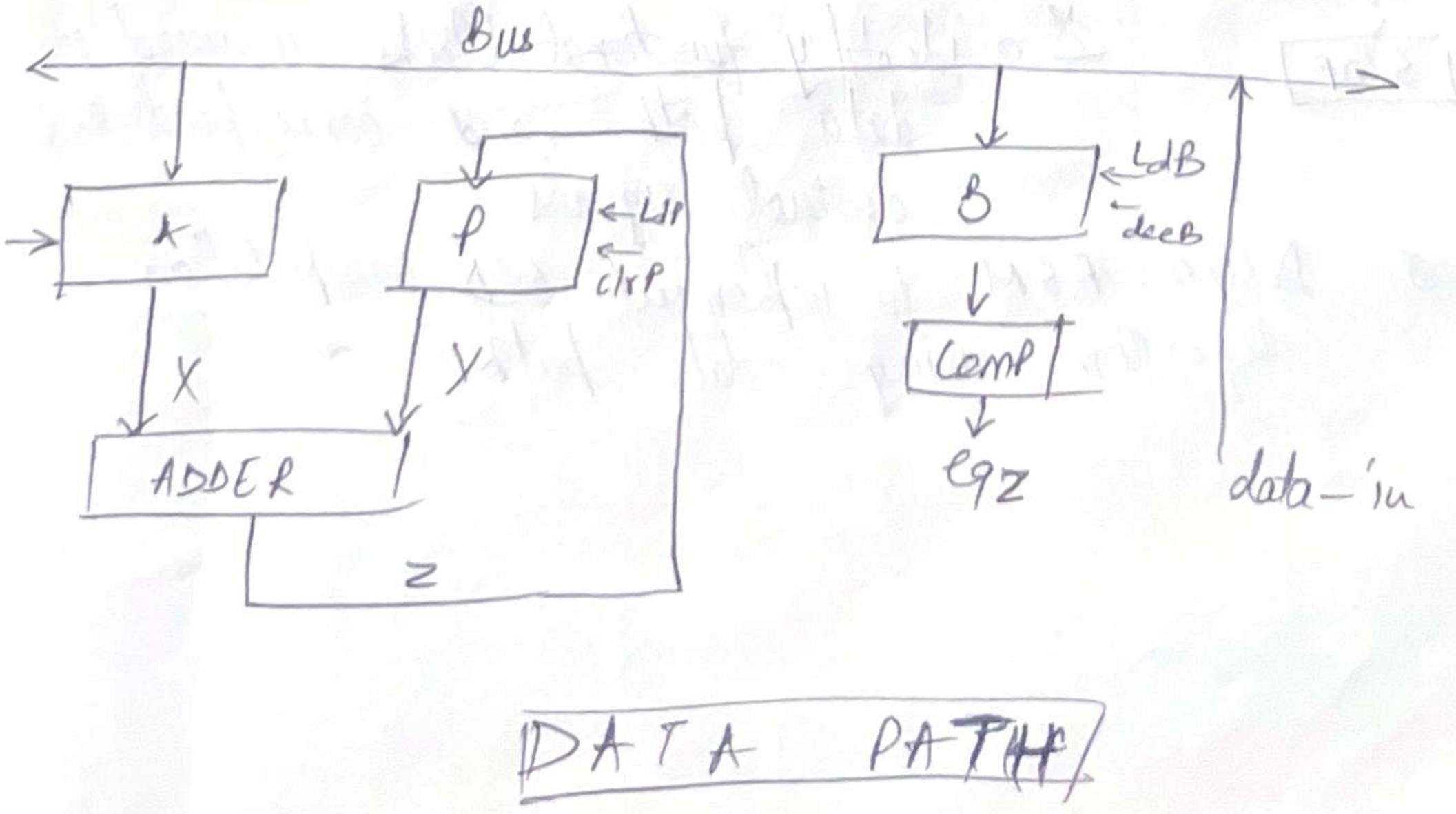
Also takes inputs from data path regarding various status info.

Example 1 Multiplication by Repeated Addition

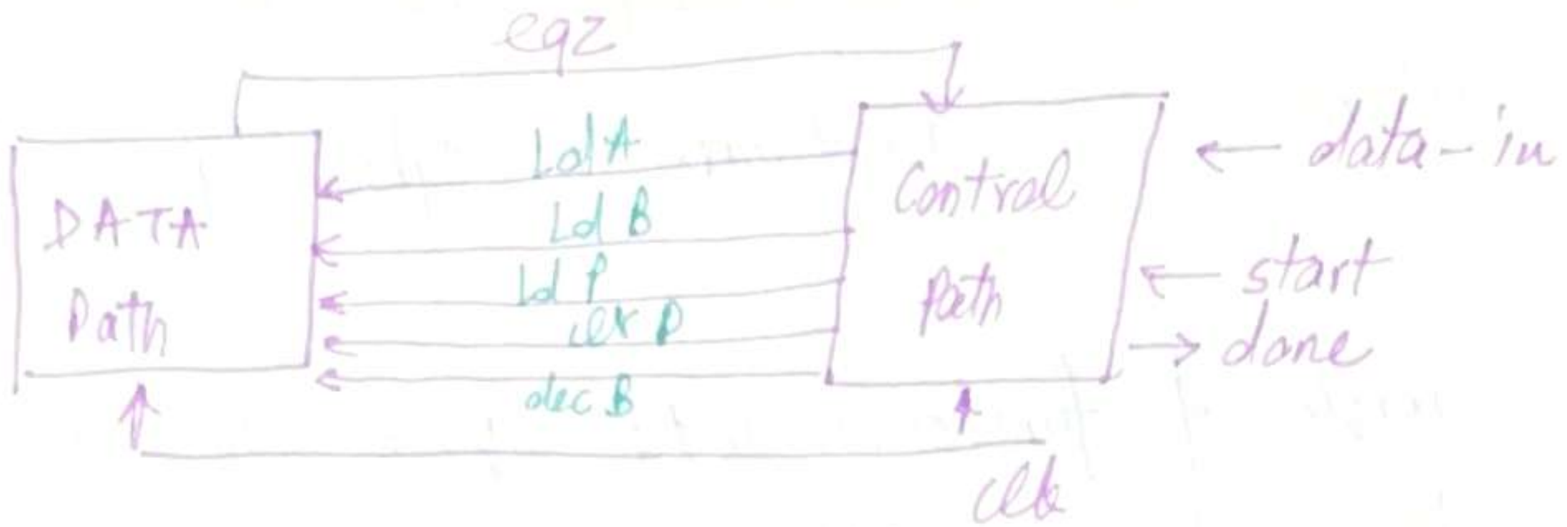
We identify functional blocks required in data path, and corresponding control signals

Then we design FSM to implement multiplication algo using data path





DATA PATH



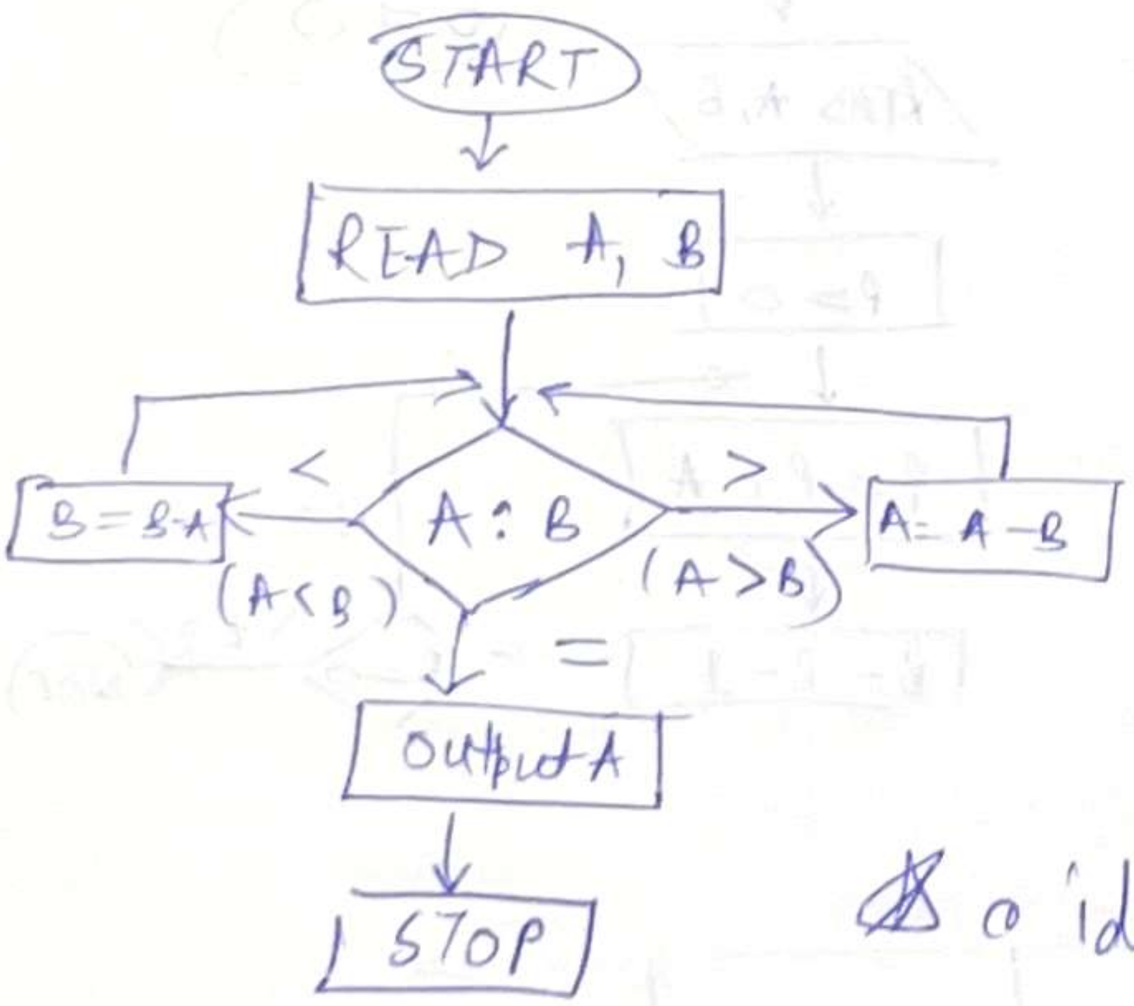
Part 2 28 Dec

Better style of modelling Data/ Control path

→ only trigger the state change in clock activated "always" block

- In seperate "always" block using blocking assignments compute next state
- As in previous example, in separate "always" block, gener control signals for data path

