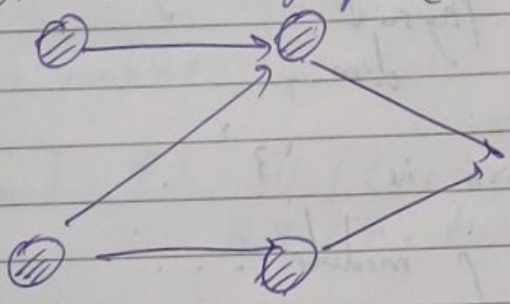
NPTEL Verilog

- feature size → smallest feature of the transistor that you can fabricate.

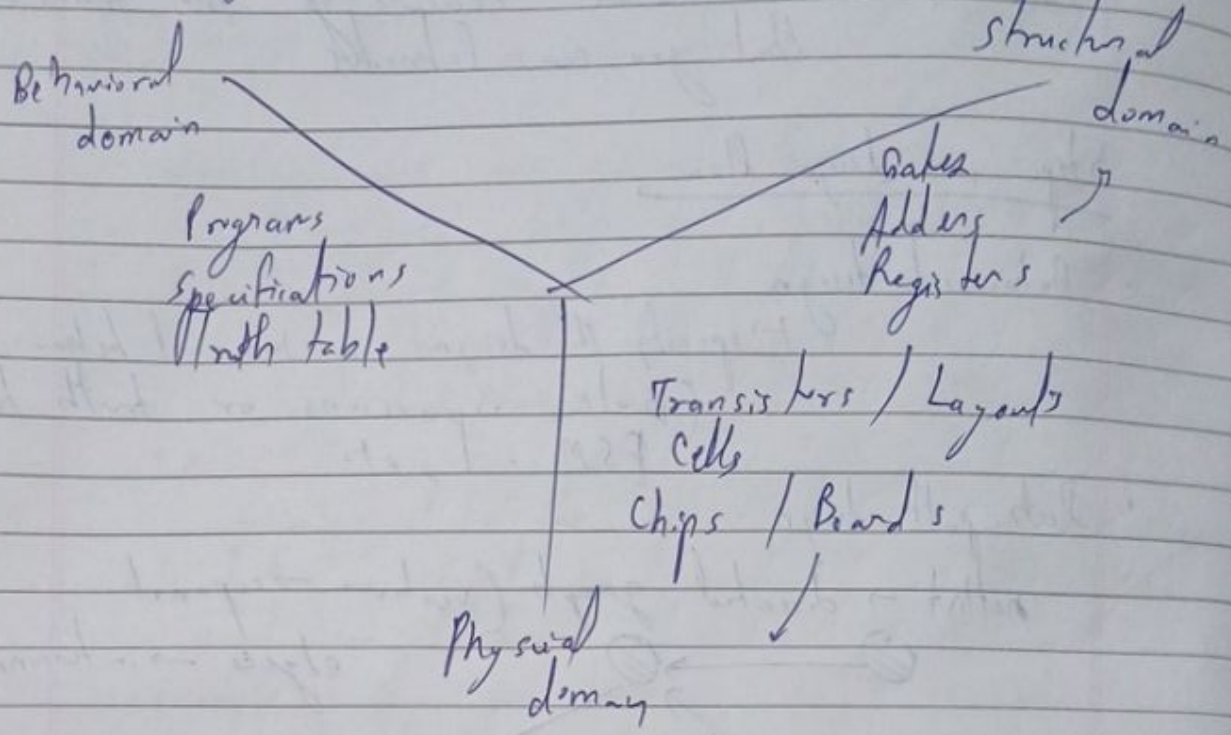
Steps in design flow

- Behavioral design
  ↳ specify the design in terms of behav...
  (Boolean expressions or truth tables
  FSM ..., etc
- Data path design
  netlist → directed graph (vertices → components
  edges → interconnections)

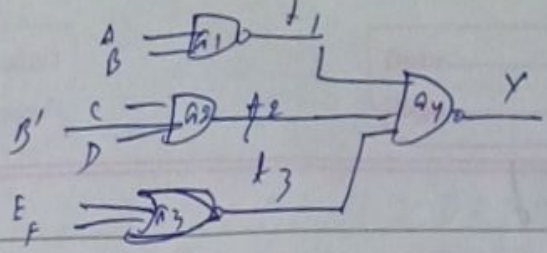


- Logic design
  ↳ generate a netlist of ~~gates~~/flip flops or standard cell.
  - minimizing gates, no. of gate levels & signal transmission activities
- Physical design & Manufacturing.
  - fabrication
  - Alternatively, FPGA (but less speed)

Y-diagram



- don't care notion is '?'.
- Module 1 { module ... ; end module
- module 2
- module 3
- ASIC → Application specific Integrated circuit
  FPGA → Field programmable gate array.



Example

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

delay   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
```

We can combine the same type of gate together

```
        nand #1 G1 (t1, A, B);
               G4 (Y, t1, t2, t3) ;
```

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

    initial
        begin
if any of ——— $monitor ($time, " A=%b B=%b,
changes, it will      C=%b, D=%b, E=%b, Y=%b",
is show                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;
        #5 F=1;
        #5 $finish;
```

binary

```
    end
endmodule
```

- Simulation results:

```
0   A=x, B=x, C=x, D=x, E=x, F=x, Y=x
5   A=1, B=0, C=0, D=1, E=0, F=0, Y=x
8   A=1, B=0, C=0, D=1, E=0, F=0, Y=1
10  A=0, B=0, C=1, D=1, E=0, F=0, Y=1
13  A=0, B=0, C=1, D=1, E=0, F=0, Y=0
15  A=1, B=0, C=0, D=1, E=0, F=0, Y=0
18  A=1, B=0, C=0, D=1, E=0, F=0, Y=1
20  A=1, B=0, C=0, D=1, E=0, F=1, Y=1
```

Command in iVerilog:
a) iverilog -o mysim example.v example_tb.v
b) vvp mysim

: While modifications
~~reg A, B, E~~

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

  initial
    begin
      $dumpfile ("example.vcd");
      $dumpvars (0, testbench);
      $monitor ($time, "A=%b, B=%b, C=%b,
                D=%b, E=%b, F=%b, Y=%b",
               A, B, C, D, E, F, Y);
```

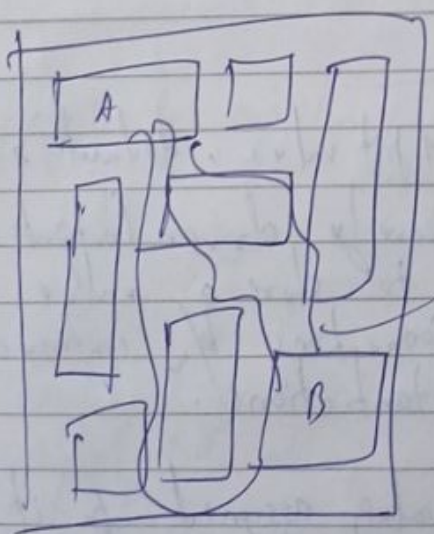
```
#5 A=1; B=0; C=0; D=1; E=0; F=0;
#5 A=0; B=0; C=1; D=1; E=0; F=0;
#5 A=1; C=0;
#5 F=1;
#5 $finish
end
endmodule
```

To display the waveforms, run the command

```
gtkwave example.vcd
```

Stick diagram

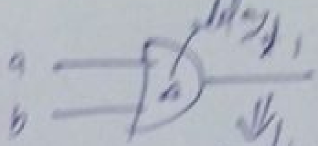
- Blue lines are metal connections
  red lines are poly silicon connections
  green & brown are the diffusions.
  Black crosses are the interconnections.



Placement

Routing

- In CMOS tech, it is easier to design NAND, NOR & NOT gates.



continuously driven (not the level triggered)

Point to note:

- The "assign" statement represents continuous assignment whereby the variable on the LHS gets updated whenever the expression on the RHS changes

assign variable = expression;

- The LHS must be a "net" type variable, typically a "wire".
- The RHS can contain both "register" and "net" type variables.
- A verilog module can contain any number of assign statements; they are typically placed in the beginning after the port declarations.
- The "assign" statement models behavioral design style and is typically used to model combinational ckts.

Data types in Verilog

- a) Net (by default are 1 bit value, default z state) -
  - must be continuously driven (output of block)
  - cannot be used to store a value
  - used to model connections b/w continuous assignments and instantiations.
- b) Register
  - Retains the last value assigned to it.
  - Often used to represent storage elements, but sometimes it can translate to combinational ckts also.

Net

- Various "Net" data types → wire, wor, wand, tri, supply0, supply1, etc..
- "wire" and "tri" are equivalent
- "wor" "wand" → insert an OR & and AND gate, respectively at connection.
- "supply0" and "supply1" model power supply connections.

wor f;

assign f = A & B;
assign f = C | D;

↳ So, here function is realised

$$f = (A \& B) \mid (C \mid D)$$

in case of wand '&'

- supply0 & supply1 have the greatest signal strength.

```
module using_supply_wire (A, B, C, f);
input A, B, C;
output f;
supply0 gnd;
supply1 vdd;

nand A1 (f1, vdd, A, B);
xor  f2 (f2, C, gnd);
and  f2 (f2, C, gnd);
and  f3 (f, f1, f2);
endmodule
```

Value held	represents
0	logic 0 state
1	logic 1 state
x	unknown logic state
z	high impedance state

Initialization:
- all unconnected nets are set to "z".
- all register variables set to "x".

Register

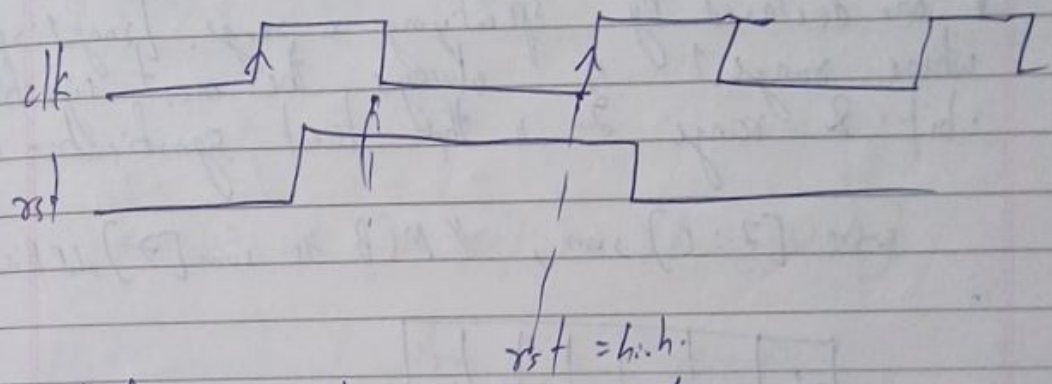
- It can hold a value
- eg: reg : most widely used
  integer : used for loop counting
  real : used to store floating-point numbers
  time : keeps track of simulation time (not used in synthesis)

```
reg x,y; // Single bit register variables
reg [15:0] bus; // A 16-bit bus
```

- "reg" must be used when we model actual sequential hardware elements like counters, shift registers, etc.

Example 32 bit counter with synchronous reset.
- Count value increases at the positive edge of the clock.
- If "rst" is high, the counter is reset at the positive edge of the next clock

```
module simple_counter (clk, rst, count);
  input clk, rst;
  output [31:0] count;
  reg [31:0] count;

  always @(posedge clk)        → synchronous rst
  begin
    if (rst)
      count = 32'b0;
    else
      count = count + 1;
  end
endmodule
```



for asynchronous reset →

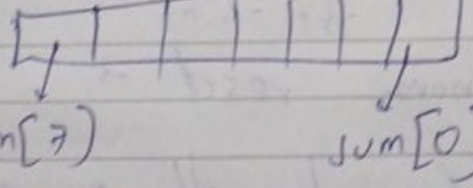
always @(posedge clk or posedge rst)

- "time" data type
- In verilog, simulation is carried out with respect to a logical clock called simulation time.
- The "time" data type can be used to store simulation time.
- The system function "$time" gives the current simulation time.
- Example:

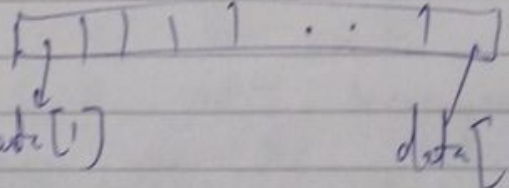
```
time curr_time;
initial
...
curr_time = $time;
```

- Values
- ▮ are declared by specifying a range [range_1 : range_2], where range_1 is always the most significant bit & range_2 is the least significant bit.

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



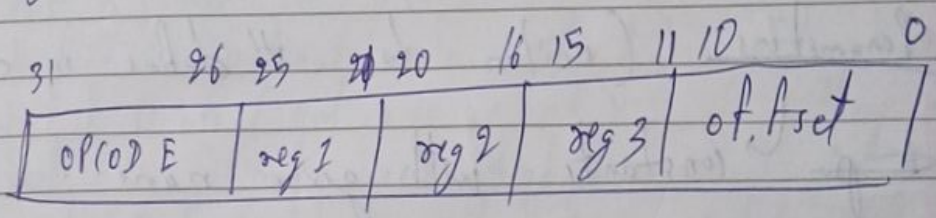
reg [1:10] data



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

eg - A 32 bit instruction register, that contains a 6-bit opcode, three register operands of 5 bits each and 11-bit offset.

```
reg [31:0] IR;
reg [5:0] opcode;
reg [4:0] reg1, reg2, reg3
reg [10:0] offset

opcode = IR[31:26];
reg1 = IR[25:21];
reg2 = IR[20:16];
reg3 = IR[15:11];
offset = IR[10:0];
```




```
sum = IR[25:21] + IR[20:16]
```

Multi-dimensional Arrays & Memories

- Multi-dimensional arrays of any dimension can be declared in Verilog.
- Example:

```
reg [31:0] register_bank [15:0]; // 16 32-bit registers
integer matrix [7:0] [15:0];
```

- Memories can be modeled in Verilog as a 1-D array of registers.

eg

```
reg mem_bit [0:2047];    // 2K 1-bit words
reg [15:0] mem_word [0:1023]; // 1K 16-bit words
```

Specifying constant values

- syntax: <size>'<base><number>

example:

```
4'b0101     // 4 bit binary number 0101
2'b0        // logic 0 (2 bit)
12'hB3C     // 12 bit number in hex
12'h8xF     // 12 bit number 1000 xxxx 1111
25          // signed number, in 32 bit.
```

Parameters (similar to #define in c)

↳ constants with given name.

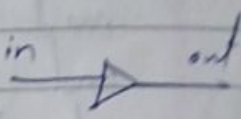
Ex
```
parameter HI = 25, LO = 5;
parameter up = 2'b00, down = 2'b01;
parameter RED = 3'b100, YELLOW = 3'b010;
```

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

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

. Predefined logic gates in verilog

List of Primitive gates

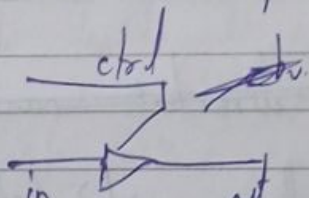
```
and  G (out, in1, in2);
nand G (out, in1, in2);
or   G (out, in1, in2);
nor  G (out, in1, in2);
xor  G (out, in1, in2);
xnor G (out, in1, in2);
not  G (out, in);
buf  G (out, in);
bufif1 G (out, in, ctrl);
bufif0 G (out, in, ctrl);
notif0 G (out, in, ctrl);
notif1 G (out, in, ctrl);
```



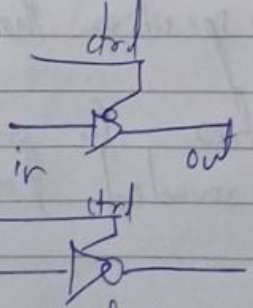
these are gates with tristate controls

. and G (a, b, c, d, e, f);

output — inputs



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

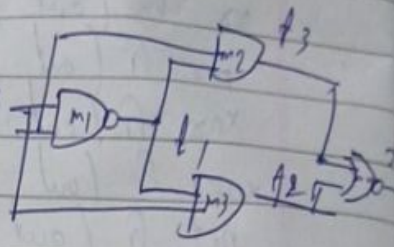


not of 1, z if 0

Ex:

Deriving exclusive-or ~~using nand gates~~:

```
`timescale 10ns / 1ns

module exclusive_or (f, a, b);
  input a, b;
  output f;
  wire t1, t2, t3;
  nand #5 m1 (t1, a, b);
  and #5 m2 (t2, a, t1);
  and #5 m3 (t3, t1, b);
  nor #5 m4 (f, t2, t3);
endmodule.
```



The `timescale Directive

- Often in a single simulation, delay values in one module need to be specified in terms of some time unit, while those in some other module need to be specified in terms of some other time unit.
- The `timescale compiler directive can be used:

~~`timescale compiler directive can be us~~

`timescale <reference_time_unit> / <time_precision>

- The <reference_time_unit> specifies the unit of measurement for time.
- The <time_precision> specifies the precision to which the delays are rounded off during simulation.

- Valid values for specifying time unit and time precision are 1, 10, 100.

ex: `timescale 10ns / 1ns

- If we specify #5 as delay, it will mean 50ns.
- Units are s, ms, us, ps, & fs.

Specifying connectivity during instantiation :-

- Positional Association (same order as ~~defin~~ in definition)
  - eg: nand #1 A1 (Y, A, B);
- Explicit Association

example

```
module example (A, B, C, D, E, F, X);
  wire t1, t2, t3, Y;
  nand #1 G1 (t1, A, B);
  and #2 G2 (t2, C, ~B, D);
  :
  :
end module
```

positional association.