GCD Computation → simple algo using repeated subtraction



26 1 65

26 < 65 ⇒
A B

B = 39

B = 39 - 26

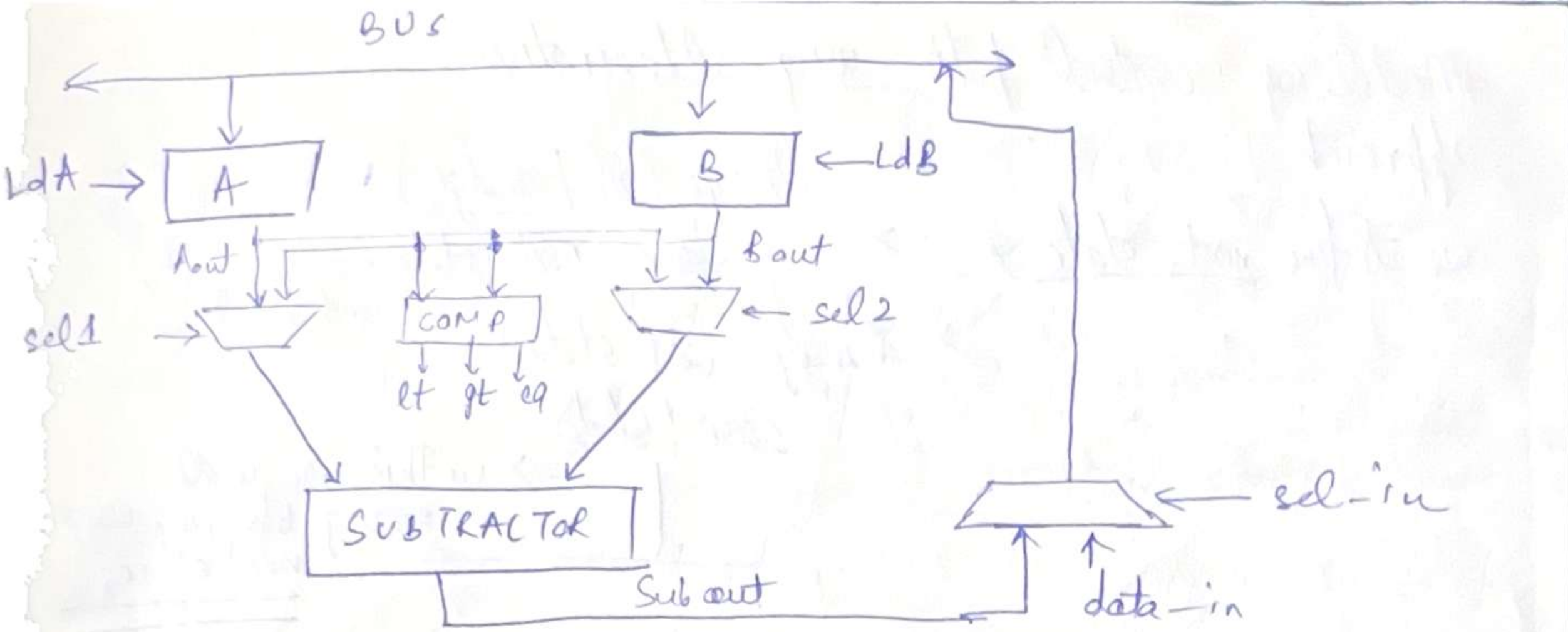
A = 26 - B

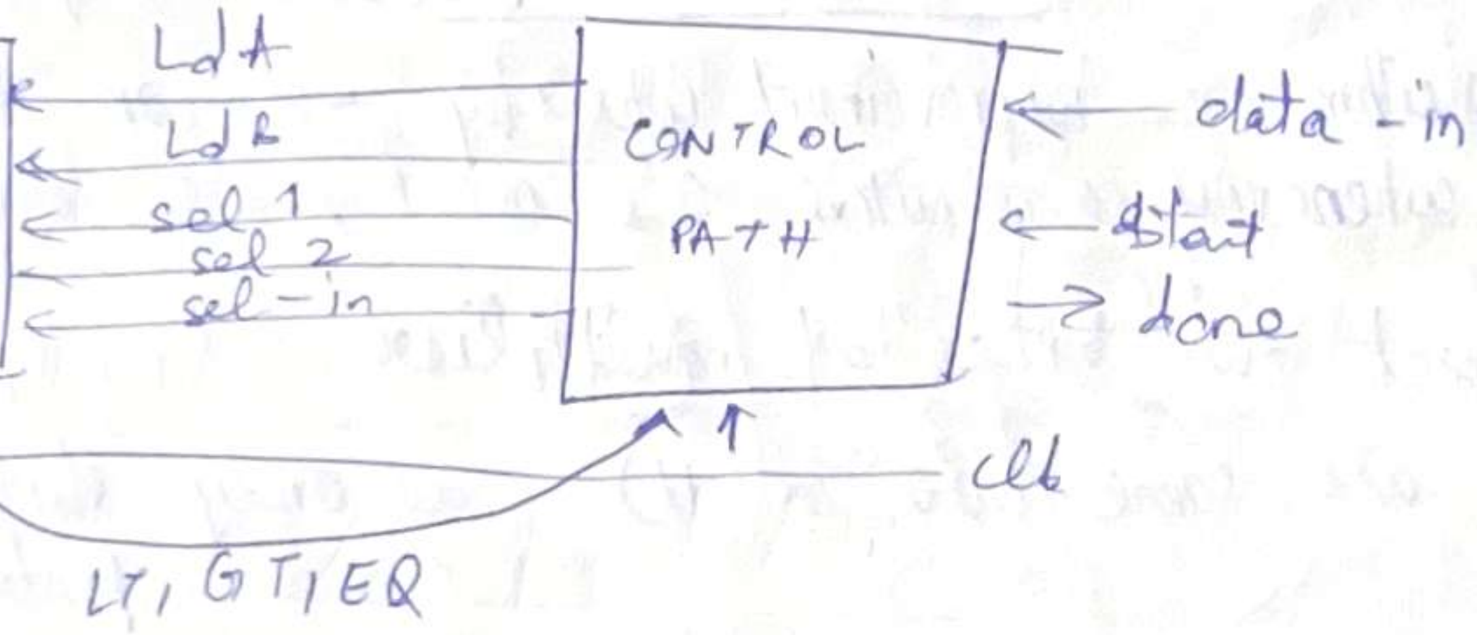
B = 13

A = B

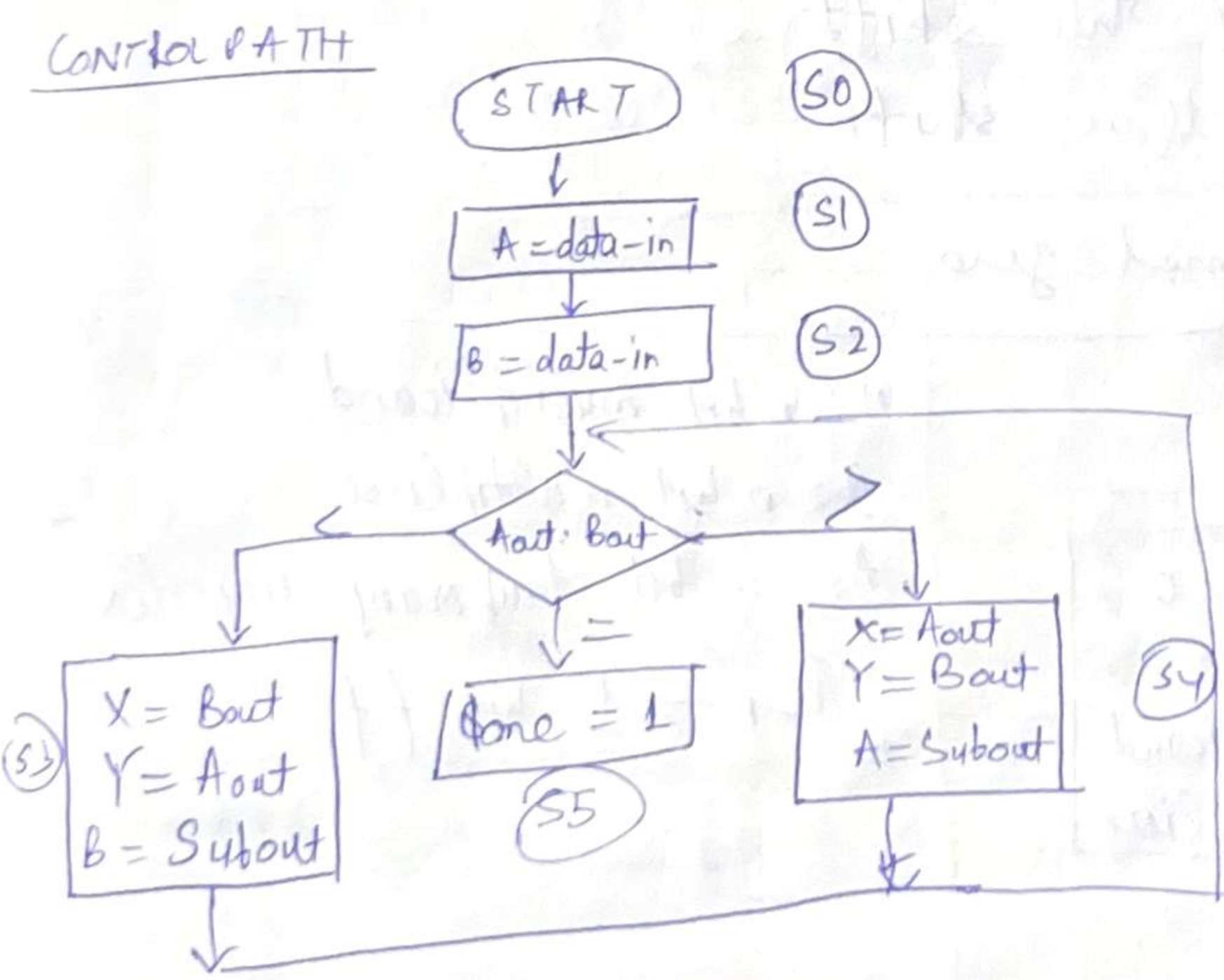
A = 13

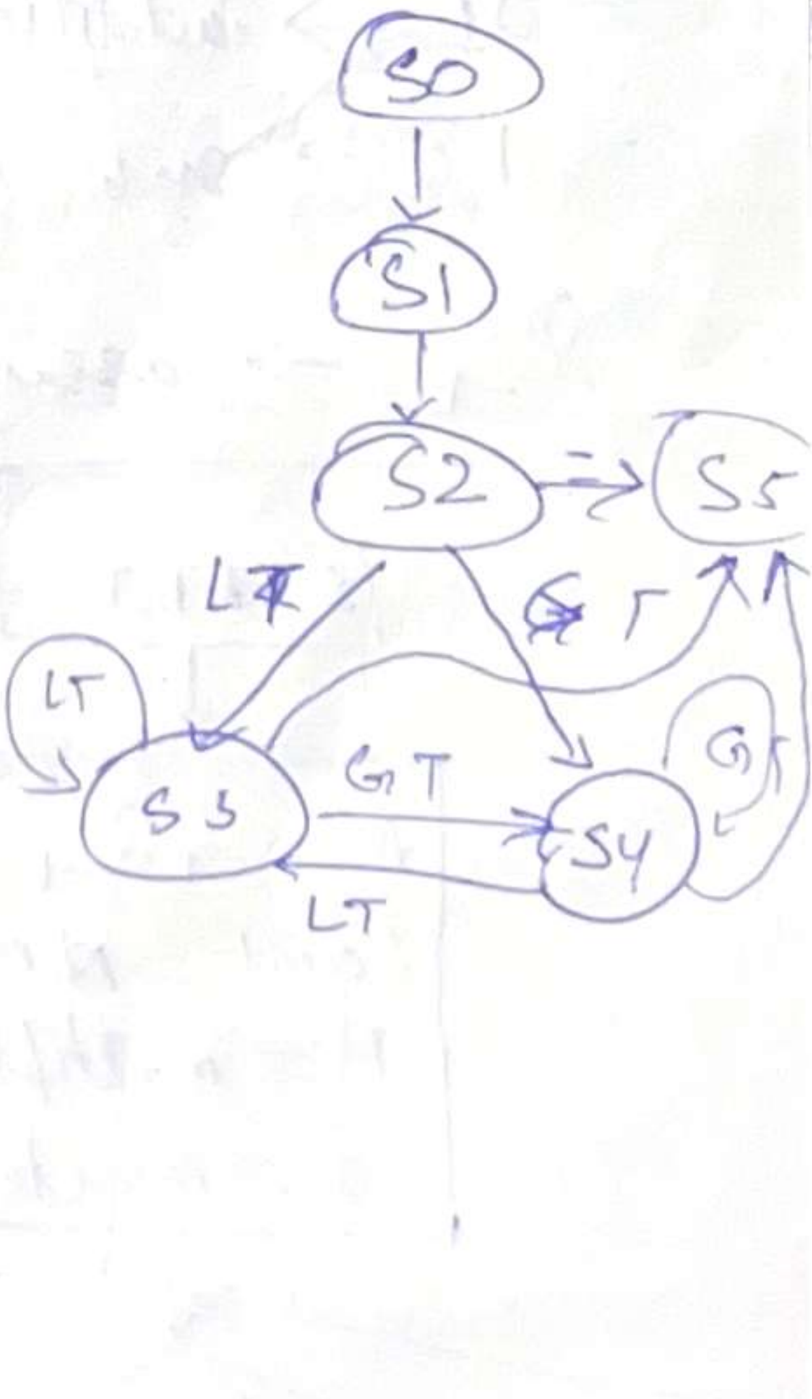
- identify functional blocks required in data path, and corresponding control signals
- Design FSM to implement GCD computation algorithm using data path





CONTROL PATH





Modelling conted path using alternative approach

we define next_state → always @(posedge) → ~~always~~ next state <= next_st.;

→ always @(state)