```
module testbench
  reg x1, x2, x3, x4, x5, x6; wire OUT;
  example DUT (.OUT(Y), .x1(A), .x2(B), .x3(C),
               .x4(D), .x5(E), .x6(F));
  initial
    begin
    :
    :
```

explicit association

```
end mdu
```

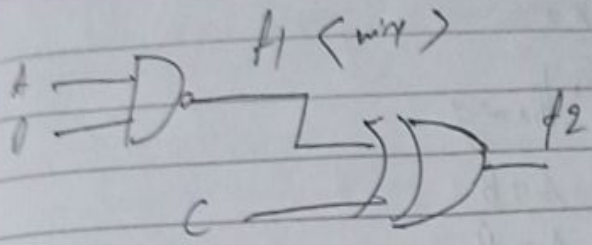
Hardware modeling issues

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

- A variable in Verilog can be either "net" or "register".
  - A "net" data type always map to a "wire" during synthesis.
  - A "register" data type maps either to a "wire" or a "storage cell" depending upon the context under which a value is assigned.

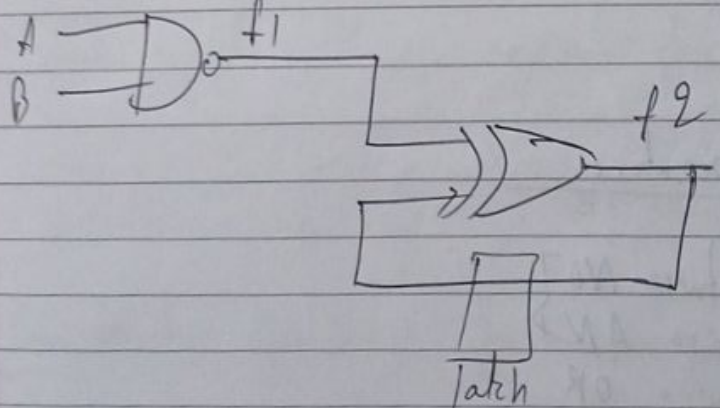
```
module reg_maps_to_wire (A, B, C, f1, f2);
  input A, B, C;
  output f1, f2;
  wire A, B, C;
  reg f1, f2;
  always @ (A or B or C).        → whenever A or B or C changes, always block implemented
  begin
    f1 = ~(A & B);
    f2 = f1 ^ C;
  end
endmodule
```

- The synthesis system will generate a wire for f1.




```
module a_problem-case (A, B, C, f1, f2);
  input A, B, C;
  output f1, f2;
  wire A, B, C;
  reg f1, f2;
  always @ (A or B or C)
  begin
    f2 = f1 ^ f2;
    f1 = ~(A & B);
  end
endmodule
```

The synthesis system will generate a wire for f1, and a storage cell for f2.



Verilog Operators

- Arithmetic operators

unary	binary
+A	A+B
-B	A-B
-(A+B)	A*B

- % → modulus
- ** → exponentiation

- Logical operators

! → logical negation
&& → logical AND
|| → logical OR

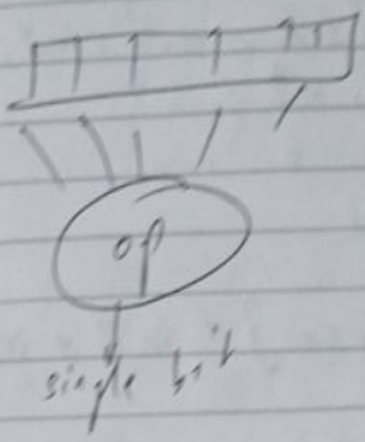
- Relational operators

!= not equal
== equal
>= greater or equal
<=
>
<

Bitwise operators

~ bitwise NOT
& bitwise AND
| bitwise OR
^ bitwise exclusive - OR
~^ bitwise exclusive - NOR

Reduction operators — unary



wire [3:0] x; wire y;

assign y = &x

Shift operators:

>> shift right
<< shift left
>>> arithmetic shift right

ex

wire [15:0] data, target

assign target = data >>> 3;

shift while preserving the sign

Conditional Operator:

cond_expr ? true_expr : false_expr ;

eg

assign a = (b > c) ? b : c ;

Concatenation Operator: {..., ..., ...}

Joins together bits from two or more comma-separated expressions.

// An 8 bit adder description

```
module parallel_adder (sum, cout, in1, in2, cin);
    input [7:0] in1, in2;
    input cin;
    output [7:0] sum;
    output cout;
    assign #90 {cout, sum} = in1 + in2 + cin;
endmodule
```

- Modeling

Behavioral — Starting Point → Structural — Hardware implementation

Example : 16 x 1 mux

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

  assign out = in[sel];
end module
```

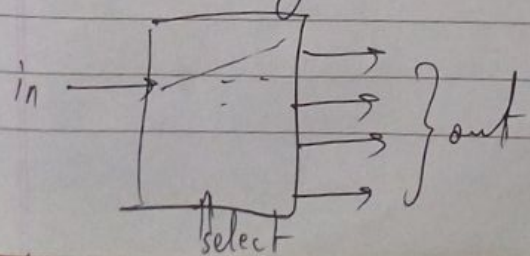
```
module muxtest;
  reg [15:0] A; reg [3:0] S; wire F;

  mux16to1 M (.in(A), .sel(S), .out(F));
```

```
initial
  begin
    $dumpfile("          ")
    $dumpvars(0,          )
    $monitor($time, "A = %h, S = %h, F = %b",
                A, S, F);
    #5 A = 16'h3f0a; S = 4'h0;
    #5 S = 4'h1;
    #5 S = 4'h6;
    #5 S = 4'hc;
    #5 $finish;
  end
endmodule
```

1. Whenever there is conditional statements → a mux is generated.

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

  assign out[select] = in;
endmodule
```

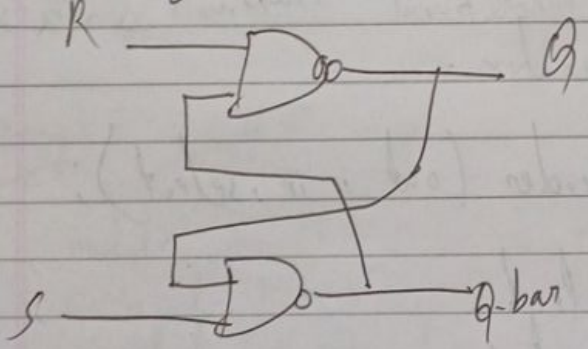
Non const. in expression on LHS generated a decoder



```
module level_sensitive_latch (D, Q, En);
  input D, En;
  output Q;

  assign Q = en ? D : Q;
endmodule
```

En	D	Q_n
0	x	Q_{n-1}
1	0	0
1	1	1

- Modelling SR latch



S	R	Q_n
1	1	Q_{n-1}
0	1	0
1	0	1
0	0	?

```
module SR_latch (Q, Qbar, S, R);
  input S, R;
  output Q, Qbar;

  assign Q = ~(R & Qbar);
  assign Qbar = ~(S & Q);
endmodule
```

```
module latchtest;
  reg S,R; wire Q, Qbar;
  SR_latch LAT(Q, Qbar, S, R);

  initial
    begin
    $monitor($time, "S = %b, R = %b, Q = %b,
                Qbar = %b", S, R, Q, Qbar);
      S = 1'b0; R = 1'b1;
    #5 S = 1'b1; R = 1'b1;
    #5 S = 1'b1; R = 1'b0;
    #5 S = 1'b1; R = 1'b1;
    #5 S = 1'b0; R = 1'b0;
    #5 S = 1'b1; R = 1'b1;
    end
endmodule.
```

- while loop

```
integer mycount;

initial begin
    while (mycount <= 255)
      begin
        $display ("My count: %d", mycount);
        mycount = mycount + 1;
      end
end
```

Behavioral style : Procedural Assignment

- Two kinds of procedural blocks are supported in Verilog :
  - The "initial" block
    - Executed once at the beginning of simulation
    - Used only in test benches; cannot be used in synthesis.
  - The "always" block
    - A continuous loop that never terminates
- The procedural block defines :
  - A region of code containing sequential statements
  - The statements execute in the order they are written.

The "initial" block

- All statements inside an "initial" statement constitutes an "initial block".
  - Grouped inside a "begin ... end" structure for multiple statements.
  - The statements starts at time 0, and execute only once.
  - If there are multiple "initial" blocks, all the blocks will start to execute concurrently at time 0

- The "initial" block is typically used to write test benches for simulation :
  - Specifies the stimulus to be applied to the design-under-test (DUT)
  - Specifies how the DUT outputs are to be

displayed / handled.
- Specifies the file where the waveform information is to be dumped.

```
module testbench_example;
  reg a, b, cin, sum, cout;

  initial
    cin = 1'b0;
  initial
    begin
      #5 a = 1'b1; b = 1'b1;
      #5 b = 1'b0;
    end
  initial
    #25 $finish;
endmodule
```

The "always" block

- An "always" statement starts at time 0 and executes the statements inside the block repeatedly, and never stops.

```
module generating_clock;
  output reg clk;
  initial
    clk = 1'b0; // initialized to 0 at time 0
  always
    #5 clk = ~clk; // Toggle after time 5 units
  initial
    #500 $finish;
endmodule
```

- "initial" and "always" blocks can coexist within the same Verilog module.
- They all execute concurrently, "initial" only once & "always repeatedly.

- The @(event-expression) part is required for both combinational & sequential ckt descriptions.

Basic syntax of "always" block:

```
always @(event-expression)
begin
    sequential-statement-1;
    sequential-statement-2;
    ...
    sequential-statement-n;
end
```

- Only "reg" type variable can be assigned within an "initial" or "always" block.
- Basic reason:
  - The sequential "always" block executes only when the event expression triggers.
  - At other times the block is doing nothing.
  - An object being assigned to must therefore remember the last value assigned (not continuously driven).
  - So, only "reg" type variables can be assigned within the "always" block.
  - Of course, any kind of variable may appear in the event expression (reg, wire, etc).

a) begin -- end

```
begin
  sequential-statement-1;
  sequential-statement-2;
  ...
  sequential-statement-n;
end
```

→ if n=1, "begin... end" is not required

b) if ... else

```
if (<expression 1>)
  seq-statement;
else if (...)
  ...
else default-statement;
```

a single statement or a group of statements within "begin -- end"

c) case

```
case (<expression>)
  expr1 : seq-statement;
  expr2 : seq-statement;
  ...
  exprn : seq-statement;
  default : default-statement;
end case;
```

Two variations: "case ?" and "case x"
- The "case ?" statement treats all "?" values in the case alternatives or the case expression as don't care.

- The "casex" statement treats all "x" and "z" values in the case item as don't cares.

```
reg [3:0] state; integer next_state;
casex (state)
    4'b1xxx : next_state = 0;
    4'bx1xx : next_state = 1;
    4'bxx1x : next_state = 2;
    4'bxxx1 : next_state = 3;
    default : next_state = 0;
endcase
```

d) "while" loop

```
while (<expression>)
    sequential_statement;
```

e) "for" loop

```
for (expr1; expr2; expr3)
    sequential_statement
```

```
integer mycount;
reg [100:2] data;
integer i;
initial
    for (mycount = 0; mycount <= 255; mycount = mycount + 1)
        $display ("My count : %d", mycount);
initial
    for (i = 1; i <= 100; i = i+1)
        data [i] = 1'b0;
```

f) "repeat" loop → fixed no. of times

repeat (<expression>)
    sequential-statement;

g) "forever" loop → executes forever until $finish is encountered in testbench.

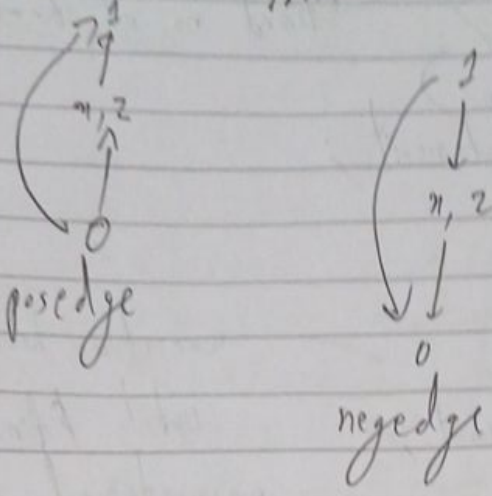
forever
    sequential-statement;

The "forever" loop is typically used along with timing specifier.
- If delay is not specified, the simulator would execute this statement indefinitely without advancing time
- Rest of design will never be executed.

@ (event-expression)

- The event expression specifies the event that is required to resume execution of the procedural block.
- The event can be any one of the following:
  a) Change of a signal value.
  b) Positive or negative edge occurring on signal (posedge or ~~negat~~ negedge).
  c) List of above-mentioned events, separated by "or" or comma.
- A "posedge" is any transition from {0, x, z} to

2 ; and from 0 to {z, x}.



Examples

- @ (in) // "in" changes
- @ (a or b or c) // any of 'a', 'b' or 'c' changes
- @ (a, b, c) // " " " " " "
- @ (posedge clk) // positive edge of "clk"
- @ (posedge clk or negedge reset)
- @ (*) // any variable changes

// D flip-flop with synchronous set and reset

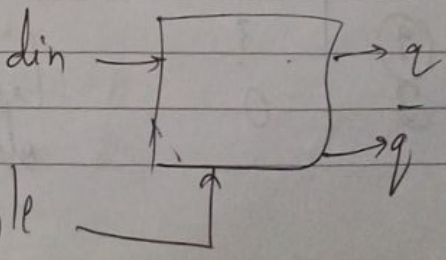
```
module dff (q, qbar, d, set, reset, clk);
  input d, set, reset, clk;
  output reg q; output qbar;

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

// D flip-flop with asynchronous 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 or negedge set or negedge reset)
    begin
      if (reset == 0) q <= 0;
      else if (set == 0) q <= 1;
      else q <= d;
    end
endmodule
```

// transparent latch with enable

```
module latch (q, qbar, din, enable);
  input din, enable;
  output reg q; output qbar;

  assign qbar = ~q;

  always @ (din or enable)
    begin
      if (enable) q = din;
    end
endmodule
```



latch => level triggers

```
// A combinational logic example
module mux21 (in1, in0, s, f);
    input in1, in0, s;
    output reg f;

    always @ (in1 or in0 or s)
        if (s)
            f = in1;
        else
            f = in0;
endmodule

// Sequential logic example

module incomp_state_spec (curr_state, flag);
    input [0:1] curr_state;
    output reg [0:1] flag;

    always @ (curr_state)
        case (curr_state)
            0, 1 : flag = 2;
            3    : flag = 0;
        endcase
endmodule
```

curr_state	flag	
0	2	⇒ if curr_state is 2, then flag will not change.
1	2	
(2)	?	↳ flag will map to a storage element (latch).
3	0	

// A small modification

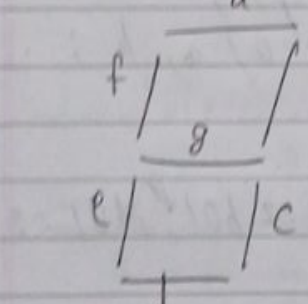
```
module incomp_state_spec (curr_state, flag);
  input [0:1] curr_state;
  output reg [0:1] flag;

  always @ (curr_state)
  begin
      flag = 0;
      case (curr_state)
        0, 1 : flag = 2;
        3    : flag = 0;
      endcase
  end
endmodule
```

. It is up to the designer to code the design in such a way that latch can be avoided where possible.

// A simple 4-function ALU

```
module ALU_4bit (f, a, b, op);
  input [1:0] op;   input [7:0] a, b;
  output reg [7:0] f;

  parameter ADD = 2'b00, SUB = 2'b01, MUL = 2'b10,
            DIV = 2'b11;
  always @ (*)
      case (op)
          ADD : f = a+b;
          SUB : f = a-b;
```

```
        MUL : f = a * b;
        DIV : f = a a/b;
    end case
endmodule

module priority_encoder (in, code);
    input [7:0] in;
    output reg [2:0] code;

    always @ in
        begin
            if (in[0]) code = 3'b000;
            else if (in[1]) code = 3'b001;
            else if (in[2]) code = 3'b010;
            else if (in[3]) code = 3'b011;
            else if (in[4]) code = 3'b100;
            else if (in[5]) code = 3'b101;
            else if (in[6]) code = 3'b110;
            else (in[7]) code = 3'b111;
        end
endmodule
```



Segment bit assignment:
(a, b, c, d, e, f, g)

A seg glows when the corresponding bit of seg is 0.