case (state)

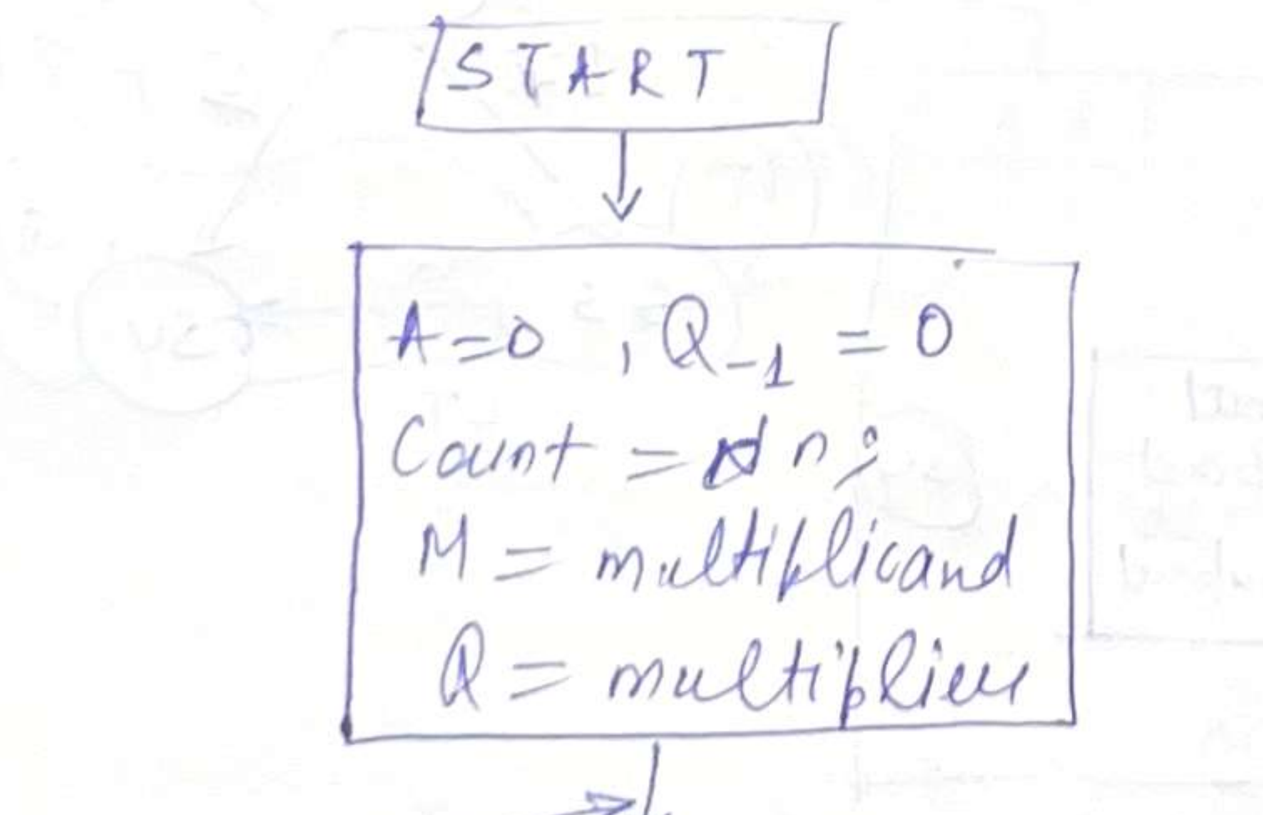
→ in This we will ~~assig~~ blocking assign next state

Lec 27 Part 3 Booth's multiplication Algorithm

Booth's algorithm is improvement whereby we can avoid the additions whenever consecutive 0's or 1's are detected in multiplier

- We inspect two bits of multiplier (Q_i, Q_{i-1}) at time
  if bits are same (00 or 11), we only shift the partial product.
  - 01 → addition then shift
  - 10 → sub then shift

Q_{-1} → assumed zero

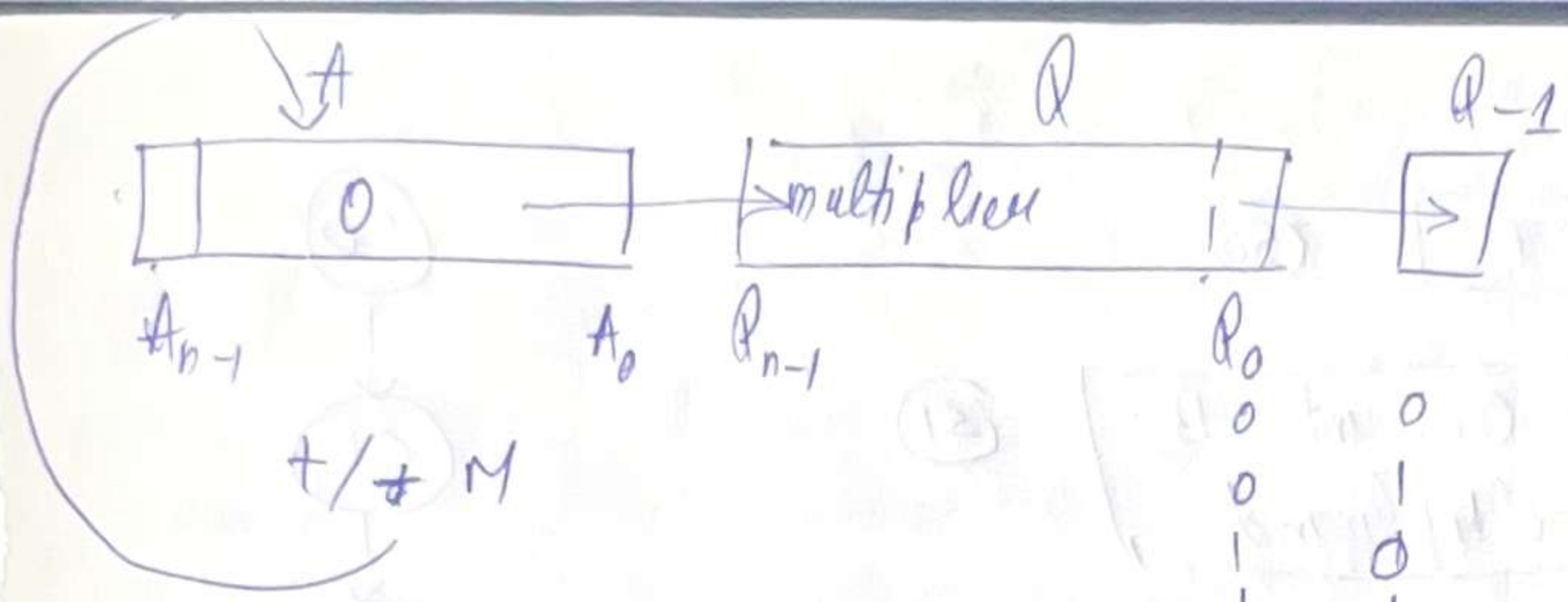


M: n bit multiplicand

Q: n bit multiplier

A: n bit temporary register

Q_{-1} → 1 bit ff.



right shift

$A \rightarrow Q \rightarrow Q_{-1}$

remember signed bit will be shifted back

Example $(-10) \times (13)$

$M : (10110)_2$

$-M : (01010)_2$

$Q : (01101)_2$

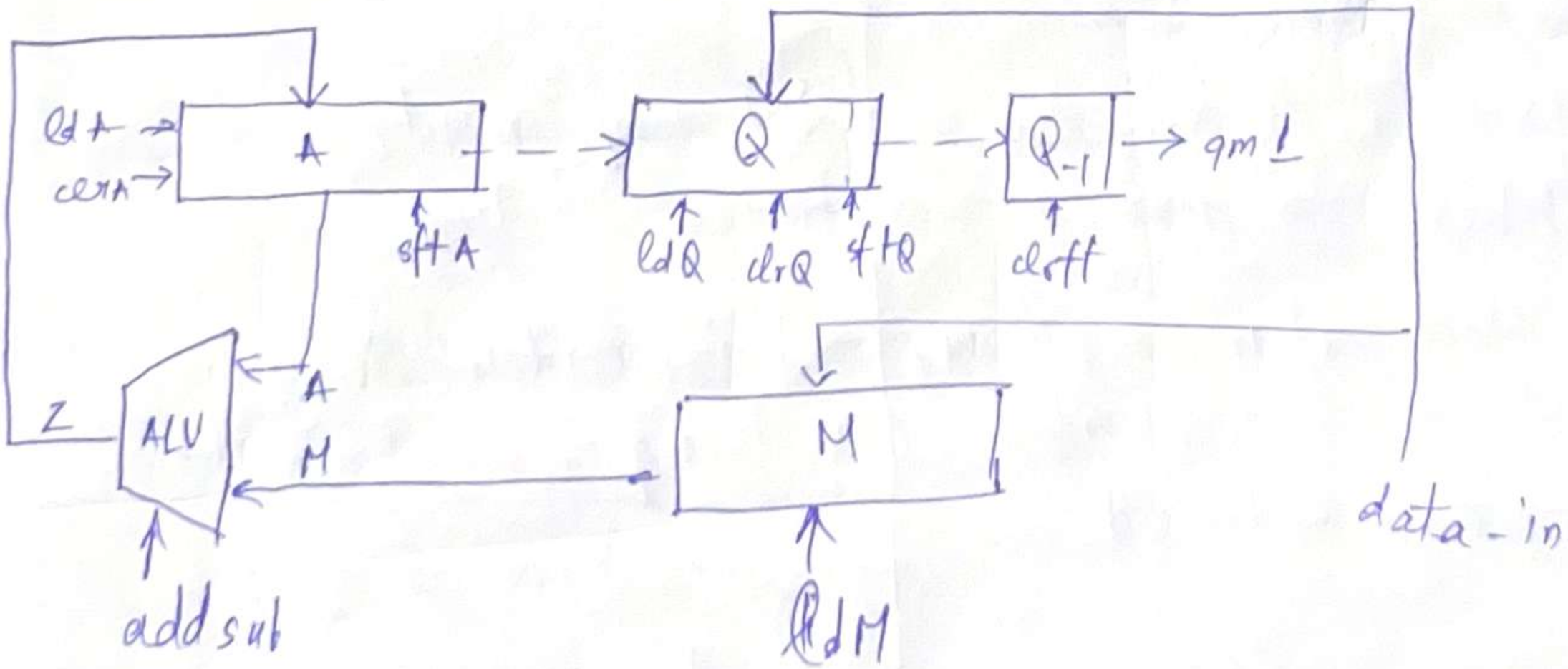
product: -130

$= (1101111110)_2$

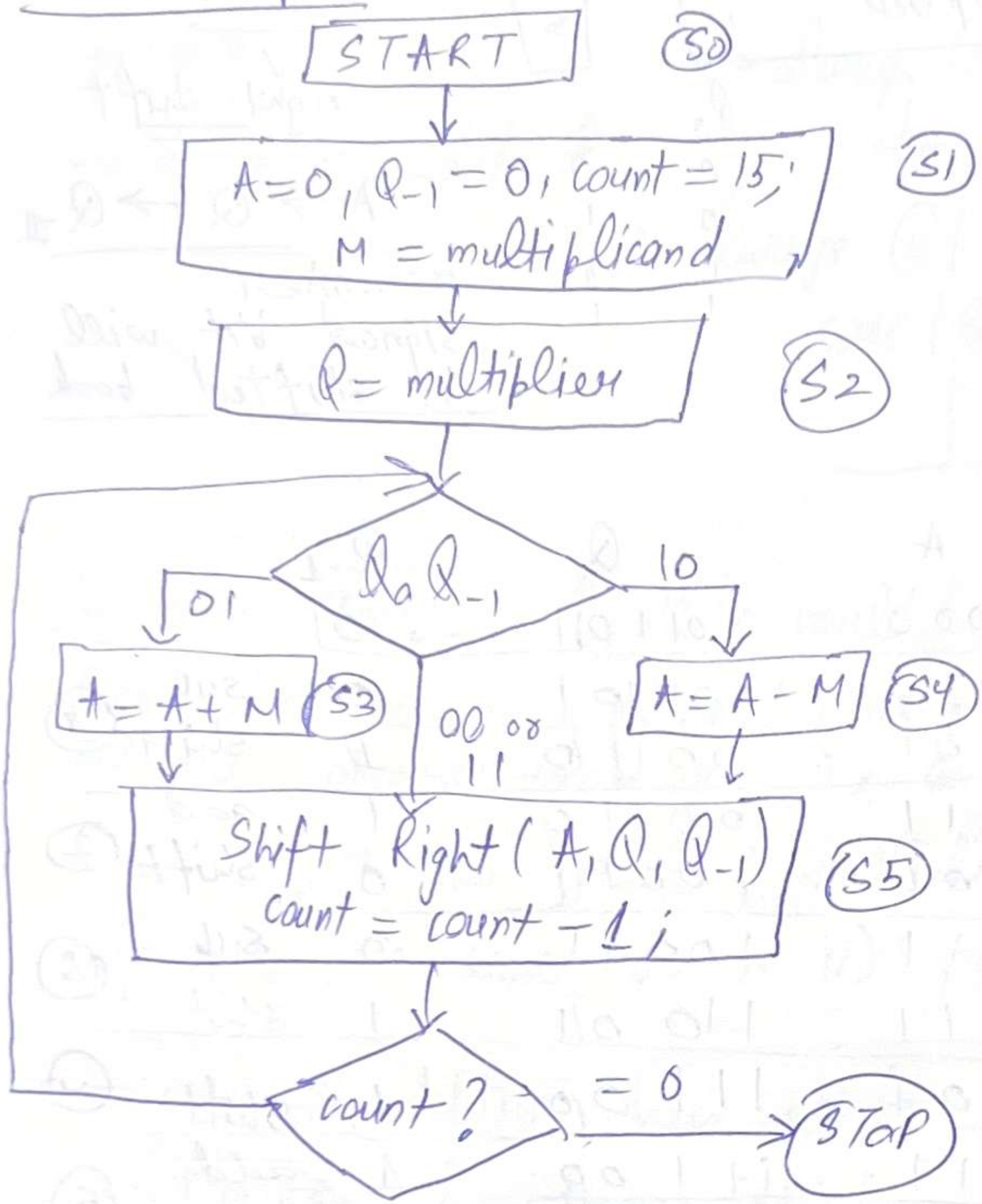
	A	Q	Q_{-1}		
	00000	01101	0		
	01010	01101	0	sub	(1)
	00101	00110	1	shift	
	11011	00110	1	add	(2)
	11101	10011	0	shift	
	00111	10011	0	sub	(3)
	00011	11001	1	shift	
	00001	11100	1	shift	(4)
	10111	11100	1	add	
ANS =	11011	11110	0	shift	(5)

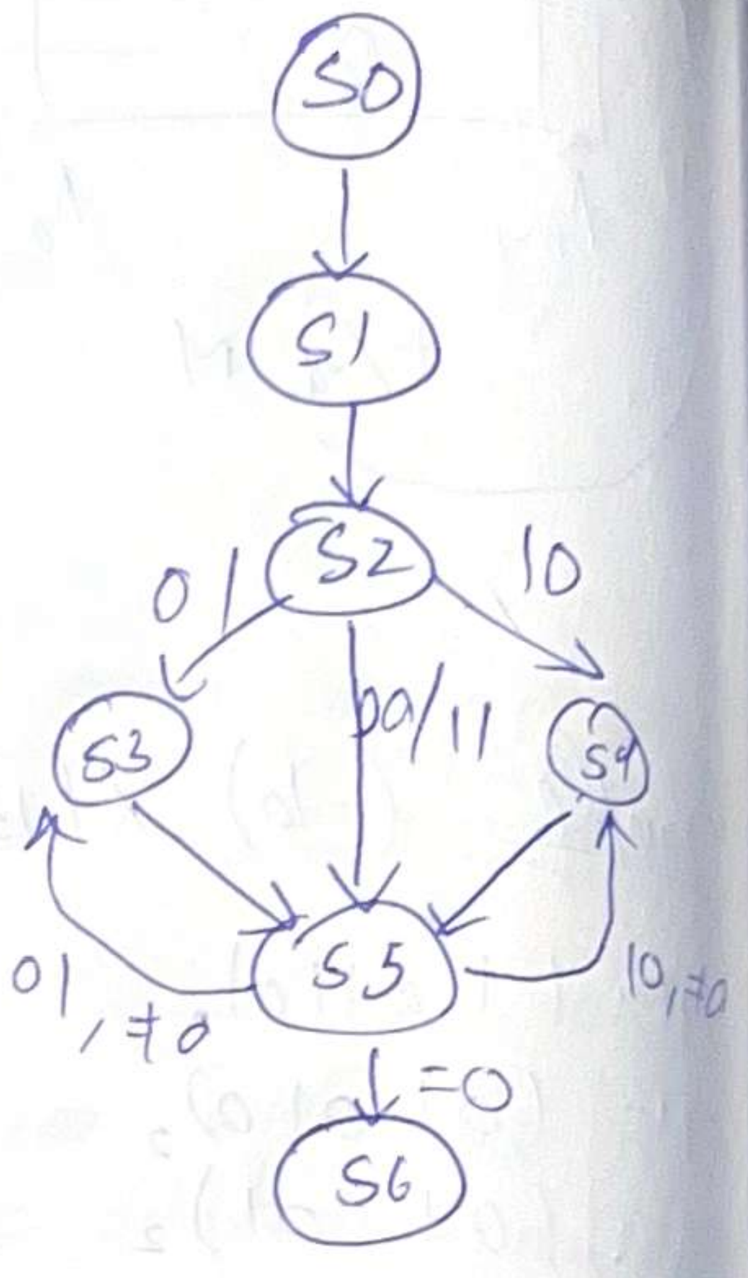
COUNT = 0

DATA path



CONTROL path





Lec 28 Synthesizable verilog → Language subset that can be synthesized:

Rules for combinational

- Avoid techno # delay
- Avoid feedback connections
- Give ~~all~~ outputs for all cases (if not, infers a latch)
  (if..else, "case" constructs.

Styles for synthesizable Combi. logic

- Netlist of verilog built-in primitives like gate instances (AND, OR, NAND)
- Combinational UDP (not ~~much~~ supported by all)
- Continuous assigns
- Functions
- Behavioral statements
- Tasks without event or delay control

↓

Functions in verilog

```
module fa ( s, cout, a, b, cin);
  input  a, b, cin;
  output s, cout;
  assign   s = sum ( a, b, cin);
  assign   cout = carry ( a, b, cin);
endmodule
```

```
function sum;
  input x, y, z;
  begin
    sum = x ^ y ^ z;
  end
function carry;
  input x, y, z;
  begin
    carry = (x & y) | (y & z)
                | (z | x);
```

* Input arguments appear in same order

Task → can be used anywhere

arguments must be specified in same order as they appear in task declaration. More than one output value can be returned.

```
module fulladden( s, cout, a, b, cin);
 input a, b, cin;
 output reg s, cout;
 always @(a, b, cin)
   FA ( s, cout, a, b, cin);

 task FA;
   output sum, carry;
   input A, B, C;
   begin
     #2 sum = A ^ B ^ C;
       carry = (A & B) | (B & C) | (C | A);
     end
   end task
endmodule
```

Diff b/w function & task.