```
module bcd_to_7seg (bcd, seg);
  input [3:0] bcd;
  output reg [6:0] seg;

  always @(bcd)
    case
        0 : seg = 6'b0000001;
        1 : seg = 6'b1001111;
        2 : seg = 6'b0010010;
        3 : seg = 6'b0000110;
        4 : seg = 6'b1001100;
        5 : seg = 6'b0100100;
        6 : seg = 6'b0100000;
        7 : seg = 6'b0001111;
        8 : seg = 6'b0000000;
        9 : seg = 6'b0000100;
      default : seg = 6'b1111111;
    endcase
endmodule

// An n-bit comparator
module compare (A, B, lt, gt, eq);
  parameter word_size = 16;
  input [word_size-1:0] A, B;
  output reg lt, gt, eq;

  always @(*)
    begin
      gt = 0; lt = 0; eq = 0;
```

```
        if (A > B) gt = 1;
        else if (A < B) lt = 1;
        else eq = 1
    end
endmodule
```

```
module alu_example (alu_out, A, B, operation, en);
  input [2:0] operation; input [3:0] A, B;
  input en;
  output [7:0] alu_out; reg [7:0] alu_reg;

  assign alu_out = (en == 1) ? alu_reg : 4'b2;

  always @ (*)
    case (operation)
      3'b000 : alu_reg = A + B;
      3'b001 : alu_reg = A - B;
      3'b011 : alu_reg = ~A;
      default : alu_reg = 4'b0;
    end case
endmodule
```

- 2 types of procedural assignments :
  - Blocking (denoted by "=")
  - non-blocking (denoted by "<=")

i) Block assignment

- General syntax

```
variable_name = [delay_or_event_control] expression;
```

- The "=" operator is used to specify blocking assignment.
- Blocking assignment statements are executed in order they are specified in a procedural block.
  - the target of an assignments gets updated before the next sequential statement in the procedural block is executed.
  - they do not block execution of statements in other procedural blocks.
- Recommended to combinational ckts.

ii) Non-blocking assignment

General syntax:

variable_name <= [delay_or_event_control] expression;

- Non-block assignment statements allow scheduling of assignments without blocking execution of statements that follow the within the procedural block.
  - the assignment to the target gets scheduled for the end of simulation cycle (at the end of the procedural block)
  - Statements subsequent to the instruction under consideration are not blocked by assignment.
  - Allows concurrent procedural assignment, suitable for subsequent logic.

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

→ results

a = 35 at time 5

b = 15 " "

c = -10 " "

different from blocking assignments.

Swapping values of two variables "a" and "b"

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

ex

```
module non_blocking_assgn;
  integer a, b, c, d;
  reg clock;
  always @ (posedge clk)
    begin
      a <= b+c;
      d <= a-3;
      b <= d+10;
      c <= c+1;
    end
```

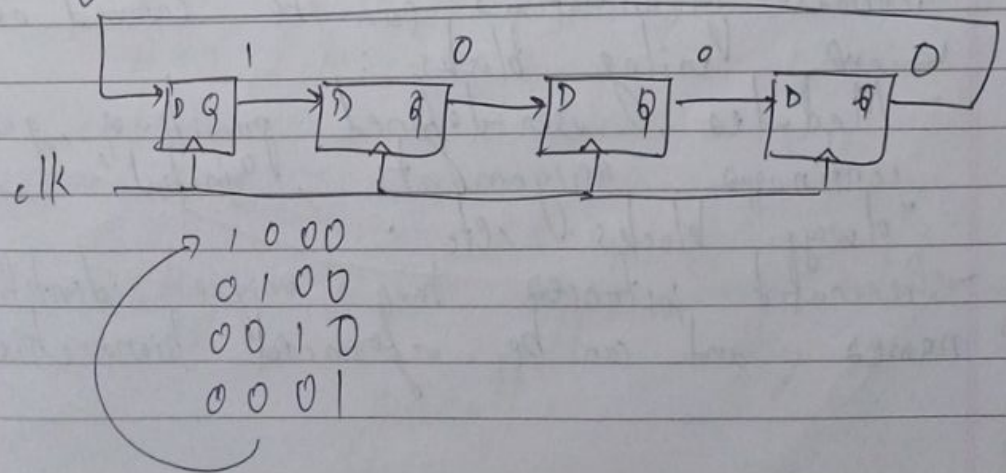
```
initial
  begin
    $monitor ($time, "a = %4d, b = %4d,
              c = %4d, d = %4d ", a, b, c, d);

    a = 30; b = 20; c = 15; d = 5;
    clock = 0;
    forever #5 clock = ~clock;
  end

initial
    # 100 $finish;
endmodule
```

- It is recommended not to use blocking, & non-blocking assignments in same always block.
- A variable cannot appear as the target of block a blocking & non-blocking assignment

✓ `x = x + 5;`

✗ `x <= y;`

// 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
```

- Whenever a clock-triggered assignment → use non-blocking

- Generate Blocks

- "generate" statements allow Verilog code to be generated dynamically before the simulation or synthesis begins.
  - Very convenient to create parameterized module descriptions.
- Generate instantiations can be carried out for various Verilog blocks:
  - Modules, user-defined primitives, gates, "continuous assignments", "initial" and "always" blocks, etc.
  - Generated instances have unique identifiers named and can be referenced hierarchically.

```
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 xorblp
            xor xG (f[p], a[p], b[p]);
        end
    endgenerate
endmodule

module generate_test;
    reg [15:0] x, y;
    wire [15:0] out;

    xor_bitwise G (.f(out), .a(x), .b(y));

    initial begin
        $monitor ("x: %b, y: %b, out: %b", x, y,
                  out);
        x = 16'haaaa; y = 16'h00ff;
        #10 x = 16'h0f0f; y = 16'h3333;
        #20 $finish;
    end
endmodule
```

generating N-bit ripple carry adder using genvar & generate

```
module RCA (carry_out, sum, a, b, carry_in);
  parameter N=8;
  input [N-1:0] a, b; input carry_in;
  output [N-1:0] sum; output carry_out;
  wire [N:0] carry; // carry[N] is carry_out

  assign carry[0] = carry_in;
  assign carry_out = carry[N];

  genvar i;
  generate for (i=0; i<N; i=i+1)
    begin fa_loop
      wire t1, t2, t3;
      xor G1 (t1, a[i], b[i]); G2 (sum[i], t1,
                                   carry[i]);
      and G3 (t2, a[i], b[i]), G4 (t3, t1, carry[i]
      or G5 (carry[i+1], t2, t3);
    end
  end generate
endmodule
```

- Some of the relative hierarchical instance names that are generated are:
  - fa_loop[0].G1, fa_loop[1].G2, fa_loop[7].G
    etc
- Some of the nets ("wires") that are generated are:
  - fa_loop[0].t1, fa_loop[1].T2, fa_loop[0].t
    etc.

User defined Primitives (UDP)

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

- for combinational functions, truth table entries are specified as:
  <input 1> <input 2> ... <input N> : <output>;

- for sequential functions, state table entries are specified as:
  <input 1> <input 2> ... <input N> : <present-state> : <next-state>

Some rules for using UDPs

- The input terminals to a UDP can only be scalar variables.
  - Multiple input terminals can be used
  - The input terminals are declared as "input".
  - Input entries in the table must be in the same order as the "input" terminal list.
- Only one scalar output terminal must be used
  - The output terminal must appear in the by
  - for combinational UDPs, the output terminal is declared as "output".
  - for sequential UDPs, the output terminal is declared as "reg".
- for sequential UDPs, the state can be initialized with an "initial" statement".

Some Guidelines

- User defined Primitives (UDPs) model functionality only
  - They do not model timing or process technology
- A functional block can be modelled as a UDP only if it has exactly one output.
  - If a block has more that one outputs, it has to be modeled as a module.
  - As an alternative, multiple UDPs can be used, one per output.
- Inside the simulator, a UDP is typically implemented as a lookup table in memory.
- The UDP state tables should be specified as completely as possible.
  - For unspecified cases, the output is set to "x".

```
// 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 ;
        0   1   0  :  1 ;
        0   1   1  :  0 ;
        1   0   0  :  1 ;
        1   0   1  :  0 ;
        1   1   0  :  0 ;
        1   1   1  :  1 ;
    end table
endprimitive
```

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

```
// full adder carry generation
// Using don't care ("?")

primitive udp_cy (cout, a, b, c);
    input a, b, c;
    output cout;
    table
    // a  b  c   cout
       0  0  ?  : 0;
       0  ?  0  : 0;
       ?  0  0  : 0;
       1  1  ?  : 1;
       1  ?  1  : 1;
       ?  1  1  : 1;
    end table
endprimitive

// Instantiating UDP's
// A full adder description

module full_adder (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
```

// A level-sensitive D type latch

```
primitive Dlatch (q, d, clk, clr);
    input d, clk, clr;
    output reg q;

    initial
        q = 0;  // This is optional

    table
    // d   clk  clr   :  q   :  q_new
        ?    ?    1   :  ?   :  0 ;  // latch is clear
        0    1    0   :  ?   :  0 ;  // latch is reset
        1    1    1   :  ?   :  1 ;  // latch is set
        ?    0    0   :  ?   :  - ;  // remain previous state

    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 ;  // FF is cleared
        ?   (10) :  ?  :  - ;  // ignore -ve edge of "clr"
       (10)  0   :  1  :  0 ;  // FF toggles @ +ve edge
       (10)  0   :  0  :  1 ;  // -do-
       (0?)  0   :  ?  :  - ;  // ignore +ve edge of clk
```

```
endtable
endprimitive

// constructing a 6-bit ripple counter using T flip flops
module ripple-counter (count, clk, clr);
    input clk, clr;
    output [5:0] count;

    TFF F0 (count[0], clk, clr);
    TFF F1 (count[1], count[0], clr);
    TFF F2 (count[2], count[1], clr);
    TFF F3 (count[3], count[2], clr);
    TFF F4 (count[4], count[3], clr);
    TFF F5 (count[5], count[4], clr);
endmodule

// A negative edge sensitive JK flip-flop
primitive JK FF (q, j, k, clk, clr);
    input j, k, clk, clr;
    output reg q;
    table
    // j  k  clk  clr  q  q_new
       ?  ?  ?    1    ?  : 0 ;  // clear
       ?  ?  ?    (10) ?  : - ;  // ignore
       0  0  (10) 0    ?  : - ;  // no change
       0  1  (10) 0    ?  : 0 ;  // reset condition
       1  0  (10) 0    ?  : 1 ;  // set condition
       1  1  (10) 0    0  : 1 ;  // toggle condition
       1  1  (10) 0    1  : 0 ;  // toggle condition
       ?  ?  (01) 0    ?  : - ;  // no change
```

- '?' cannot be used in output field
- 'z' can only be specified in output field.
- shortcut "x" → for rising edge, instead of (01).
- shortcut "f" → for falling edge, instead of (10).
- shortcut (*) indicates any value change in signal.

Testbench

How to write test benches?

- Create a dummy template
  - Declare inputs to the design-under-test (DUT) as "reg", and the outputs as "wire".
    - Because we have to initialize the DUT inputs inside procedural block(s), typically "initial", where only "reg" type variables can be assigned.
  - Instantiate the DUT.
- Initialization and Monitoring
  - Assign some known values to DUT inputs
  - Monitor the DUT outputs for functional verification

- For synchronous sequential ckts, we need clock generation logic.

The Simulator directives

- $display ("<format>", expr1, expr2, ...);
  - Used to print the immediate values of text or variables to stdout.
- $monitor ("<format>", var1, var2, ...)
  - Similar in syntax to $display, but does not print immediately.

- It will print the value(s) whenever the value of some variable(s) in the given list changes.
- Has the functionality of event-driven print.

1. $finish;
   - Terminates the simulation process.
1. $dumpfile (<filename>);
   - Specifies the file that will be used for storing the values of the selected variables so that they can be graphically visualized later.
   - The file typically has an extension .vcd (Value Change Dump), and contains information about any value changes on the selected variables.

- $dumpoff;
  - This directive stops the dumping of variables. All variables are dumped with "x" values and the next change of variables will not be dumped.
- $dumpon;
  - This directive starts previously stopped dumping of variables.

- $dumpvars (level, list_of_variables_or_modules);
  - Specifies which variables should be dumped on to the .vcd file.
  - Both the parameters are optional; if both are omitted all variables are dumped.
  - If level = 0, then all variables within the modules from the list will be dumped. If any module from the list contains module

instances, then all variables from these modules will also be dumped.

- If level = 1, then only listed variables and variables of listed modules will be dumped.

- $dumpall;
  - The current values of all variables will be written to the file, irrespective of whether there has been any change in their values or not.

- $dumplimit (filesize);
  - Used to set the maximum size of the .vcd file.

// testbench for Full adder using $display

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

  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, sum= %b, cout= %b",
                     $time, a, b, c, sum, cout);
        end
      #5 $finish
    end
endmodule
```

generate for output file called adder

iverilog -o adder fa.v fa_test.v

// testbench for shift register

module shift_test;
reg clk, clr, in; wire out; integer i;
shiftreg_4bit SR (clk, clr, in, out);

initial
begin clk = 1'b0; #2 clr = 0; #5 clr = 1; end

always #5 clk = ~clk;

initial begin #2;
repeat (2)
begin #10 in = 0; #10 in = 0; #10 in = 1;
#10 in = 1; end
end

initial
begin
$dumpfile ("shiftreg.vcd");
$dumpvars (0, shift_test);
#200 $finish;
end
endmodule

- Shall display 1 if outputs are correct; and display 0 otherwise!

```
module fulladder_test;
  reg a, b, c;
  wire s, cout;
  integer correct;

  fulladder FA (a, b, c, s, cout);

  initial
    begin
      correct = 1;
      #5 a=1; b=1; c=0; #5;
        if ((s != 0) || (cout != 1))
          correct = 0;
      #5 a=1; b=1; c=1; #5;
        if ((s != 1) || (cout != 1))
          correct = 0;
      #5 a=0; b=1; c=0; #5;
        if ((s != 1) || (cout != 0))
          correct = 0;
      #5 $display ("%d", correct);
    end
endmodule
```

- The system task $random can be used to generate a random number.
- It is called as: $random (<seed>)
  - The value of <seed> is optional and is used to ensure that the same sequence

of random numbers are generated each time the test is run.

```
module test_adder;
  reg [7:0] a, b;
  wire [7:0] sum;
  integer myseed;

  adder ADD (sum, cout, a, b);

  initial myseed = 15;

  initial
    begin
      repeat (5)
        begin
          a = $random (myseed);
          b = $random (myseed); #10;
          $display ("T: %3d, a: %h, b: %h,
            sum: %h", $time, a, b, sum);
        end
    end
endmodule
```

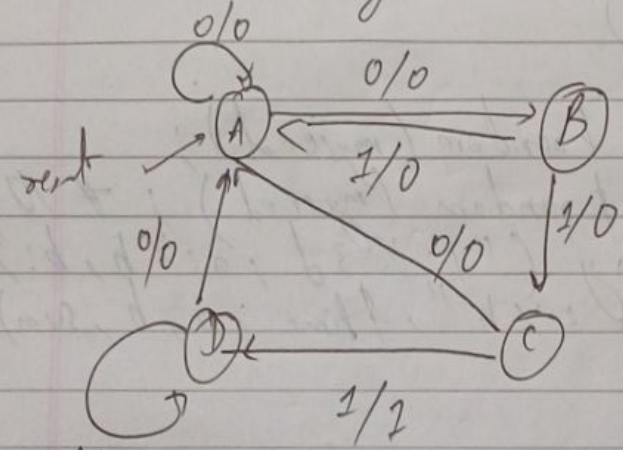
Finite State Machine (FSM)

- A FSM can be represented either in the form of a state table or in the form of a state diagram/transition diagram
  - Variation exist, e.g. Algorithmic State Machine (ASM) chart.

State table "111" detection

Reset	PS	Input	NS	Output
1	-	-	A	0
0	A	0	A	0
0	A	1	B	0
0	B	0	A	0
0	B	1	C	0
0	C	0	A	0
0	C	1	D	1
0	D	0	A	0
0	D	1	D	1

State diagram

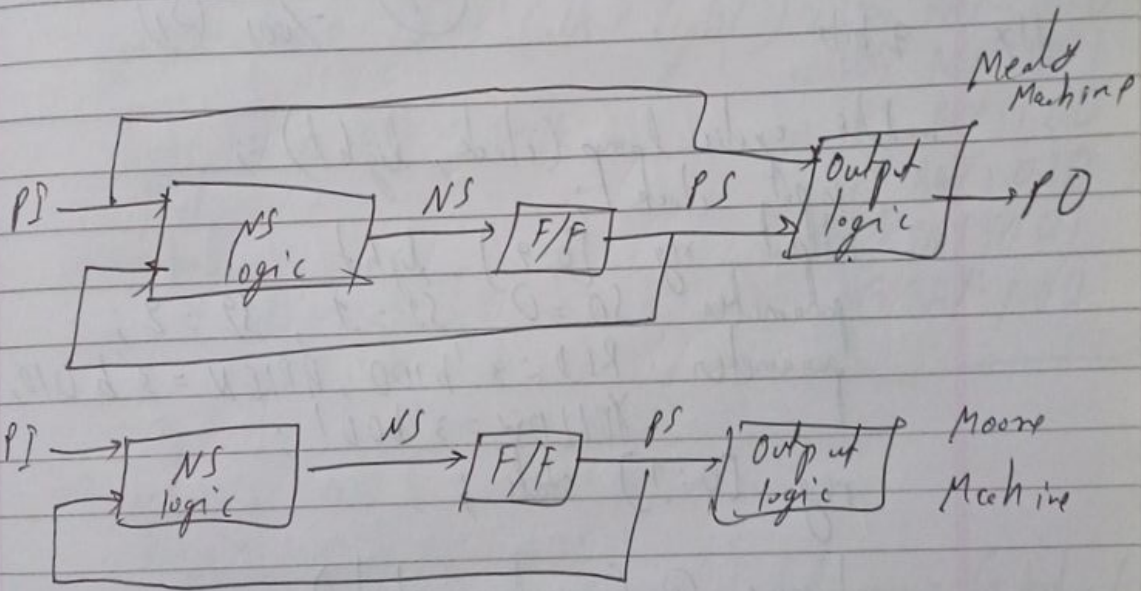


Mealy and Moore FSM

- A deterministic FSM can be mathematically defined as a 6-tuple $(\Sigma, \Gamma, S, s_0, \delta, \omega)$ where Σ (sigma) is the set of input combinations, Γ (gamma) is the set of output combinations, S (capital S) is a finite set of states, $s_0 \in S$ is the initial state, δ (delta) is the

state transition function, and ω is the output function.

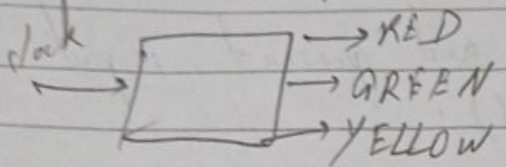
- Here, $\delta : S \times \Sigma \rightarrow S$
  - Present state (PS) and present input determines the next state (NS).
- for Mealy machine, $\omega : S \times \Sigma \rightarrow \Gamma$ (output depends on state + inputs)
- For Moore machine, $\omega : S \rightarrow \Gamma$ (output depends only on the state).

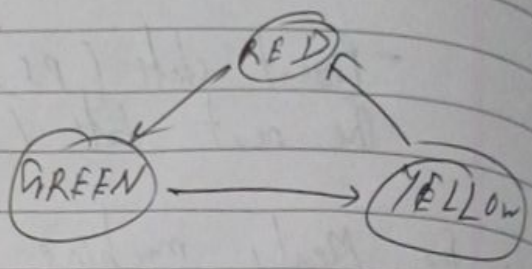


Ex 1

- There are 3 lamps, RED, GREEN, and YELLOW, that should glow cyclically with a fixed time interval.
- Some observations:
  - The FSM will have 3 states, corresponding to the glowing state of the lamps.
  - The input set is NULL; state transition will

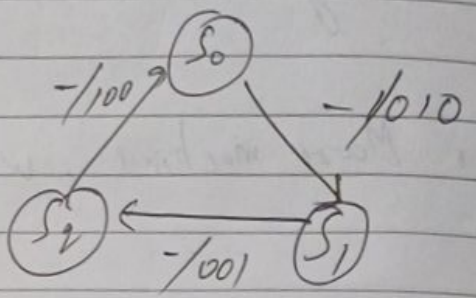
occurs whenever clock signal comes.
- This is a Moore machine, since the lamp that will glow only depends on the state and not on the inputs (here null).





00 S_0 → RED
01 S_1 → GREEN
10 S_2 → YELLOW

11× 2 bits




```
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                    // S0 means RED
             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

//testbench

module test_cyclic_lamp;
  reg clk;
  wire [0:2] light;

  cyclic_lamp LAMP (clk, light);

  always #5 clk = ~clk;
  initial
    begin
      clk = 1'b0;
      #100 $finish;
    end
  initial
    begin
      $dumpfile("cyclic.vcd"); $dumpvars(0, test_cyclic_lamp);
      $monitor ($time, "RGY : %b", light);
    end
endmodule
```

Output

```
0  RGY : xxx
5  RGY : 100
15 RGY : 010
25 RGY : 001
35 RGY : 100
45 RGY : 010
55 RGY : 001
65 RGY : 100
75 RGY : 010
85 RGY : 001
95 RGY : 100
```

- Some comments on solution:
  - The synthesis tool will generate five flip-flops: 2 for state, and 3 for light.
  - The three output lines are also getting stored in flip-flops.
    - We have used non-blocking assignment triggered by clock edge.
  - But actually we do not need separate flip-flops for the outputs, as the outputs can be directly generated from the state.
  - How to achieve this?
    - Modify the Verilog code such that all assignments to light is made in a separate "always" block.
    - Use blocking assignment, triggered by state change, and not by clock.