Function	Task
func can call another func but not another task	task can call other tasks & functions
func executes in 0 simulation time	task may execute in nonzero simulation time
func cannot contain any delay, event (or) timing control ~~Assignment~~ statement.	task can contain delay, event, or timing control statements
returns single val	can pass multiple ~~pass~~ values through "output" and "inout" type arguments
func must have at least one input argument	can have zero or more arguments of type "input", "output", "inout"

Construct to Avoid in Combinal

- ~~fork~~ - "join", "wait", "disable"
- Delay loops
- Data dependent loops
- feedback loop
- sequential user defined primitives

Summary: Synthesisable Verilog construct

- "module .. endmodule"
  - Instantiation of synthesisable module
  - "always": assign
  - Built-in gate primitives
  - Bits & part select of vectors.
  - UDPs for Combinational only
- "parameter"
- "func" and "task"
- "for"
- Block + Nonblocking: "if-else", "case", "casex", "casez"

Lecture 30 Modelling Memory (Study Memory part also)

memory typically included by instantiating a pre-designed module from design library.

Modelled using d 2-dimensional arrays.

- Array of register variables (behavioral model)
- Used for synthesis of small size memories
- or for synthesis

Example

```
module memory_(--)
    reg [7:0] mem [0:1023];
endmodule
```

Each memory word is of type [7:0] bits
Access memory words as mem[0], mem[1], ... mem[1023].

How to initialise memory?

- By reading memory data patterns from a specified disk file
  - used for simulation & test benches
- Two verilog functions can be used:

$readmemb (filename, memname, startaddr, stopaddr)
↳ read in binary form

$readmemh ↳ read in hexadecimal form

omission of These 2 leads to reading the entire memory.

Single-port RAM with synchronous Read / write

```
module ram_1 (addr, data, clk, rd, wr, cs);
 input [9:0] addr; input clk, rd, wr, cs;
 inout [7:0] data;
 reg [7:0] mem [1023:0]; reg [7:0] d_out;
 assign data = (cs & rd) ? d_out : 8'bz;
 always @ (posedge clk)
    if (cs && wr && !rd) mem[addr] = data;
 always @ (posedge clk)
    if (cs && rd && !wr) d_out = mem[addr];
endmodule
```

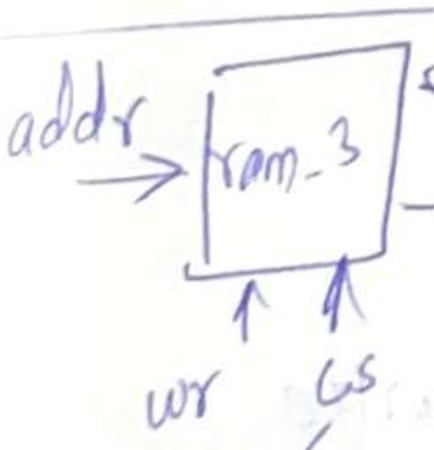
ROM Example

case statement

Important pt to Note (Regarding "inout")

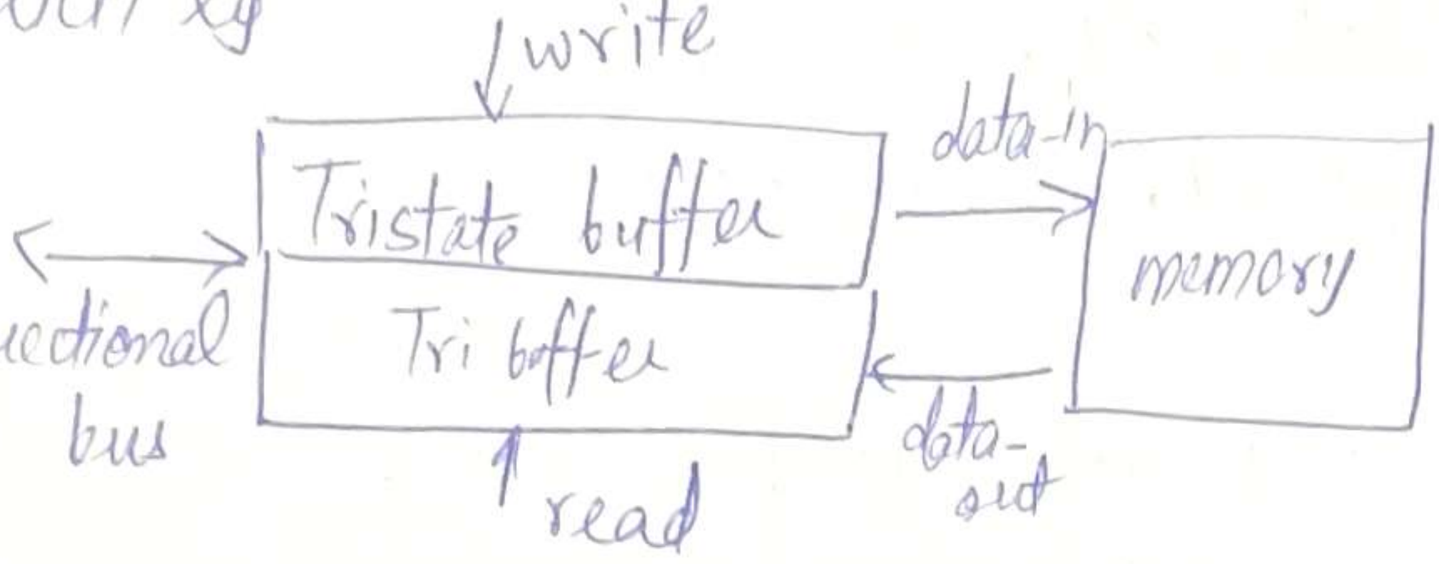
① Some simulation or synthesis tools give inconsistent behaviour when using the "inout" data type. (Should be avoided)

② Better way to design → keep data i/p and data o/p bus lines separate.




```
module rom_3 (data_out, data_in, addr, wr, cs);
input [data_size] data_out;
parameter addr_size = 10, word_size = 8,
          memory_size = 1024;
input [addr_size - 1 : 0] addr;
input [word_size - 1 : 0] data_in;
input wr, cs;
output [word_size - 1 : 0] data_out;
reg [word_size - 1 : 0] mem [memory_size - 1 : 0];
assign data_out = mem [addr];
always @ ( wr or cs )
   if ( wr ) mem[addr] = data_in;
endmodule
```

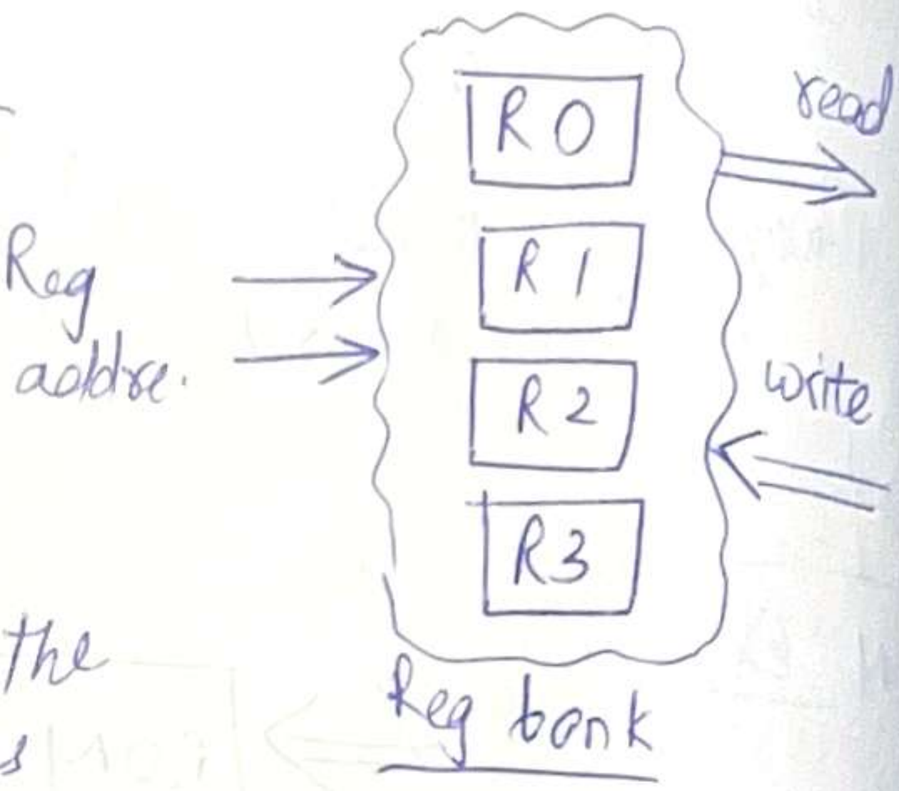
°For bidirectional data bus, tristate buffers can be included explicitly



Lec 31

Modelling Register Banks

A register bank or register file is a group of registers, any of which can be randomly accessed

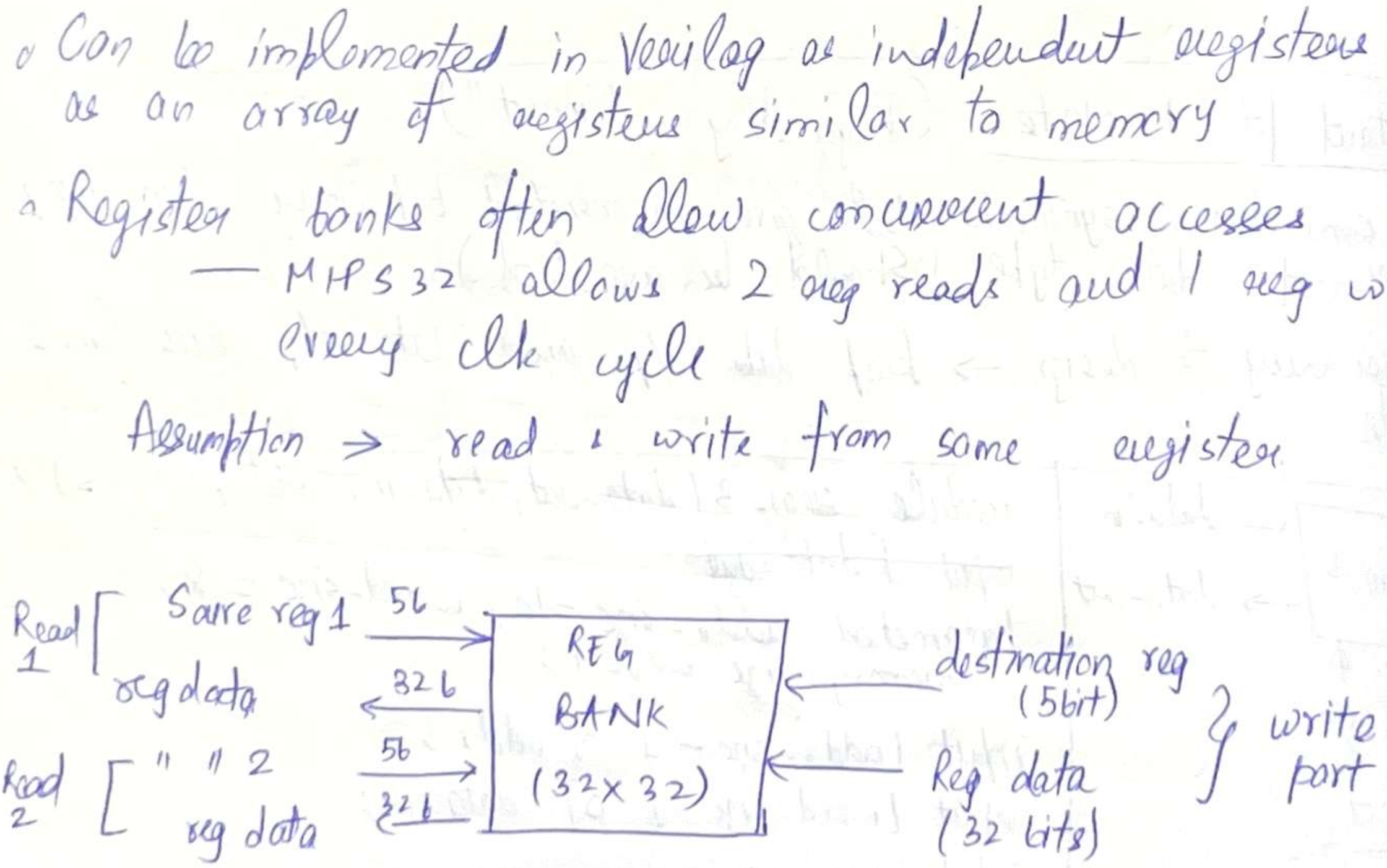


- Commonly used in computers to store the user-~~accessed~~ accessible registers
- For example, In the MIPS 32 processor, there are 32 32bit registers, referred to as R0, R1, R2 - - R31.

° Can be implemented in Verilog as independent registers, or as an array of registers similar to memory

° Register banks often allow concurrent accesses
— MIPS 32 allows 2 reg reads and 1 reg write every clk cycle

Assumption → read, write from same register



4 32x32 reg file

```
module regbank ( rdData1, rdData2, wrData, sr1, sr2, dr,
                 write, clk );
  input clk, write;
  input [4:0] sr1, sr2, dr ; // sources, destination registers
  input [31:0] wrData;
  output [31:0] rdData1, rdData2;

  reg [31:0] regfile [0:31];

  assign rdData1 = regfile [sr1];
  assign rdData2 = regfile [sr2];

  always @(posedge clk)
    if (write) regfile [dr] <= wrData;
endmodule
```

(refer vids for testbenches)

Wk 32 Basic Pipelining Concepts

Mechanism for overlapped execution of several i/p sets by ~~partion~~ partitioning some computation into a set of k-sub-computations (or stages)

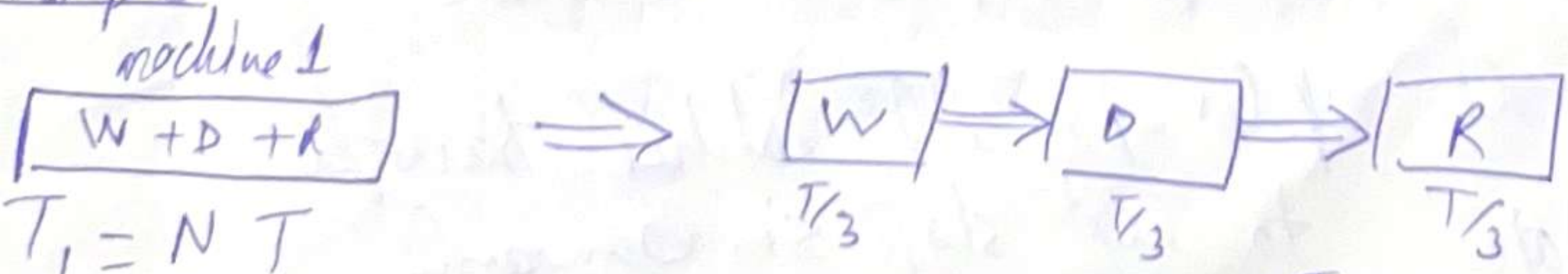
- Very minimal inc in cost of implementation
- Very significant speedup (ideally, k).

* Where is pipelining used in computer system?

- Instruction execution :
- Arithmetic computation
- Memory access → made to consecutive locations

Example

machine 1



$T_1 = N T$

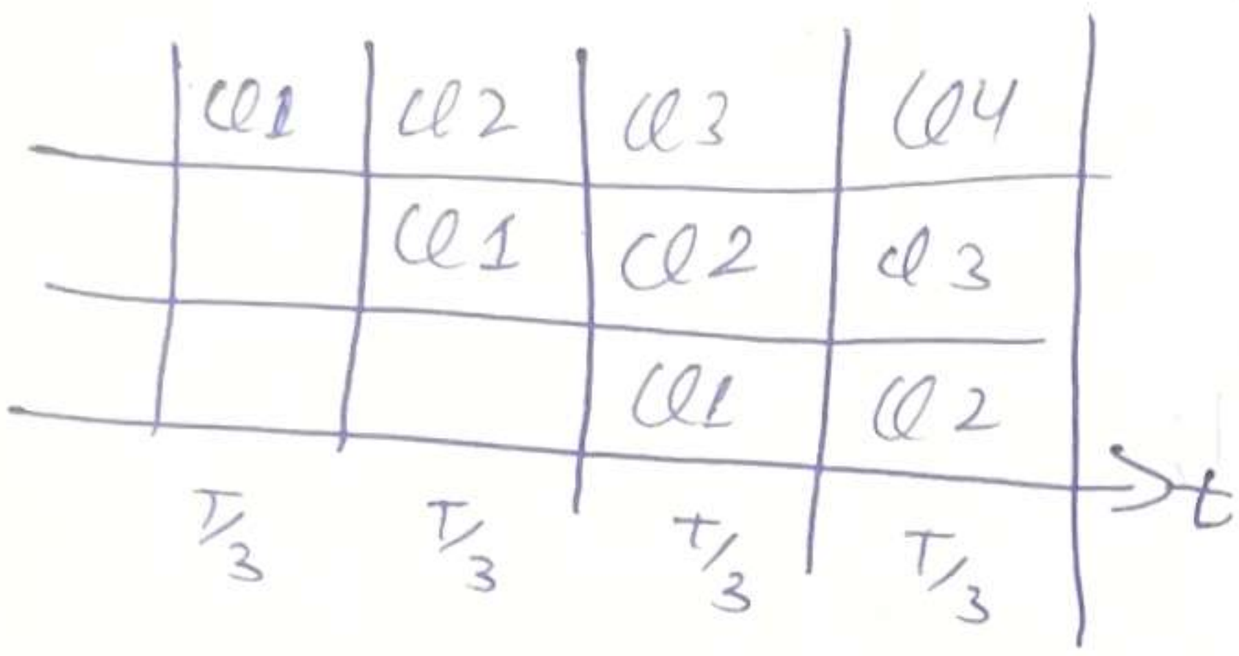
↓

no. of items

$T_3 = (2+N) T/3$

while one machine is running, the other behind is free for more items.

How does pipeline work?



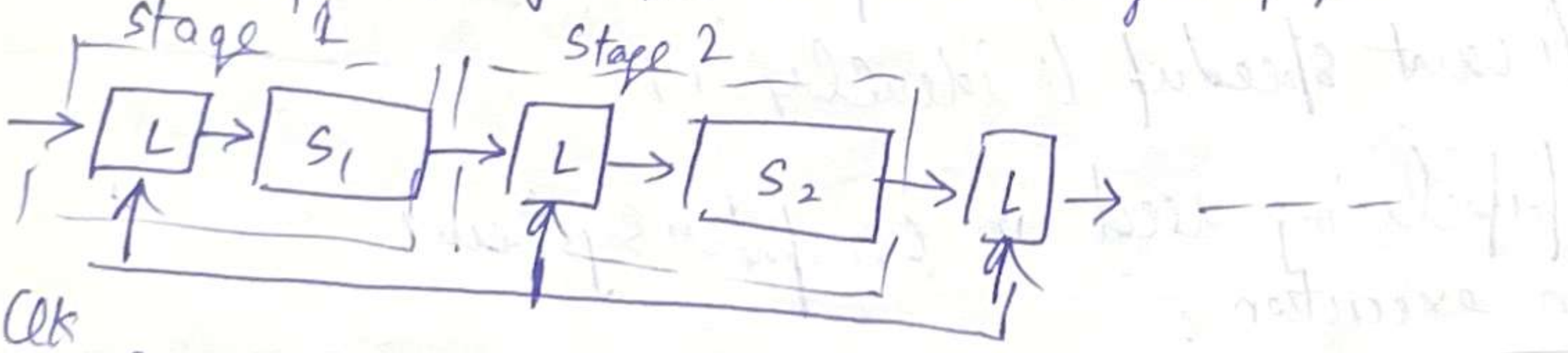
Processor pipeline

To attain k times speed up for computation, split computation into k stages.

- Need for buffering:-

In hardware pipeline, we need a latch b/w successive stages to hold the intermediate results temporarily.

Model of a synchronous k-stage pipeline.



- latches made with master-slave flip flops, and serve the purpose of isolating i/p's from o/p's pipeline stages typically combinational

When clk apply → all latches transfer data to next stage simultaneously.