```
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 clk)
  case (state)
    S0 : state <= S1;
    S1 : state <= S2;
    S2 : state <= S0;
    default : state <= S0;
```

```
always @(state)
    case (state)
        S0 : light = RED ;
        S1 : light = GREEN ;
        S2 : light = YELLOW ;
        default : light = RED ;
    endcase
endmodule
```

- The synthesis tool will be generating only 2 flip-flops corresponding to the first clock-triggered "always" block.
- The second "always" block will be generating a combinational ckt that takes state as input and produces light as outputs.

state ($S_1 S_0$)	light	RGY
S0 : 00		100
S1 : 01		010
S2 : 10		001
11		xxx

minimization:

$R = S_0 \cdot S_1'$

$G = S_0$

$Y = S_0$

R	0	1
0	(1)	0
1	0	x

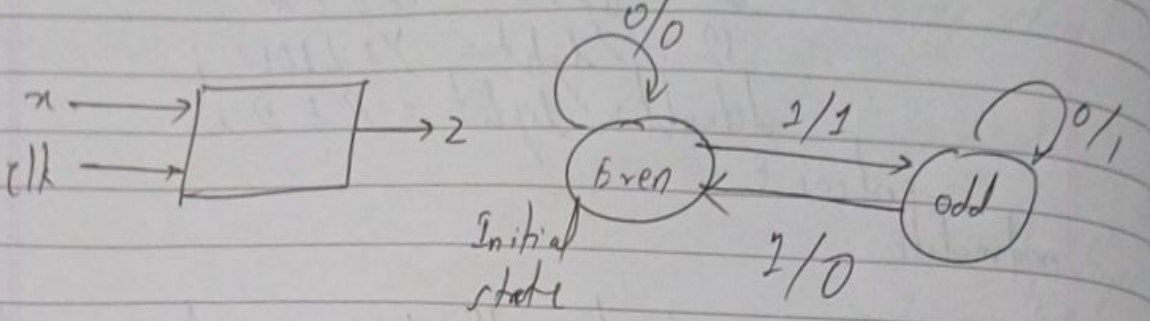
G	0	1
0	0	(1)
1	0	(x)

Ex 2

Design of a serial parity detector

- A continuous stream of bits is fed to a ckt in synchronism with a clock. The ckt will be generating a bit stream as output, where a 0 will indicate

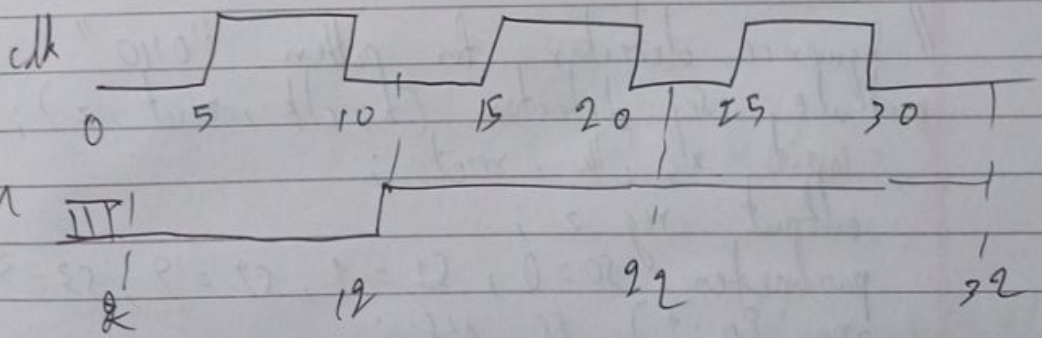
"even number of 1's seen so far" and a 1 will indicate "odd number of 1's seen so far".
- Also a Moore Machine.




```
module parity_gen (x, clk, z);
  input x, clk; output reg z;
  reg even_odd; // the machine state
  parameter EVEN=0, ODD=1;
  always @ (posedge clk)
    case (even_odd)
      EVEN : even_odd <= x ? ODD : EVEN;
      ODD : even_odd <= x ? EVEN : ODD;
      default : even_odd <= EVEN;
    endcase
  always @ (even_odd)
    case (even_odd)
      EVEN : z = 0;
      ODD : z = 1;
    endcase
endmodule
```

- This design will not cause the synthesis tool to generate a latch for the output "z".

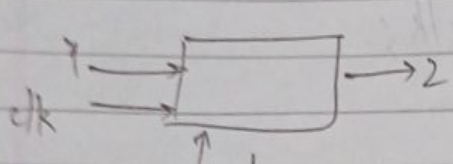
```
//test bench

module test_parity;
  reg clk, x; wire z;
  parity_gen PAR (x, clk, z);
  initial
    begin
      $dumpfile ("parity.vcd"); $dumpvars (0, test_parity);
      clk = 1'b0;
    end
  always #5 clk = ~clk;

  initial
    begin
      #2  x=0; #10 x=1; #10 x=1; #10 x=1;
      #10 x=0; #10 x=1; #10 x=1; #10 x=0;
      #10 x=0; #10 x=1; #10 x=1; #10 x=0;
      #10 $finish;
    end
endmodule
```



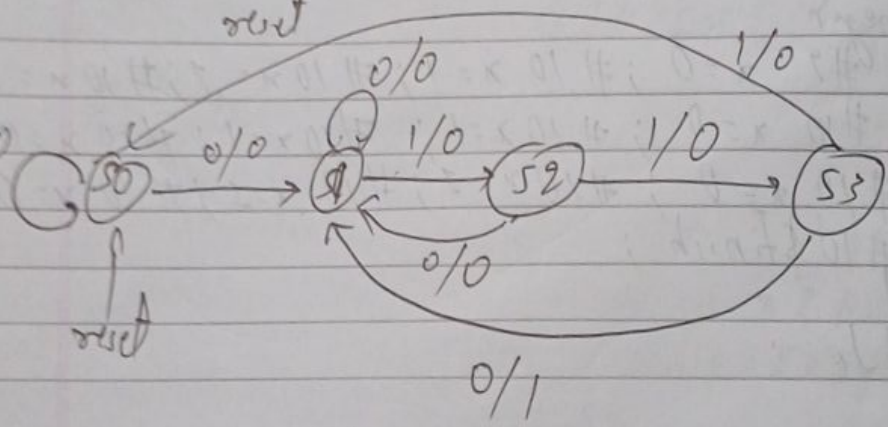
ex 3. Design of a sequence detector.
- A circuit accepts a serial bit stream "x" as input and produces a serial bit stream "z" as output.

- Whenever the bit pattern "0110" appears in the input stream, it outputs z = 1; In all other times, z = 0.
- Overlapping occurrences of the pattern are also detected
- This is a Mealy Machine.

x : 0 1 1 0 1 1 0
z : 0 0 0 1 0 0 1



The input bits "x" are applied in synchronism with the clock



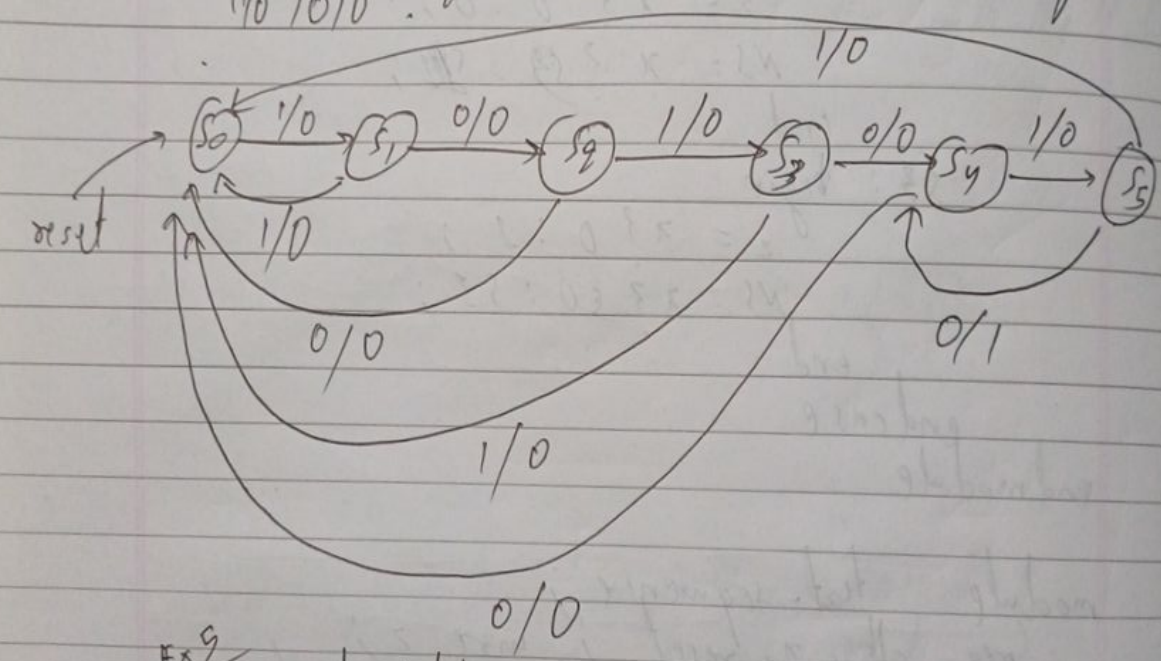

```
// Sequence detector for pattern "0110"
module seq_detector (x, clk, reset, z);
  input x, clk, reset;
  output reg z;
  parameter S0 = 0, S1 = 1, S2 = 2, S3 = 3;
  reg [0:1] PS, NS;
  always @ (posedge clk or posedge reset)
    if (reset) PS <= S0;
    else PS <= NS;
```

```
always @(PS, x)
    case (PS)
        S0 : begin
                z = x ? 0 : 0;
                NS = x ? S0 : S1;
             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
    endcase
endmodule

module test_sequence;
    reg clk, x, reset; wire z;
    seq_detector SEQ (x, clk, reset, z);
    initial
        begin
            $dumpfile ("sequence.vcd"); $dumpvars (0, test_sequence);
            clk = 1'b0; reset = 1'b1;
            #15 reset = 1'b0;
        end
    always #5 clk = ~clk;
```

```
initial
  begin
    #12 x=0; #10 x=0; #10 x=1; #10 x=1;
    #10 x=0; #10 x=1; #10 x=1; #10 x=0;
    #10 x=0; #10 x=1; #10 x=1; #10 x=0;
    #10 $finish;
  end
endmodule
```

Ex4

Design a sequence detector for the bit pattern "1010".



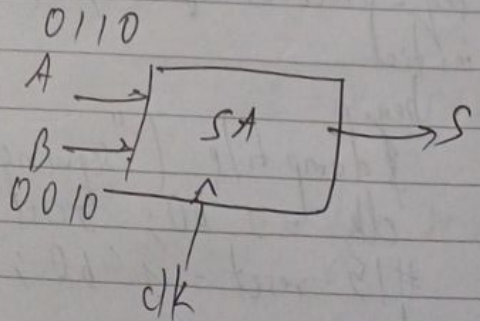
Ex5

Serial Adder:

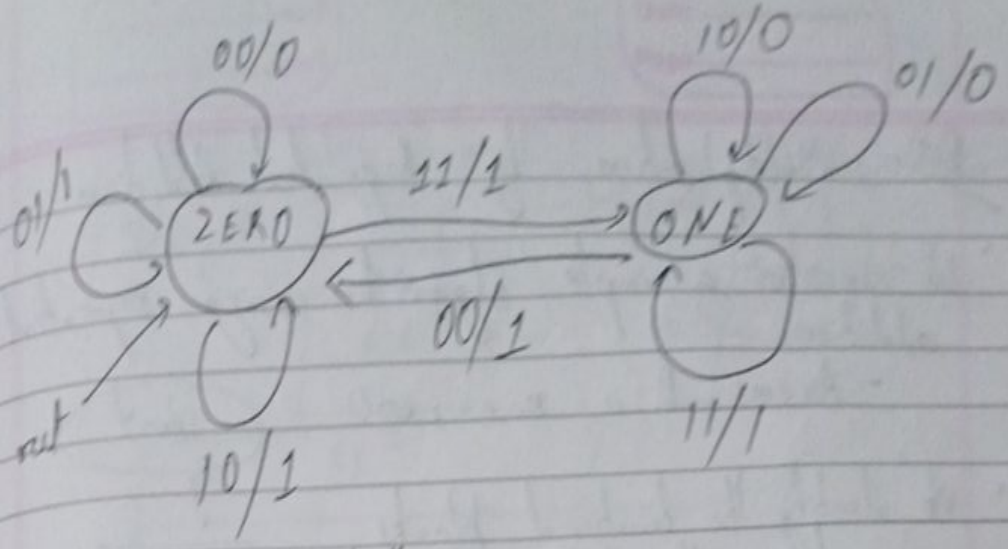
A: 0 1 1 0

B: 0 0 1 0

S: 0 1 0 0



State : CARRY (1 bit)



Datapath & Controller

- In a complex digital system, the hardware is typically partitioned into two ways:

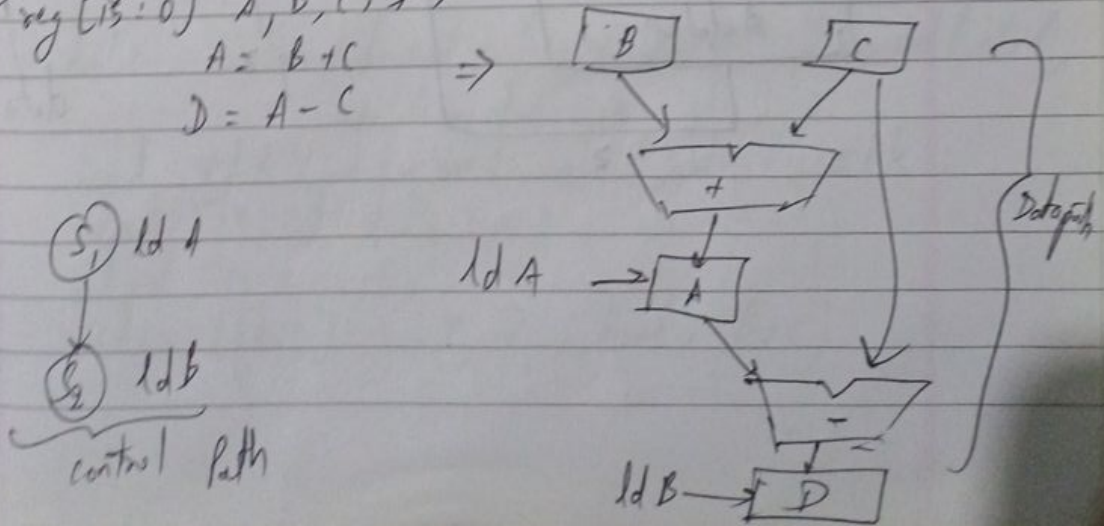
a) Data path, which consists of the functional units where all computations are carried out.
- Typically consists of registers, multiplexers, buses, adders, multipliers, counters, and other functional blocks.

b) Control Path, which implements a finite-state machine and provides control signals to the data path in proper sequence.
- In response to the control signals, various operations are carried out by the data path.
- Also takes inputs from the data path regarding various status information.

// reg [15:0] A, B, C, D;

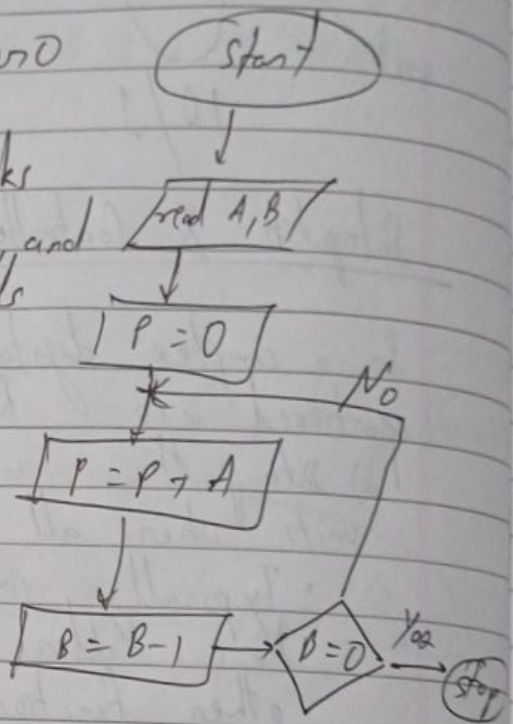
A = B + C

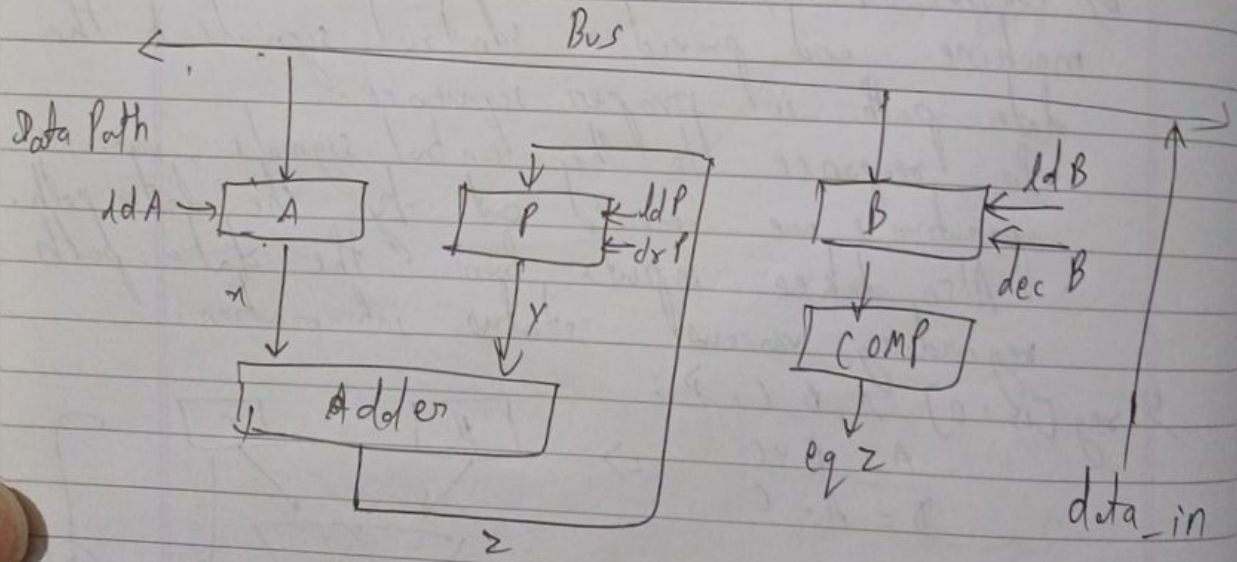
D = A - C

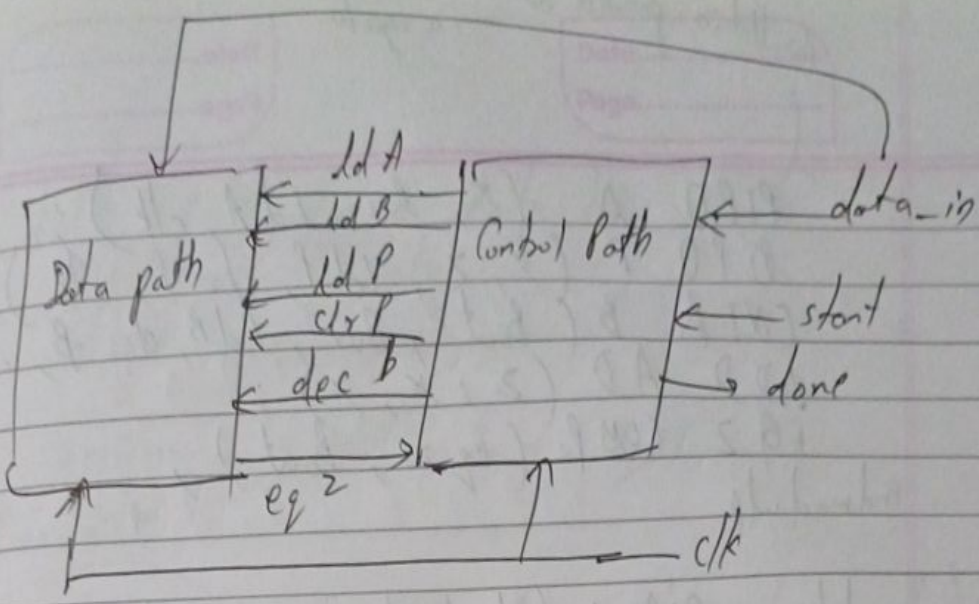


Ex: Multiplication by Repeated Addition

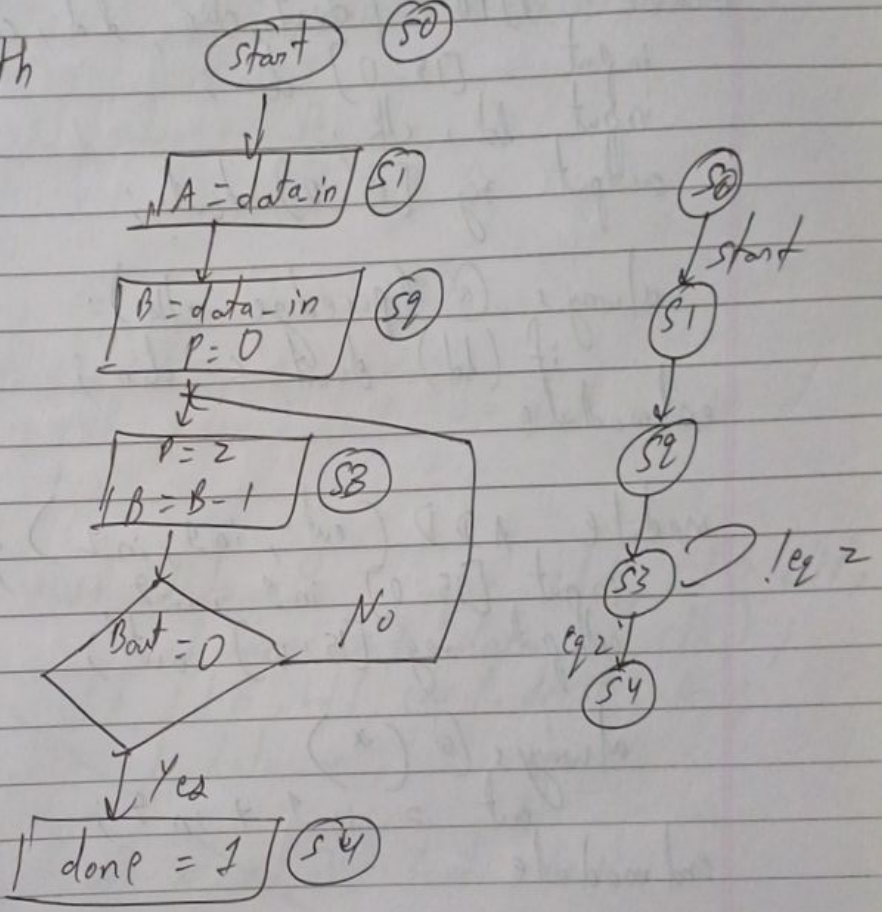
- We consider a simple algorithm using repeated addition
  - Assume B is non-zero
- We identify the functional blocks required in the data path, and the corresponding control signals
- Then we design the FSM to implement the multiplication algorithm using the data path.







Control Path




```
module MUL_datapath (eq 2, LdA, LdB, LdP, clrP,
                     dec B, data_in, clk);
input LdA, LdB, LdP, clrP, dec B, clk;
input [15:0] data_in;
output eq 2;
wire [15:0] x, y, z, Bout, Bus;
```

parallel in parallel out → a register

```
PIPO A (X, Bus, LdA, clk);
PIPO P (Y, Z, LdP, clrP, clk);
CNTR B (Bout, Bus, LdB, decB, clk);
ADD AD (Z, X, Y);
EQ2 COMP (eq2, Bout);
endmodule

module PIPO1 (dout, din, ld, clk);
  input [15:0] din;
  input ld, clk;
  output reg [15:0] dout;

  always @ (posedge clk):
    if (ld) dout <= din;
endmodule

module ADD (out, in1, in2);
  input [15:0] in1, in2;
  output reg [15:0] out;

  always @ (*)
    out = in1 + in2;
end module

module PIPO2 (dout, din, ld, clr, clk);
  input [15:0] din;
  input ld, clr, clk;
  output reg [15:0] dout;
  always @ (posedge clk):
    if (clr) dout <= 16'b0;
    else if (ld) dout <= din;
endmodule
```

```
module IBZ (eqz, data);
    input [15:0] data;
    output eqz;

    assign eqz = (data == 0);
endmodule

counter
module CNTR (dout, din, ld, dec, clk);
    input [15:0] din
    input ld, dec, clk;
    output reg [15:0] dout;

    always @ (posedge clk)
        if (ld) dout <= din;
        else if (dec) dout <= dout - 1;
endmodule

module MUL-datapath (eqz, LdA, LdB, LdP,
                     clrP, decB, data_in, clk);
    input LdA, LdB, LdP, clrP, decB, clk;
    input [15:0] data_in;
    output eqz;
    wire [15:0] x, xz, bout, Bus;

    PIPO A (x, Bus, LdA, clk);
    PIPO P (y, z, LdP, clrP, clk);
    CNTR B (Bout, Bus, LdB,
```

The control path

```
module controller (LA d, Ld B, Ld P, clr P, dec B, done,
                   clk, eqz, start);
  input clk, eqz, start;
  output reg Ld A, Ld B, Ld P, clr P, dec B, done;

  reg [2:0] state;
  parameter S0 = 3'b000, S1 = 3'b001, S2 = 3'b010,
            S3 = 3'b011, S4 = 3'b100;
  always @(posedge clk)
    begin
      case (state)
        S0 : if (start) state <= S1;
        S1 : state <= S2;
        S2 : state <= S3;
        S3 : #2 if (eqz) state <= S4;
        S4 : state <= S4;
        default : state <= S0;
      endcase
    end
endmodule

  always @(state)
    begin
      case (state)
        S0 : begin #1 Ld A = 0; Ld B = 0; Ld P = 0;
                  clr P = 0; dec B = 0; end
        S1 : begin #1 Ld A = 1 ; end
        S2 : begin #1 Ld A = 0; Ld B = 1; clr P = 1; end
        S3 : begin #1 Ld B = 0; Ld P = 1; clr P = 0; dec B = 1;
             end
        S4 : begin #1 done = 1; Ld B = 0; Ld P = 0; dec B = 0; end
```

```
        default : begin #1 LdA=0; LdB=0; LdP=0;
                  clrP=0; decB=0; end
    endcase
  end
endmodule

// Testbench

module MUL_test;
  reg [15:0] data_in;
  reg clk, start;
  wire done;

  MUL_datapath DP(eqz, LdA, LdB, LdP, clrP, decB,
                  data_in, clk);
  controller CON(LdA, LdB, LdP, clrP, decB, done,
                 clk, eqz, start);

  initial
    begin
      clk = 1'b0;
      #3 start = 1'b1;
      #500 $finish;
    end
  always #5 clk = ~clk;
  initial
    begin
      #17 data_in = 17;
      #10 data_in = 5;
    end
```

```
initial
begin
    $monitor ($time, "%d %b" DP.Y, done);
    $dumpfile ("mul.vcd"); $dumpvars(0, MUL_tb);
end
endmodule
```

a variable from DP

result			
	0	x	x
	1	x	0
	35	0	0
	45	17	0
	55	34	0
	65	51	0
	75	68	0
	85	85	0
	88	85	1

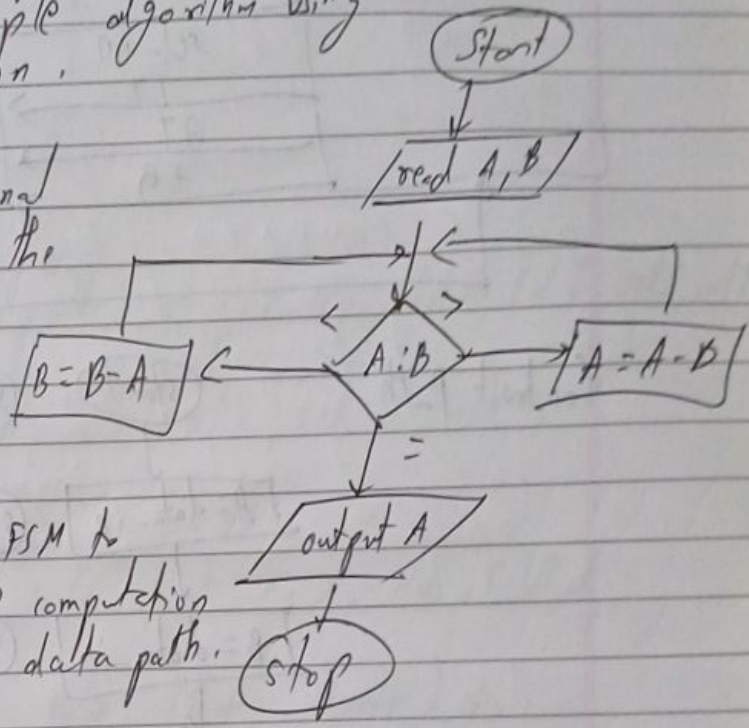
A better style of Modeling Data / Control Path

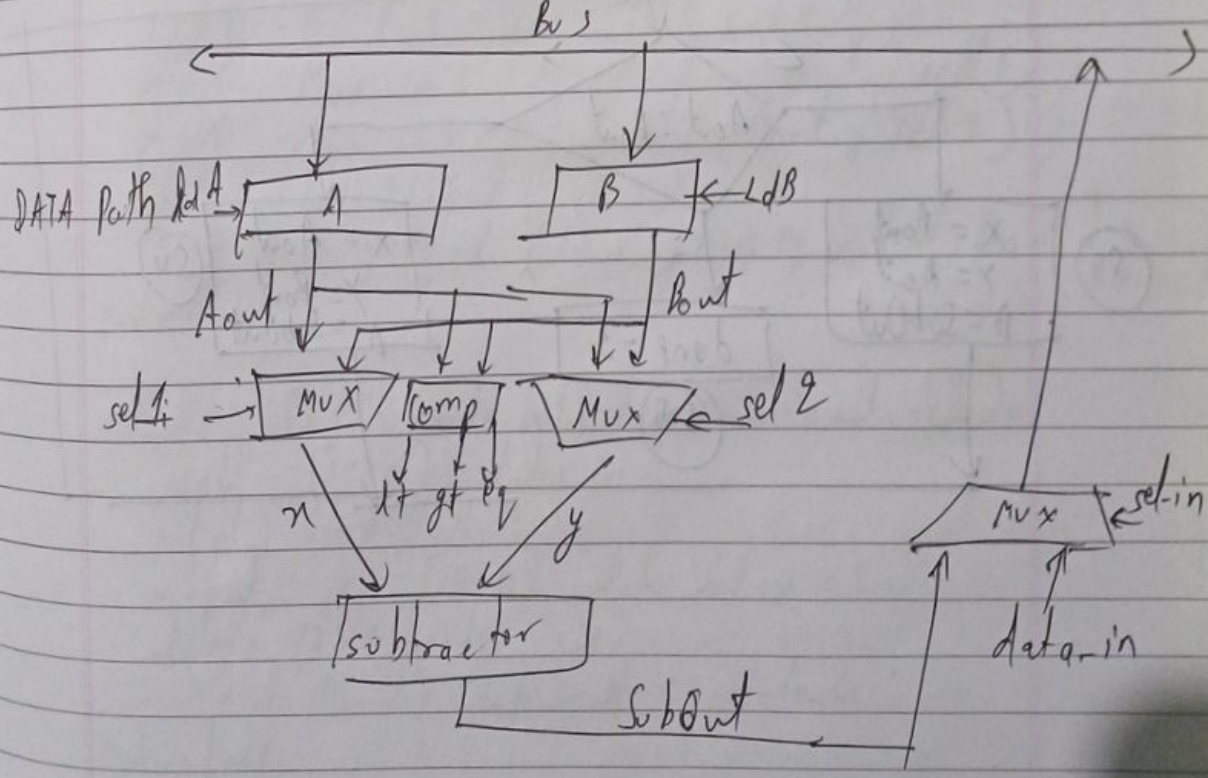
- In the previous example, in the "always" block activated by clock edge, path state change as well as computation of the next state is performed.
- A better and recommended approach:
  - Only trigger the state change in the clock activated "always" block.
  - In a separate "always" block using blocking assignments, compute the next state.
  - As in the previous example, in a separate

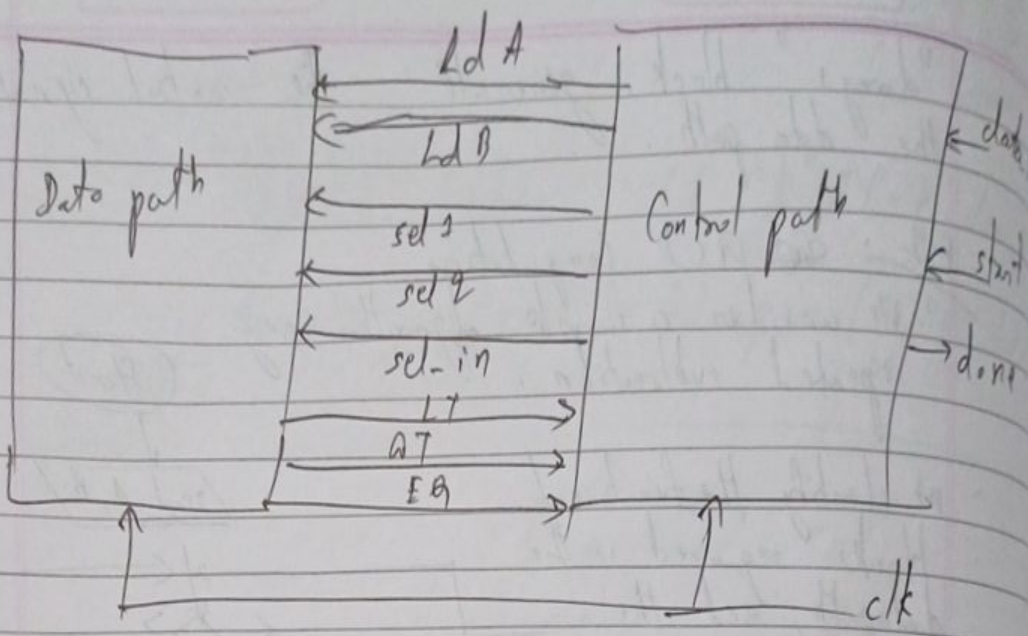
"always" block, generate the control signals for the data path.

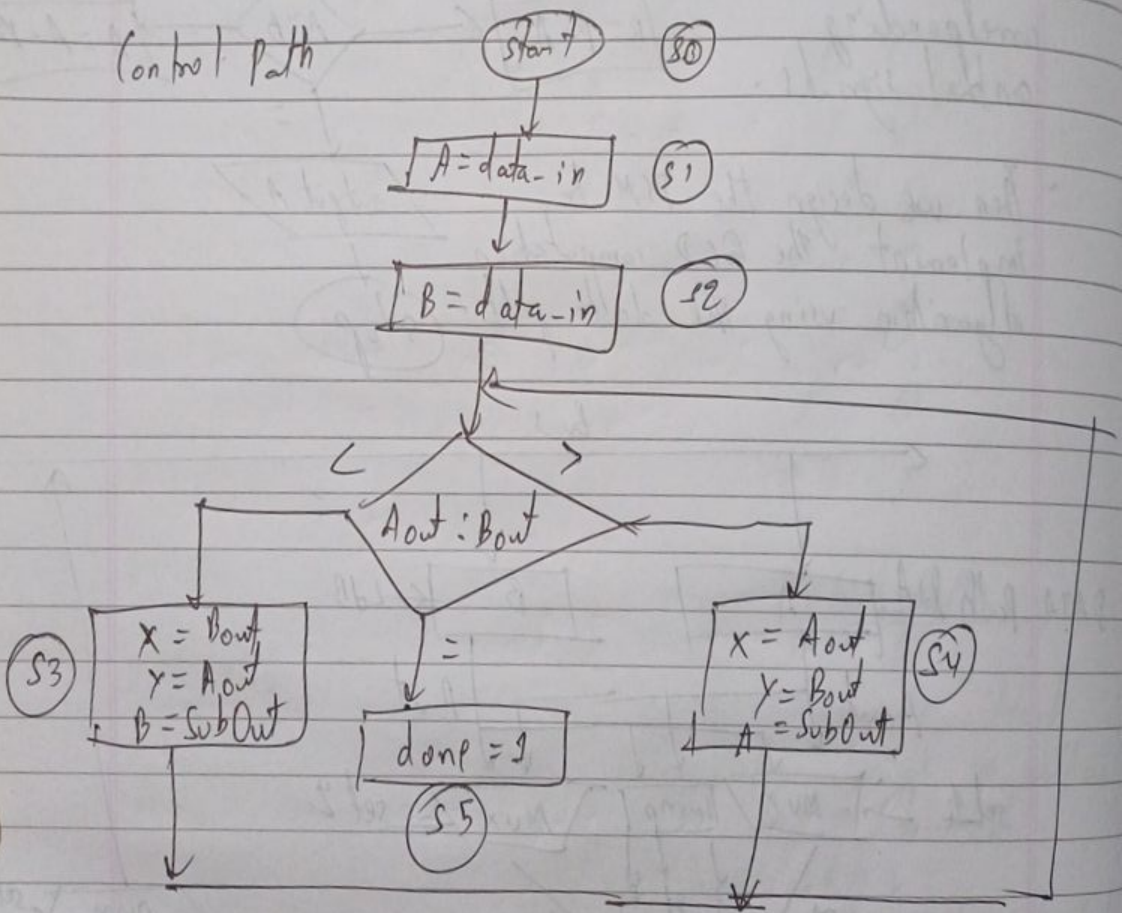
Ex2: GCD Computation

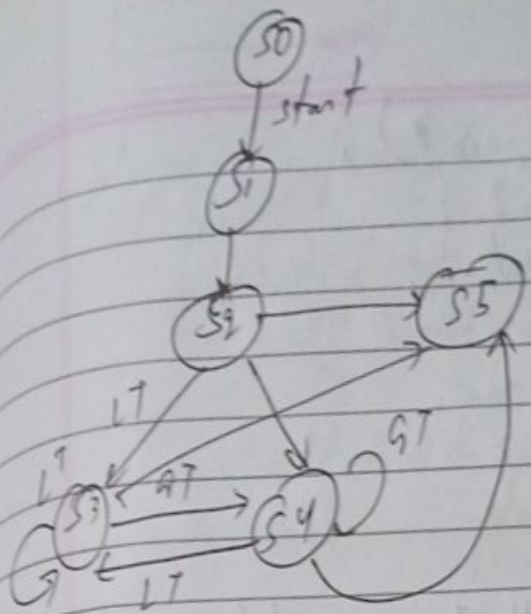
- We consider a simple algorithm using repeated subtraction.
- We identify the functional blocks required in the datapath, and the corresponding control signals.
- Then we design the FSM & implement the GCD computation algorithm using the data path.






Ld A
Ld B
Data path
sel 1
Control path
sel 2
sel_in
LT
GT
EQ
clk


Control Path
start
S0
A = data_in
S1
B = data_in
S2
Aout : Bout
<
=
>
X = Bout
Y = Aout
B = SubOut
S3
done = 1
S5
X = Aout
Y = Bout
A = SubOut
S4