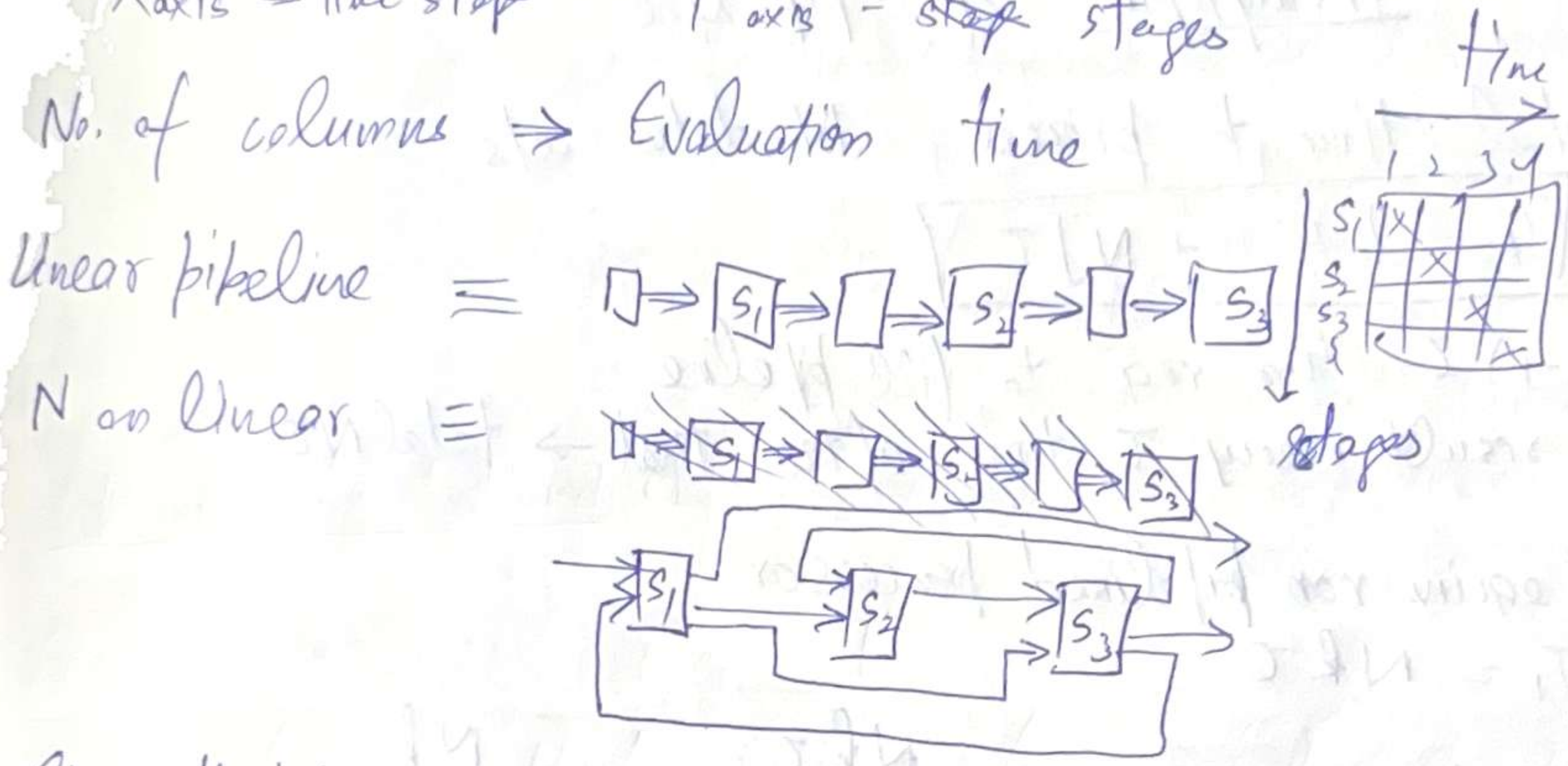
Reservation Table

data structure that represents the utilization pattern of successive stages in a synchronous pipeline.

- Basically a space-time diagram of pipeline that shows precedence relationships among pipeline stages

X axis → time steps Y axis - ~~stage~~ stages

No. of columns ⇒ Evaluation time



Linear pipeline ≡

Non Linear ≡

Characteristics

① Multiple X's in row : repeated use of same stage in different cycles.

② Contiguous X's in row : extended use of a stage over one time ~~period~~ cycle

③ Multiple X in column: multiple stages used in parallel during a clock cycle

Speedup & Efficiency

Some notations : τ :: clock period of pipeline

t_i : time delay of circuitry in stage S_i

d_i : delay of latch.

max stage delay: $\tau_m = \max\{t_i\}$

Thus $\tau = \tau_m + d_L$

Pipeline freq $f = 1/\tau$

If one result is expected to come out of pipeline every clock cycle; f will represent the max <u>throughput</u> of pipeline.

a Total time to process N data sets

$$\boxed{T_k = [(k-1) + N]\tau}$$

$(k-1)\tau$ time req to fill pipeline

1 result every τ time after that → total $N\tau$

For equiv non pipelined processor

$$T_1 = Nk\tau$$

Speed up $S_k = \frac{T_1}{T_k} = \frac{Nk\tau}{k\tau + (N-1)\tau} = \frac{Nk}{k + (N-1)}$

As $N \to \infty$, $S_k \to k$

<u>Pipeline Efficiency</u> $E_k = \frac{S_k}{k} = \frac{N}{k + (N-1)}$

closeness of performance to ideal val

Pipeline throughput: No. of operations completed per unit time.

$$\boxed{H_k = \frac{N}{T_k} = \frac{N}{[k + (N-1)]\tau}}$$

Clock Skew / Jitter / Setup Time

- The minimum clock period of pipeline must ~~be~~ satisfy the inequality:

$$\tau \geq t_{skew+jitter} + t_{logic+setup}$$

Skew: max delay difference b/w arrival of clock signals at the stage latches

Jitter: max delay diff b/w ~~a~~ arrival of clock signal at same latch. (Noise in environment)

Logic delay: max delay of slowest delay ~~of~~ in pipeline

Setup time: minimum time a signal needs to be stable at input of latch before it can be captured

PIPELINE MODELLING

Example 4 N bit unsigned integers A, B, C, D as i/p
N bit unsigned integer F as ~~output~~ o/p

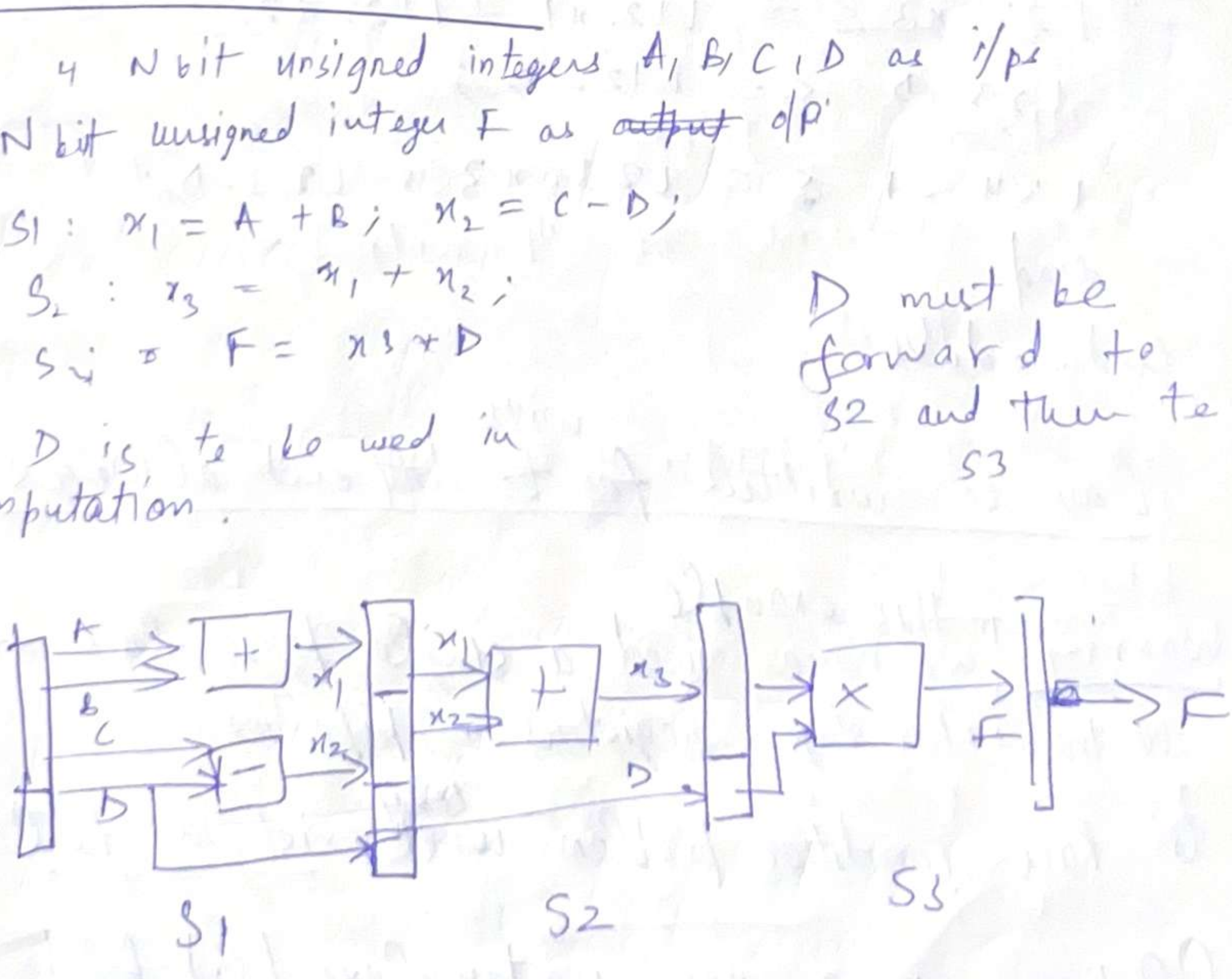
S_1 : $x_1 = A + B$; $x_2 = C - D$;

S_2 : $x_3 = x_1 + x_2$;

S_3 : $F = x_3 * D$

D must be forward to S2 and then to S3

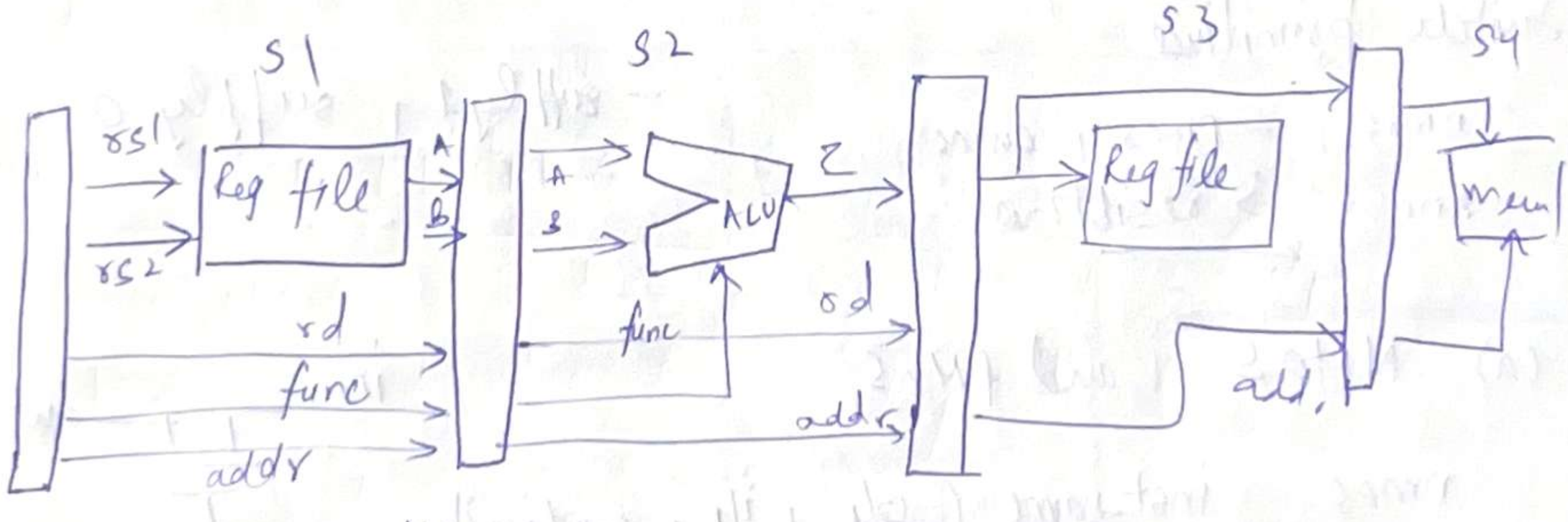
Same D is to be used in computation.