```
module GCD-datapath (gt, lt, eq, ldA, ldB, sel1, sel2,
                     sel_in, data_in, clk);
input ldA, ldB, sel1, sel2, sel_in, clk;
input [15:0] data_in;
output gt, lt, eq;
wire [15:0] Aout, Bout, X, Y, Bus, SubOut;

PIPO A(Aout, Bus, ldA, clk);
PIPO B(Bout, Bus, ldB, clk);
MUX Mux_in1(X, Aout, Bout, sel1);
MUX Mux_in2(Y, Aout, Bout, sel2);
SUB SB(SubOut, X, Y);
COMPARE COMP(lt, gt, eq, Aout, Bout);
endmodule

module PIPO (data_out, data_in, load, clk);
input [15:0] data_in;
input load, clk;
output reg [15:0] data_out;
always @(posedge clk)
    if (load) data_out <= data_in;
endmodule
```

```
module SUB (out, in1, in2);
  input [15:0] in1, in2;
  output reg [15:0] out;

  always @(*)
    out = in1 - in2;
endmodule

module COMPARE (lt, gt, eq, data1, data2);
  input [15:0] data1, data2;
  output lt, gt, eq;
  assign lt = data1 < data2;
  assign gt = data1 > data2;
  assign eq = data1 == data2;
endmodule

module MUX (out, in0, in1, sel);
  input [15:0] in0, in1;
  input sel;
  output [15:0] out;

  assign out = sel ? in1 : in0;
endmodule

module controller (ldA, ldB, sel1, sel2, sel_in,
                   done, clk, lt, gt, eq, start);
  input clk, lt, gt, eq, start;
  output reg ldA, ldB, sel1, sel2, sel_in, done;
  reg [2:0] state;
  parameter S0 = 3'b000, S1 = 3'b001, S2 = 3'b010,
            S3 = 3'b011, S4 = 3'b100, S5 = 3'b101;
```

```
always @(posedge clk)
  begin
    case (state)
      S0 : if (start) state <= S1;
      S1 : state <= S2;
      S2 : #2 if (eq) state <= S5;
           else if (lt) state <= S3;
           else if (gt) state <= S4;
      S3 : #2 if (eq) state <= S5;
           else if (lt) state <= S3;
           else if (gt) state <= S4;
      S4 : #2 if (eq) state <= S5;
           else if (lt) state <= S3;
           else if (gt) state <= S4;
      S5 : state <= S5;
      default : state <= S0;
    endcase
  end
  always @ (state)
    begin
      case (state)
        S0 : begin sel_in = 1; ldA = 1; ldB = 0;
             done = 0; end
        S1 : begin sel_in = 1; ldA = 0; ldB = 1;
             end
        S2 : if (eq) done = 1;
             else if (lt) begin
                  sel1 = 1; sel2 = 0; sel_in = 0;
                  #1 ldA = 0; ldB = 1;
             else if (gt) begin
                  sel1 = 0; sel2 = 1; sel_in = 0;
                  #1 ldA = 1; ldB = 0;
             end
```

```
S3 : if (eq) done = 1;
     else if (lt) begin
                  sel1 = 1; sel2 = 0; sel_in = 0;
                  #1 ldA = 0; ldB = 1;
          else if (gt) begin
                  sel1 = 0; sel2 = 1; sel_in = 0;
                  #1 ldA = 1; ldB = 0;
          end
S4 : if (eq) done = 1;
     else if (lt) begin
                  sel1 = 1; sel2 = 0; sel_in = 0;
                  #1 ldA = 0; ldB = 1;
          end
     else if (gt) begin
                  sel1 = 0; sel2 = 1; sel_in = 0;
                  #1 ldA = 1; ldB = 0;
          end
S5 : begin
          done = 1; sel1 = 0; sel2 = 0; ldA = 0;
          ldB = 0;
     end
default : begin ldA = 0; ldB = 0; end
endcase
end
endmodule

// Testbench
module GCD_test;
     reg [15:0] data_in;
     reg clk, start;
     wire done;
```

```
reg [15:0] A, B;
gcd_datapath DP (gt, lt, eq, ldA, ldB, sel1,
                 sel2, sel_in, data_in, clk);
controller CON (ldA, ldB, sel1, sel2, sel_in, done,
                clk, lt, gt, eq, start);
initial
  begin
    clk = 1'b0;
    #3 start = 1'b1;
    #1000 $finish;
  end
always #5 clk = ~clk;
initial
  begin
    #12 data_in = 143;
    #10 data_in = 78;
  end
initial
  begin
    $monitor ($time, "%d %b", DP.Aout, done);
    $dumpfile ("gcd.vcd"); dumpvars(0, gcd_test);
  end
endmodule
```

- Modeling the control path using the alternate approach

```
module controller (ldA, ldB, sel1, sel2, sel_in, done,
                   clk, lt, gt, eq, start);
  input clk, lt, gt, eq, start;
  output reg ldA, ldB, sel1, sel2, sel_in, done;
```

```
reg [2:0] state, next_state;
parameter S0 = 3'b000, S1 = 3'b001, S2 = 3'b010,
          S3 = 3'b011, S4 = 3'b100, S5 = 3'b101;