Lecture 34

Consider a pipeline that carries out the following stagewise operations:

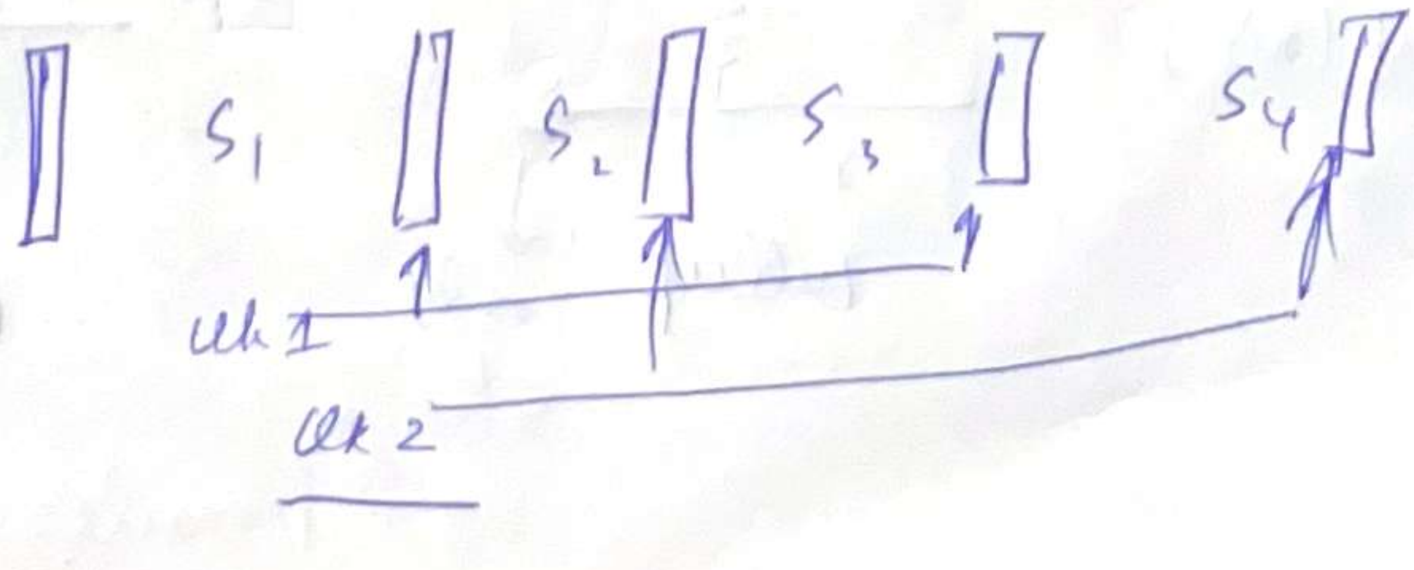
- Inputs : 3 reg addresses (rs1, rs2, rd), ALU func. (func) memory addr. (addr)
- Stage 1 : Read two 16 bit num from regs specified by "rs1" and "rs2" and store them in A and B
- Stage 2 : Perform ALU operation on A and B specified by "func" and store in Z.
- Stage 3 : write val of Z in reg specified by "rd".
- Stage 4 : write val of Z in memory location "addr".



Clocking Issue in Pipeline

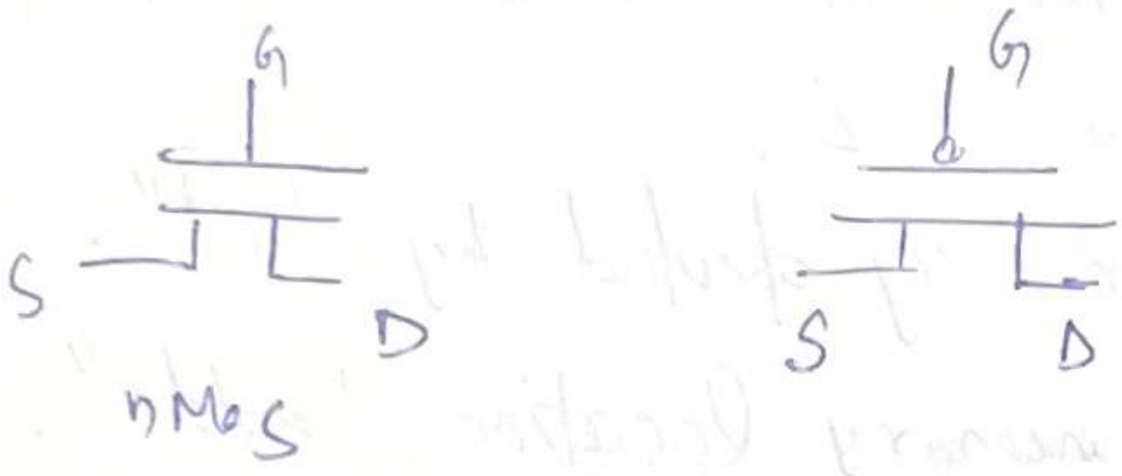
- Important that consecutive stages be applied suitable clocks for correct operation

Two options : ① master clock
② non overlapping 2 ph clk for consecutive pipeline stages



SWITCH LEVEL MODELLING (Pg 35)

- Comprises of netlist of MOS transistors
- not very common
- Verilog provides ability to model digital circuits at MOS level
  - Transistor func. as switches → either conduct (ON) or open (OFF)
- Types of switches : ideal or resistive

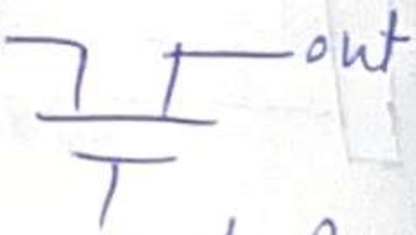


Switch primitives

- nmos, - pmos, cmos
  rnmos → resistive

- supply1, supply0

(a) NMOS and PMOS



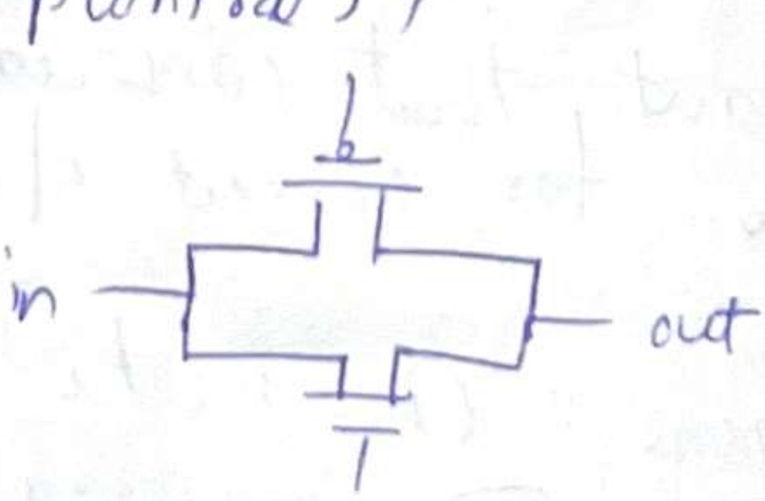
nmos inst_name (o/p, i/p, control);

optional

cmos in (o/p, i/p, ncontrol, pcontrol);

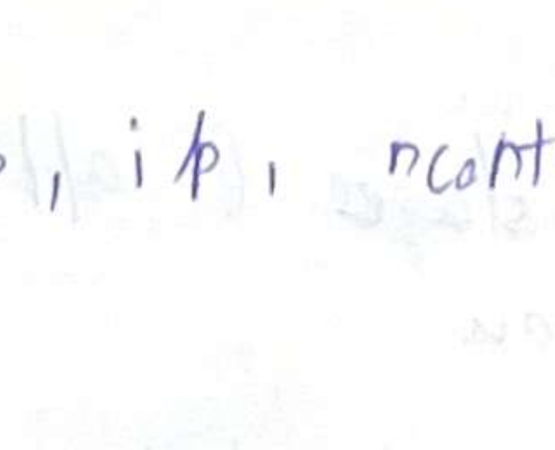
↓

better switch



pullup, pulldown

pullup (a);



only stronger signal prevails.

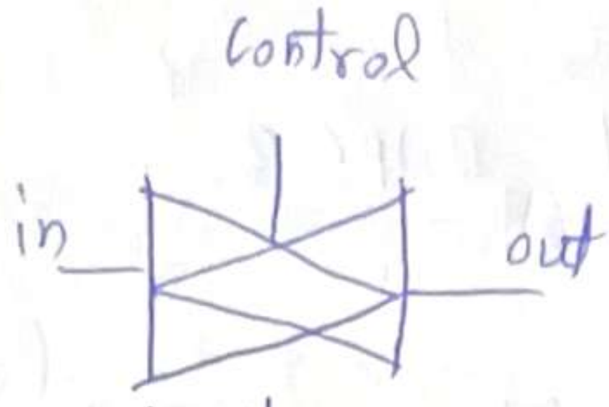
(c) Bidirectional Switches

conduct in both directions

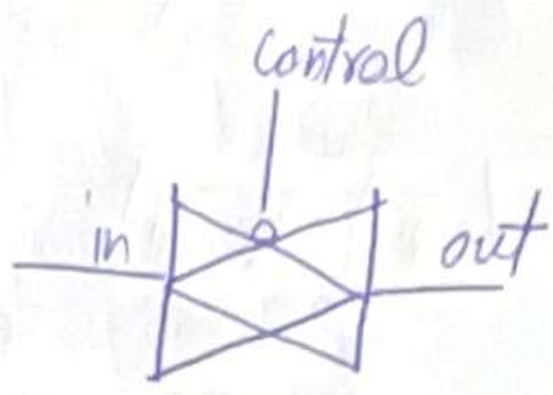
tran

tran(rtran)

↙ always conducting

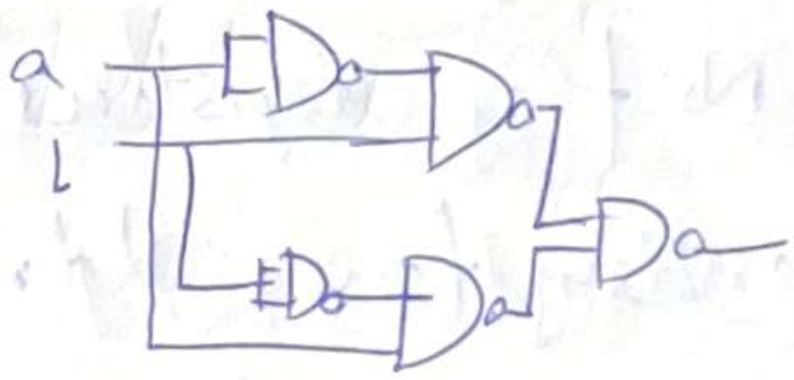


tranif1(r–)



tranif0(r–)

Part 2 Lecture 36



Data values & Signal Strengths

S2 < S1

~~connection~~

~~2 signals drive~~