always @(posedge clk)
  begin
    state <= next_state;
  end
always @(state)
  begin
  case (state)
  S0 : begin sel_in = 1; ldA = 1; ldB = 0;
       done = 0; end
  S1 : begin sel_in = 1; ldA = 0; ldB = 1; end
  S2 : if (eq) begin done = 1; next_state = S5;
       else if (lt) begin
              sel1 = 1; sel2 = 0; sel_in = 0;
              next_state = S3; #1 ldA = 0; ldB = 1;
            end
       else if (gt) begin
              sel1 = 0; sel2 = 1; sel_in = 0;
            #1 ldA = 1; ldB = 0;
            end
  S3 : if (eq) begin done = 1; next_state = S5; end
       else if (lt) begin
              sel1 = 1; sel2 = 0; sel_in = 0;
              next_state = S3; #1 ldA = 0; ldB = 1;
            end
       else if (gt) begin
              sel1 = 0; sel2 = 1; sel_in = 0;
            next_state = S4; #1 ldA = 1; ldB = 0;
            end
```

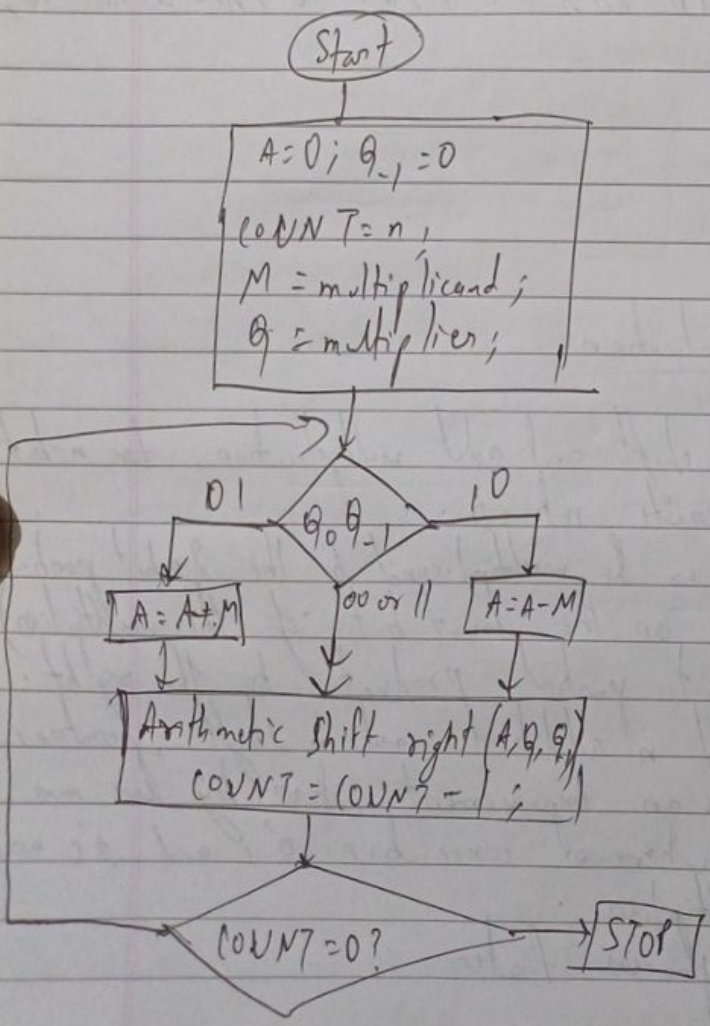
```
S4: if (eq) begin done = 1; next_state = S5; end
    else if (gt) begin
        sel1 = 1; sel2 = 0; sel_in = 0;
        next_state = S3; #1 ldA = 0; ldB = 1;
    end
    else if (lt) begin
        sel1 = 0; sel2 = 1; sel_in = 0;
        next_state = S4; #1 ldA = 1; ldB = 0;
    end
S5: begin
    done = 1; sel1 = 0; sel2 = 0; ldA = 0;
    ldB = 0; next_state = S5;
    end
default: begin ldA = 0; ldB = 0; next_state = S0;
    end
endcase
end
endmodule
```

Ex3 Booth's multiplication

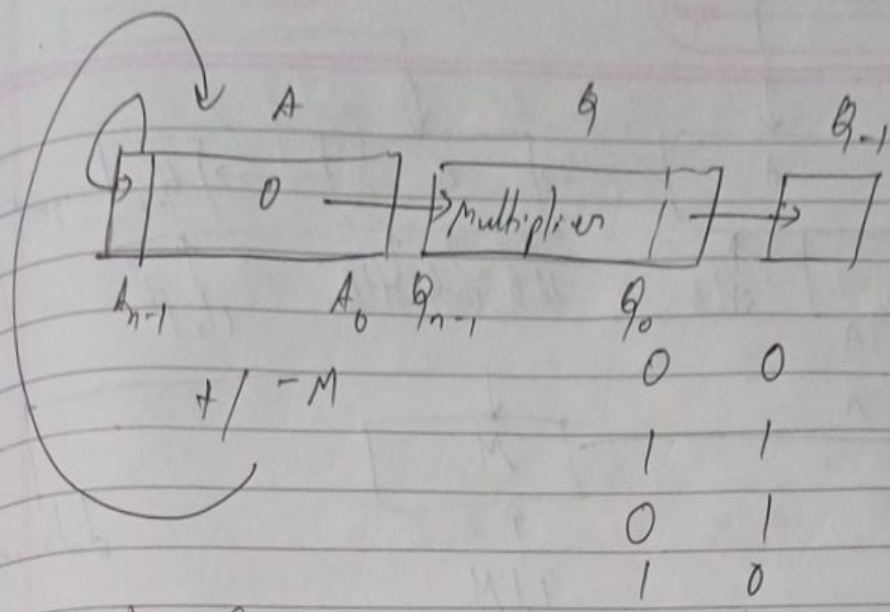
- In the conventional shift and add multiplication, for n-bit multiplication, we iterate n times.
  - Add either 0 or the multiplicand to the 2n bit partial product (depending on the next bit of the multiplier).
  - Shift the 2n-bit partial product to the right.
- Essentially we need n additions and n shift operations.
- Booth's algorithm is an improvement whereby we can avoid the additions whenever consecutive 0's and 1's are detected in the multiplier.
  - Makes the process faster

Basic idea Behind Booth's Algorithm

- We inspect two bits of the multiplier (Q_i, Q_{i-1}) at a time.
  - If the bits are same (00 or 11), we only shift the partial product.
  - If the bits are 01, we do an addition and then shift.
  - If the bits are 10, we do a subtraction and then shift.
- Q_{-1} is assumed to be equal to 0.
- Significantly reduced the number of additions / subtractions.



M : n-bit multiplicand
Q : n-bit multiplier
A : n-bit temporary register
Q_{-1} : 1-bit flip-flop



Example 1: $(-10) \times 13$

Assume 5-bit numbers.

$M: (10110)_2$

$-M: (01010)_2$

$Q: (01101)_2$

Product $= -130$

$= (1101111110)_2$

A	Q	Q-1	
00000	01101	0	initialization
01010	01101	0	A = A-M
00101	00110	1	Shift
11011	00110	1	A = A+M
11101	10011	0	Shift
00111	10011	0	A = A-M
00011	11001	1	Shift
00001	11100	1	Shift
10111	11100	1	A = A+M
11011	11110	0	Shift

Ans

Ex 2

-31×28

Assume 6-bit nos

$M: (100001)_2$

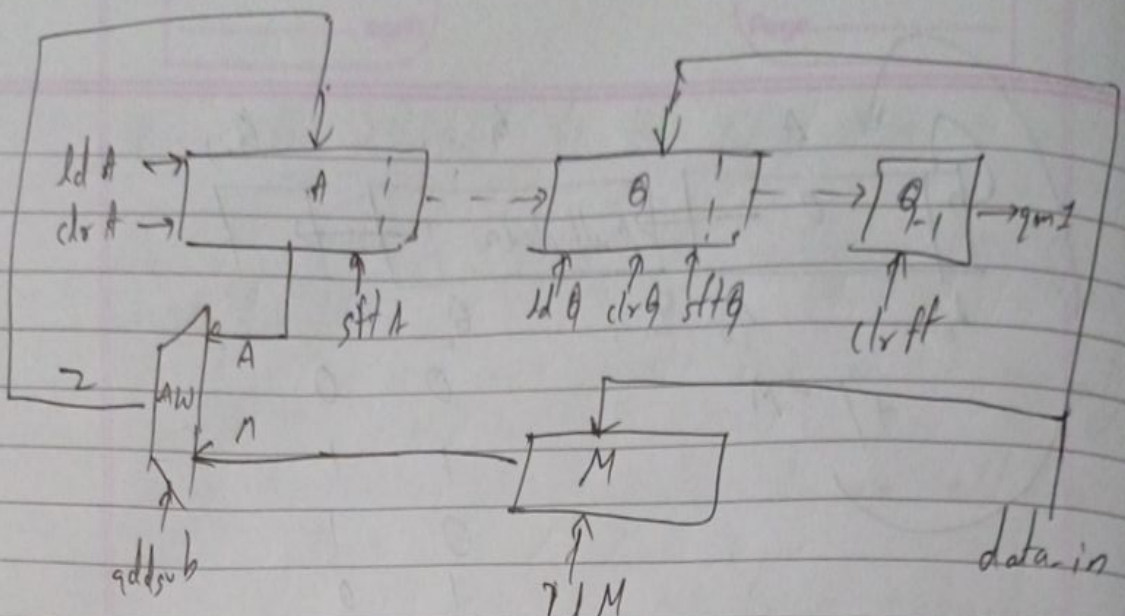
$-M: (011111)_2$

$Q: (011100)_2$

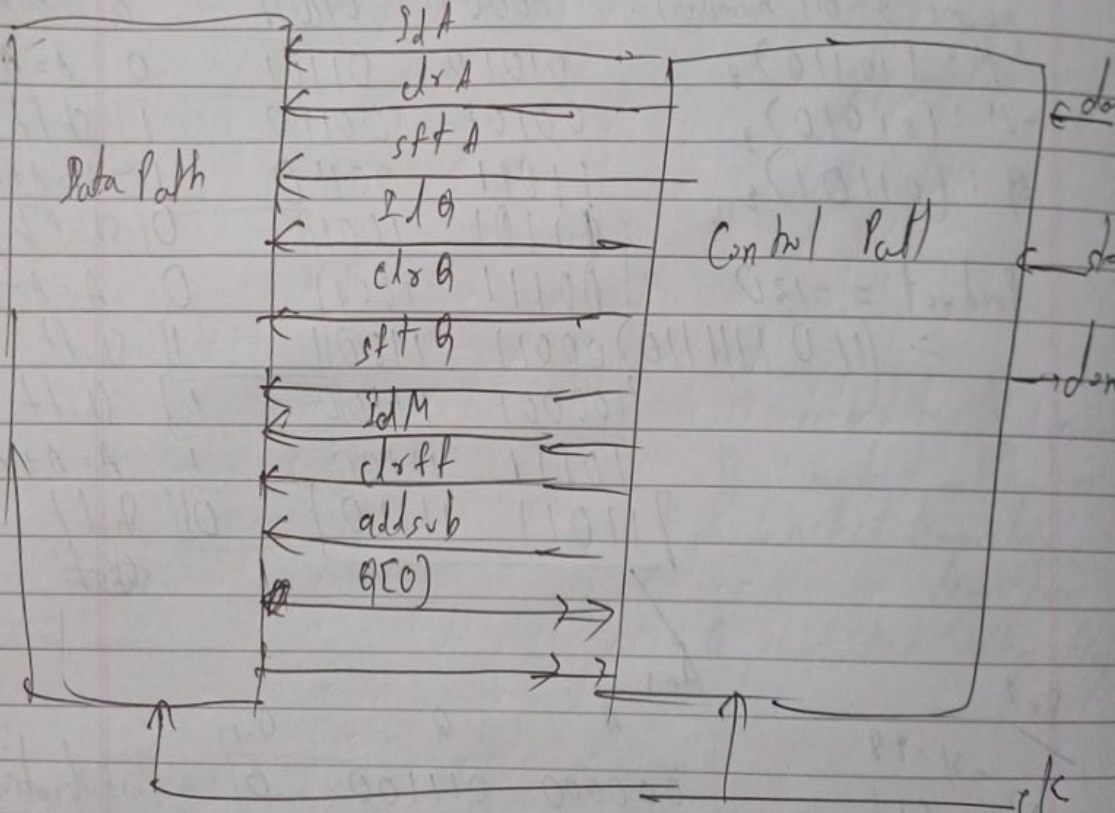
Product $= -868$

$= (1100101011100)_2$

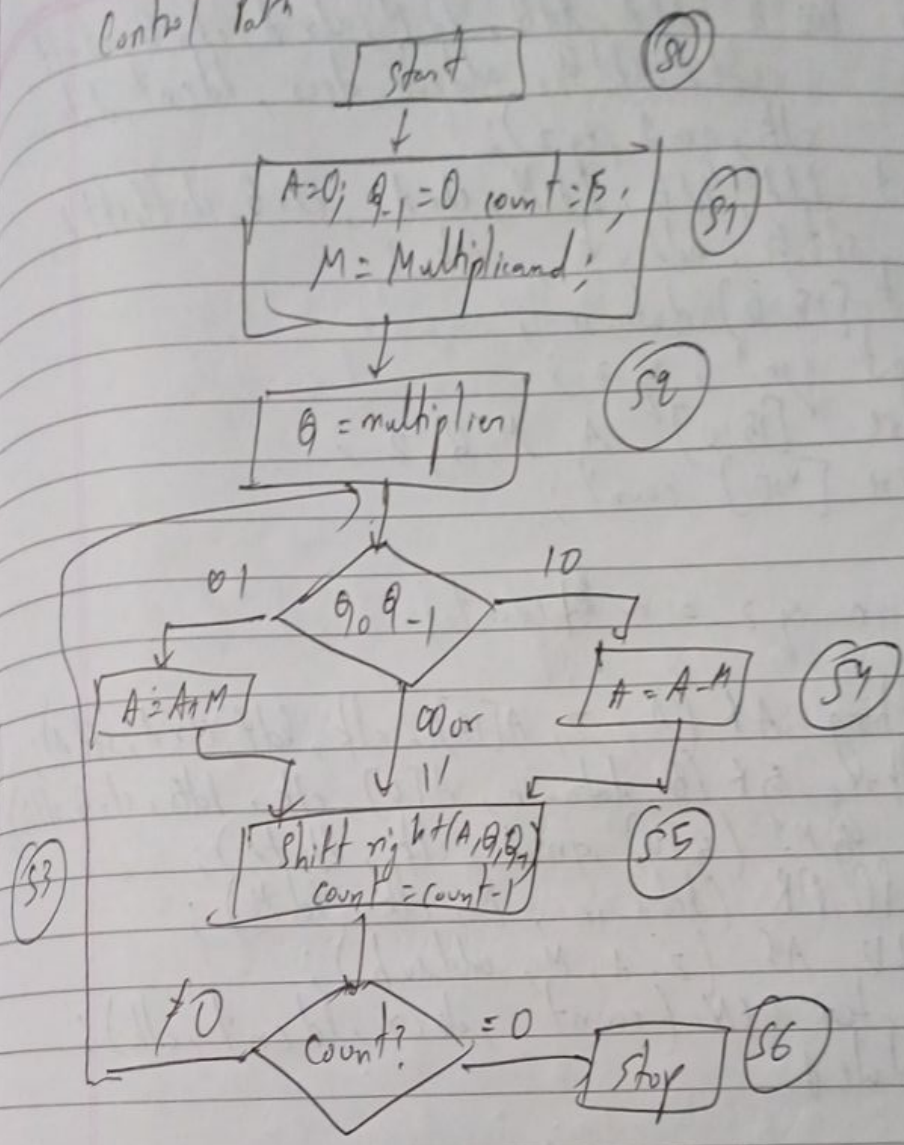
A	Q	Q-1	
000000	011100	0	initialization
000000	001110	0	Shift
000000	000111	0	Shift
011111	000111	0	A = A-M
001111	100011	1	Shift
000111	110001	1	Shift
000011	111000	1	Shift
100010	111000	1	A = A+M
110010	011100	0	Shift

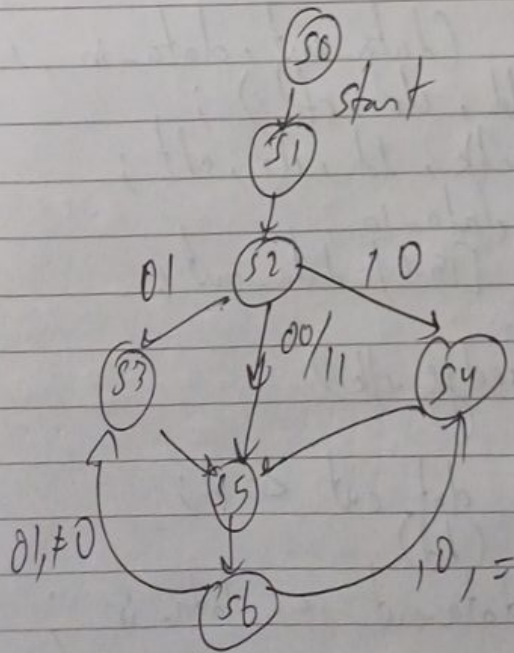

ldA
clrA
A
sftA
B
ldB
clrB
sftB
Q-1
clrff
qm1
2
AW
A
M
addsub
ldM
data_in

DATAPATH


Data Path
ldA
clrA
sftA
ldB
clrB
sftB
ldM
clrff
addsub
B[0]
Control Path
data
start
done
clk

Control Path





// The datapath

```
module BOOTH (ldA, ldQ, ldM, clrA, clrQ, clrff,
              sftA, sftQ, addsub, decr, ldcnt, data_in,
              clk, qm1, eqz);
input ldA, ldQ, ldM, clrA, clrQ, clrff, sftA,
      sftQ, addsub, clk;
input [15:0] data_in;
output qm1, eqz;
wire [15:0] A, M, Q, Z;
wire [4:0] count;

assign eqz = ~|count;

shiftreg AK (A, Z, A[15], clk, ldA, clrA, sftA);
shiftreg QK (Q, data_in, A[0], clk, ldQ, clrQ, sftQ);
dff QM1 (Q[0], qm1, clk, clrff);
PIPO MR (data_in, M, clk, ldM);
ALU AS (Z, A, M, addsub);
counter CN (count, decr, ldcnt, clk);
endmodule

module shiftreg (data_out, data_in, s_in, clk,
                 ld, clr, sft);
input s_in, clk, ld, clr, sft;
input [15:0] data_in;
output reg [15:0] data_out;

always @ (posedge clk)
  begin
    if (clr) dat_out <= 0;
    else if (ld)
        data_out <= data_in;
```

```
    else if (stf)
        dat_out <= {s_in, data_out[15:1]};
    end
endmodule

module PIPO (data_out, data_in, clk, load);
    input [15:0] data_in;
    input load, clk;
    output reg [15:0] data_out;

    always @ (posedge clk)
        if (load) data_out <= data_in;
endmodule

module dff (d, q, clk, clr);
    input d, clk, clr;
    output reg q;

    always @ (posedge clk)
        if (clr) q <= 0;
        else q <= d;
endmodule

module ALU (out, in1, in2, addsub);
    input [15:0] in1, in2;
    input addsub;
    output reg [15:0] out;
    always @ (*)
        begin
            if (addsub == 0) out = in1 - in2;
            else out = in1 + in2;
        end
endmodule
```

```
module counter (data_out, decr, ldcnt, clk)
  input decr, clk;
  output [4:0] data_out;
  always @(posedge clk)
    begin
      if (ldcnt) data_out <= 5'b10000;
      else if (decr) data_out <= data_out - 1;
    end
endmodule

module controller (ldA, clrA, sftA, ldB, clrB,
    sftB, ldM, clrFF, addsub, start, decr,
    ldcnt, done, clk, q0, qm1);
input clk, q0, qm1, start;
output reg ldA, clrA, sftA, ldB, clrB, sftB,
    ldM, clrFF, addsub, decr, ldcnt, done;

reg [2:0] state;
parameter S0 = 3'b000, S1 = 3'b001, S2 = 3'b010,
S3 = 3'b011, S4 = 3'b100, S5 = 3'b101, S6 = 3'b110;

always @(posedge clk)
  case (state)
    S0 : if (start) state <= S1;
    S1 : state <= S2;
    S2 : #2 if ({q0,qm1} == 2'b01) state <= S3;
         else if ({q0,qm1} == 2'b10) state <= S4;
         else state <= S5;
    S3 : state <= S5;
    S4 : state <= S5;
    S5 : #2 if (({q0,qm1} == 2'b01) && !eq2) state <= S3;
```

```
        else if ({q0,q-1} == 2'b01) && !eq2) state <= S4;
        else if (eq2) state <= S6;
    S6 : state <= S6;
     default : state <= S0;
    endcase
end

always @ (state)
    begin
        case (state)
            S0 : begin drA=0; ldA=0; sftA=0; drB=0;
            ldB=0; sftB=0; ldM=0; drff=0; done=0;
            end
        S1 : begin drA=1; drff=1; ldcnt=1; ldM=1; end
        S2 : begin drA=0; drff=0; ldcnt=0; ldM=0;
            ldB=0; end
        S3 : begin ldA=1; addsub=1; ldB=0; sftA=0;
            sftB=0; decr=0; end
        S4 : begin ldA=1; addsub=0; ldB=0; sftA=0;
            sftB=0; decr=0; end
        S5 : begin sftA=1; sftB=1; ldM=0; ldB=0;
            decr=1; end
        S6 : done=1;
        default : begin drA=0; sftA=0; ldB=0;
            sftB=0; end
        endcase
    end
endmodule
```

Modeling Memory

How to Model Memory?

- Memory is typically included by instantiating a pre-designed module from a design library.
- Alternatively, we can model memories using two-dimensional arrays.

```
module memory_model (...);
  ...
    reg [7:0] mem [0:1023];
  ...
endmodule
```

→ each memory word is of type [7:0], i.e 8 bits.

- The memory words can be accessed as mem[0], mem[2], ... mem[1023].

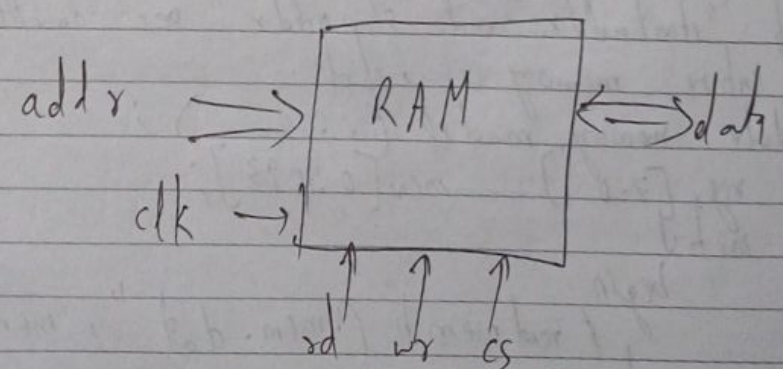
How to initialize Memory?

- By reading memory data patterns from a specified disk file.
  - Used for simulation
  - Used for test benches
- Two verilog functions can be used:
  - $readmemb (filename, memname, startaddr, stopaddr);
    (Data is read in binary format)
  - $readmemh (filename, memname, startaddr, stopaddr);
    (Data is read in hexadecimal format)
  - if "startaddr" and "stopaddr" are omitted, the entire memory is read.

```
module memory_model (...);
  reg [7:0] mem [0:1023];
  initial
    begin
      $readmemh ("mem.dat", mem);
    end
endmodule
```

eg: module memory_model (...);
reg [7:0] mem [0:1093];
initial
begin
$readmemb ("mem.dat", mem, 300, 500);
end
endmodule

ex3:
//Single-port RAM with synchronous read/write (chip select)
module ram_1 (addr, data, clk, rd, wr, cs);
input [9:0] addr;
inout [7:0] data;
reg [7:0] mem [1093:0]; 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



Ex 4 // Single-port RAM with asynchronous read/write

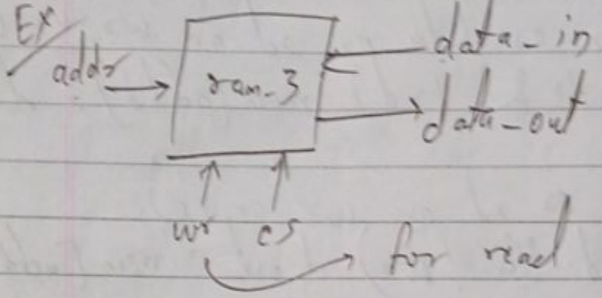
```
module ram_2 (addr, data, rd, wr, cs);
  input [9:0] addr; input rd, wr, cs;
  inout [7:0] data;
  reg [7:0] mem [1023:0]; reg [7:0] d_out;

  assign data = (cs && rd) ? d_out : 8'bz;

  always @ (addr or data or rd or wr or cs)
    if (cs && wr && !rd) mem[addr] = data;

  always @ (addr or rd or wr or cs)
    if (cs && rd && !wr) d_out = mem[addr];

endmodule
```

Ex 5 A ROM / EPROM (Erasable programmable ROM)

```
module rom (addr, data, rd_en, cs);
  input [7:0] addr; input rd_en, cs;
  output reg [7:0] data;

  always @ (addr or rd_en or cs)
    case (addr)
      0 : data = 22;
      1 : data = 45;
      . . . .
      7 : data = 12;
    endcase
endmodule
```

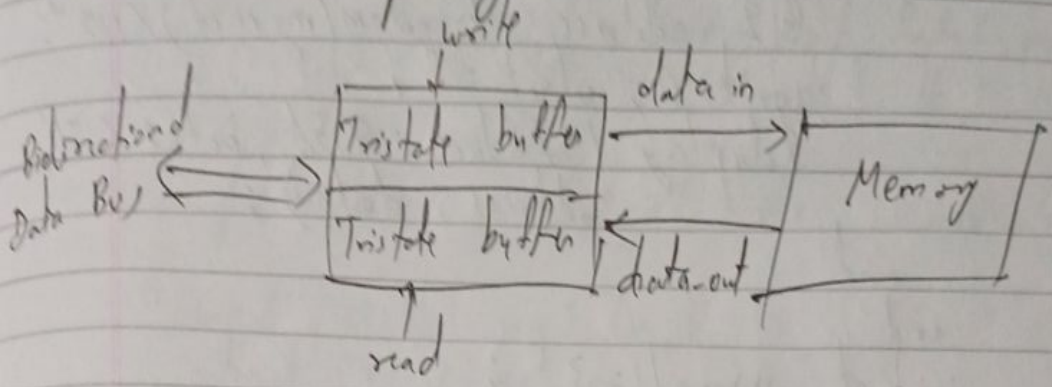
- Some simulator or synthesis tools given inconsistent behaviour when using the "inout" data type.
  - Such "inout" bidirectional data should be avoided.
- A better way to design a memory unit is to keep the data input and data output bus signal lines separate.

Ex




```
module ram_3 (data_out, data_in, addr, wr, cs);
  parameter add_size = 10, word_size = 8,
            memory_size = 1094;
  input [add_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.

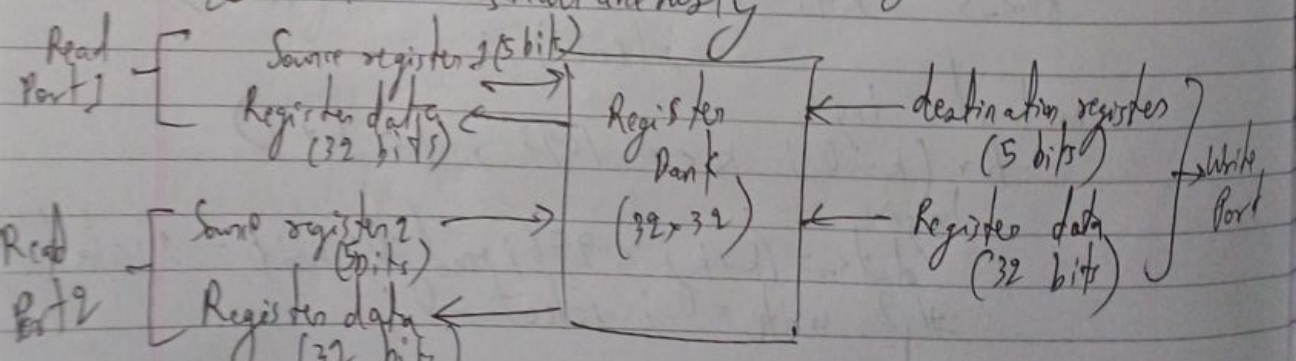



```
tri [7:0] Bus;
wire [7:0] data_out, data_in;
assign Bus = read ? data_out : 8'hzz;
assign data_in = write ? Bus : 8'hzz;

// testbench for ram_3

module RAM_test;
  reg [9:0] address;
  wire [7:0] data_out;
  reg [7:0] data_in;
  reg write, select;
  integer k, myseed;

  ram_3 RAM (data_out, data_in, address, write,
             select);
initial begin
  for (k=0; k<=1023; k=k+1)
  begin
    data_in = (k+k) % 256; read = 0; write = 1; select = 1;
    #2 write = 0; select = 0;
  end
```

```
repeat (20)
  begin
    #2 address = $random(myseed) % 1024;
    write = 0; select = 1;
    $display ("Address : %5d, Data = %4d",
              address, data_in);
  end
end
initial myseed = 35;
endmodule
```

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-accessible registers.
  - For ex, in MIPS 32 processor, there are 32 32-bit registers, referred to as R0, R1 . . R31.
- Can be implemented as independent or an array.
- Register banks often allow concurrent accesses.
  - MIPS2 allows 2 register reads & 1 register write every clock cycle.
- It is assumed that the same register is not read & written simultaneously



```
// 32 x 32 register file
module regbank_v3 (rdData1, rdData2, wrData,
        sr1, sr2, dr, write, clk);
  input clk, write;
  input [4:0] sr1, sr2, dr; // source & 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

// 32 x 32 register file with reset facility
module regbank_v4 (rdData1, rdData2, wrData, sr1,
        sr2, dr, write, reset, clk);
  input clk, write, reset;
  input [4:0] sr1, sr2, dr;
  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)
  begin
    if (reset) begin
      for (k=0; k<32; k=k+1) begin
        regfile[k] <= 0;
      end
    end
    else if begin
        if (write)
          regfile [dr] <= wrData;
        end
    end
endmodule

// testbench
module regfile_test;
    reg [4:0] sr1, sr2, dr;
    reg [31:0] wrData;
    reg write, reset, clk;
    wire [31:0] rdData1, rdData2;
    integer k;

    regbank_v4 REG (rdData1, rdData2, wrData,
        sr1, sr2, dr, write, reset, clk);
    initial clk = 0;
    always #5 clk = !clk;
    initial
      begin
        $dumpfile("regfile.vcd"); $dumpvars(0, regfile_test);
        #1 reset = 1; write = 0;
        #5 reset = 0;
    end
```

```
initial
  begin
    #7
    for (k=0; k< 32 ; k=k+1)
      begin
        dr = k; wrData = 10*k; wrik = 1;
        #10 wrik = 0;
      end
    #20
    for (k=0; k<32; k=k+2)
      begin
        sr1 = k; sr2 = k+1;
        #5;
        $display ("rg[%2d] = %d, rg[%2d]=%d",
          sr1, rdData1, sr2, rdData2);
      end
    #900 $finish;
  end
endmodule
```

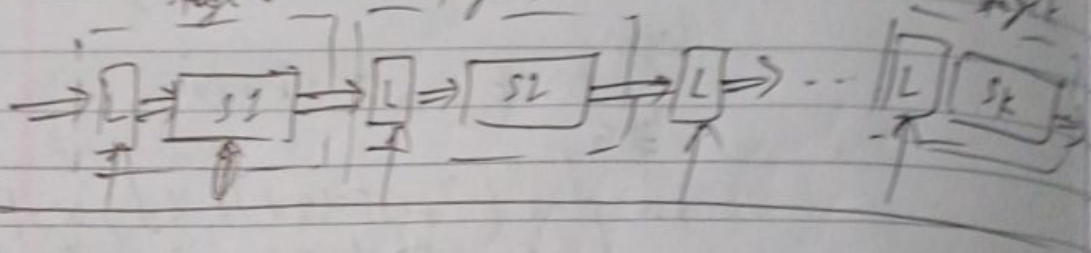
Pipeling

- A mechanism for overlapped execution of several inputs sets, by partitioning some computation into a set of k-sub computations (or stages)
  - Very significant speed up (generally k).
- Suppose we want to attain k times speed up for some computation.
  - Alternative 1: Replicate the hardware k times → cost also goes up k times.

- Alternative 2: Split the comp. flow into k stages
  → delay remains same but cost increases

- Need for buffering
  - in hardware pipeline, we need a latch between successive stages to hold the intermediate results temporarily.

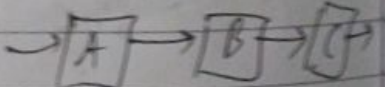
Model of a Synchronous k-stage Pipeline



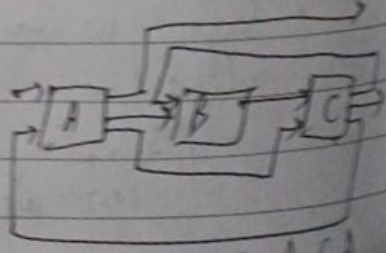
- The latches are made with master-slave flip flops, and serve the purpose of isolating input from outputs.
- The pipeline stages are typically combinational circuits.
- When CBck is applied, all latches transfer data to the next stage simultaneously.

Structure of the Pipeline

- Linear Pipeline: The stages that constitute the pipeline are executed one by one in sequence



- Non-linear Pipeline: The stages may not execute in a linear sequence

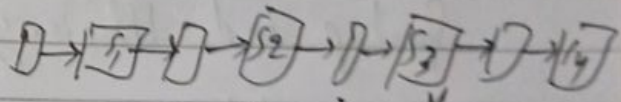


A possible sequence: A, B, A, B, C, A, C

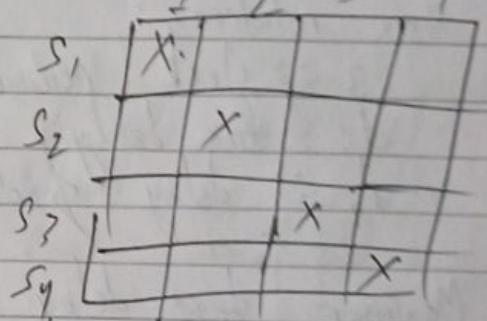
Reservation Table

- The Reservation Table is a data structure that represents the utilization pattern of successive stages in a synchronous pipeline.

- Basically a space-time diagram of the pipeline that shows precedence relationship among pipeline stages
  - X-axis shows the time steps
  - Y-axis shows stages

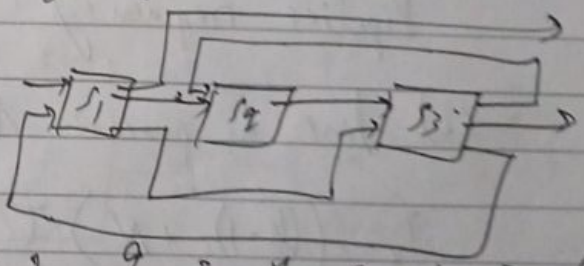


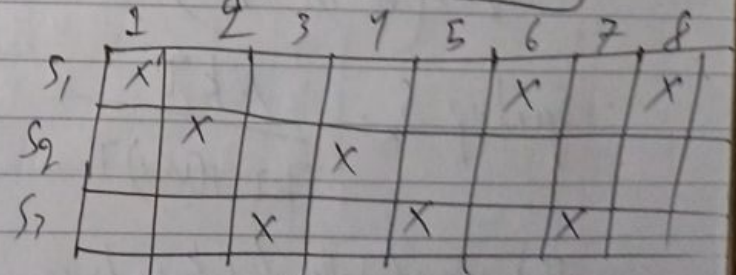
- Number of columns give evaluation time
- The reservation table for a 4-stage linear pipeline is shown

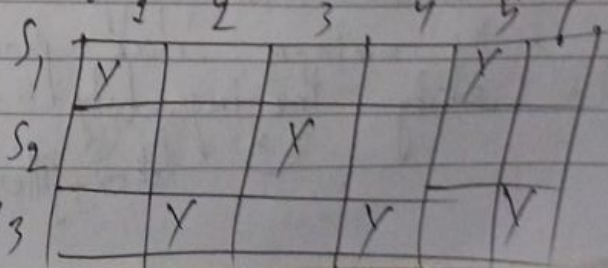


- Reservation table for a 3-stage dynamic multi-function pipeline is shown

- Contains feedforward and feedback connections.
- Two functions X and Y







- Some characteristics:
  - Multiple x's in a row :: repeated used of the same stage in different cycles.
  - Consecutive x's in a row :: extended use of a stage over more, that one cycles.
  - Multiple x's in a column :: multiple stages are used in parallel during a clock cycle.

Speedup and Efficiency

Some notations:
- τ :: clock period of the pipeline
- t_i :: time delay of the circuitry in stage S_i
- d_L :: delay of a latch

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

$\tau = \tau_m + d_L$

Pipeline frequency $f = 1/\tau$

- Total time to process N data sets is given by

$$T_k = \left((k-1) + N\right)\tau$$

- Speedup $S_k = \dfrac{Nk\tau}{k\tau + (N-1)\tau} = \dfrac{Nk}{k+N-1}$ as $N \to \infty$, $S_k \to k$

Clock skew / Jitter / Setup time

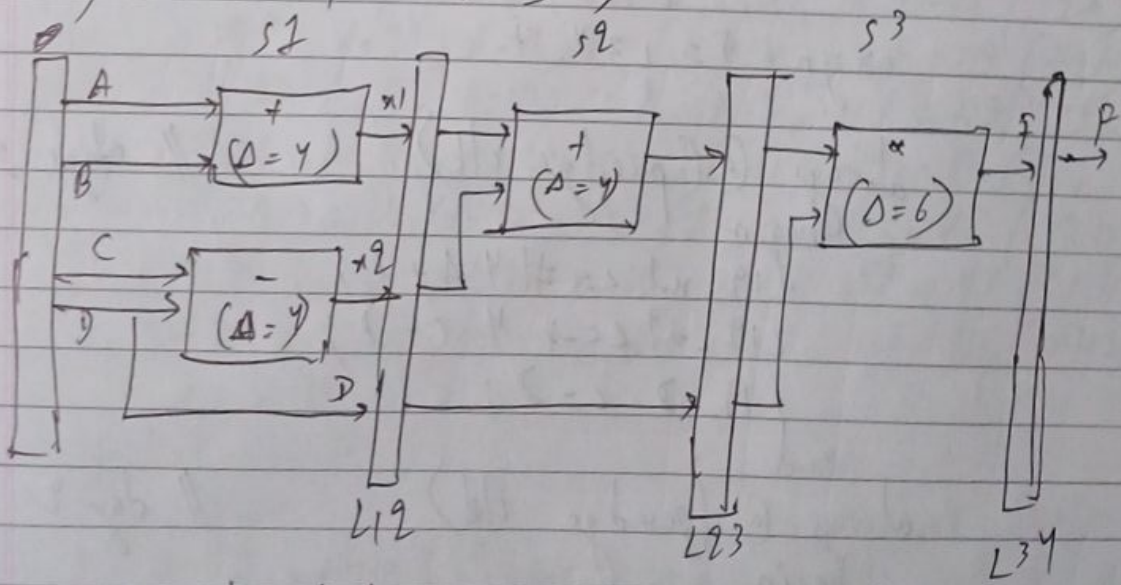
- The minimum clock period of the pipeline must satisfy the inequality:

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

- Definitions:
  - skew: Maximum delay difference b/w the arrival of clock signals at the stage latches.
  - Jitter: Maximum delay difference b/w the arrival of clock signal at the same latch.
  - Logic delay: Maximum delay of the slowest stage in the pipeline.
  - Set up time: Minimum time a signal needs to be stable at the input of a latch before it can be capture.

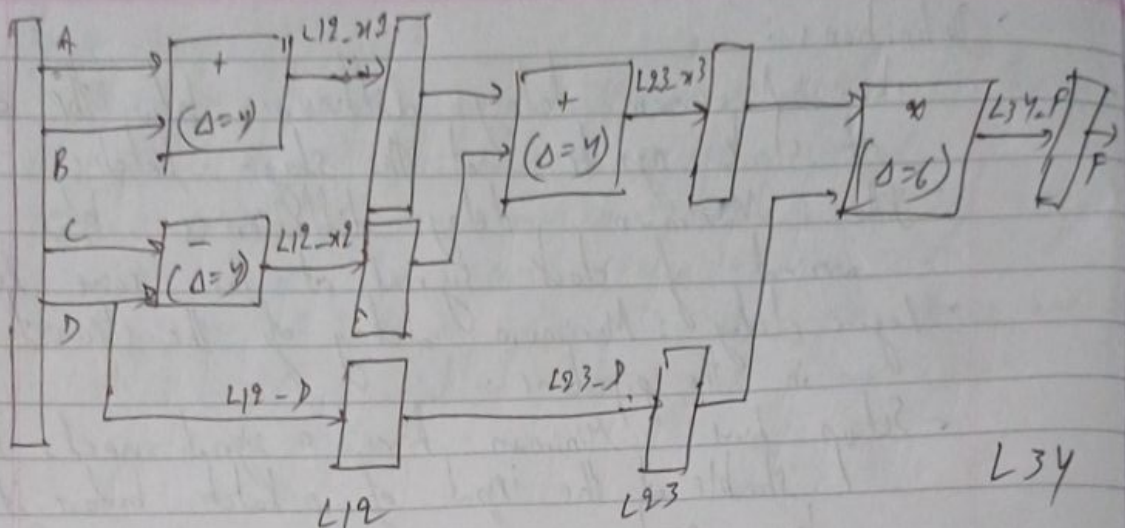
Ex: A simple example

- We consider example of a very simple 3-stage pipeline.
  - Four N-bit unsigned integers A, B, C, and D as inputs
  - An N-bit unsigned integer F as output.
  - The following computations are carried out in stages:
    a) S1: x1 = A + B; x2 = C - D;
    b) S2: x3 = x1 + x2;
    c) S3: F = x3 * D;



All the intermediate variables are allocated storage in the inter-stage latches itself.

S1 S2 S3



• One stage per "always" block

```
module pipe_ex (F, A, B, C, D, clk);
  parameter N = 10;
  input [N-1:0] A, B, C, D;
  input clk;
  output [N-1:0] F;
  reg [N-1:0] L12_x1, L12_x2, L12_D, L23_x3, L23_D,
              L34_F;
  assign F = L34_F

  always @(posedge clk)                    // stage 1
    begin
      L12_x1 <= #4 A + B;
      L12_x2 <= #4 C - 1;
      L12_D <= D;
    end
  always @(posedge clk)                    // stage 2
    begin
      L23_x3 <= #4 L12_x1 + L12_x2;
      L23_D <= L12_D;
    end
```

```
always @ (posedge clk)          // stage 3
    L34_F <= #6 L23_x3 + L23_D;
endmodule

// Testbench
module pipe1_test;
    parameter N=10;
    wire [N-1:0] F;
    reg [N-1:0] A, B, C, D;
    reg clk;

    pipe_ex MYPIPE (F, A, B, C, D, clk);

    initial clk=0
    always #10 clk = ~clk;
    initial
    begin
        #5 A=10; B=8; C=6; D=3;   // F=75 (4Bh)
        #20 A=10; B=10; C=5; D=3; // F=66 (42h)
        #90 A=20; B=11; C=1; D=4; // F=112 (70h)
        #20 A=5; B=10; C=8; D=2;  // F=62 (3Eh)
        #20 A=8; B=15; C=5; D=0;  // F=0 (00h)
        #20 A=10; B=20; C=5; D=3; // F=66 (42h)
        #90 A=10; B=10; C=30; D=1; // F=49 (31h)
        #20 A=30; b=1; c=2; D=4;  // F=116 (74h)
    end
    initial
    begin
        $dumpfile ("pipe1.vcd");
        $dumpvars (0, pipe1_test);
        $monitor ("Time: %d, F=%d", $time, F);
        #300 $finish
    end
endmodule
```

A warning

- In this example, we have used a single phase clock to load all the inter-stage registers in the pipeline.
- In a real pipeline, this may lead to race conditions.
- Possible solutions:
  - Use master-slave flip-flops in the registers.
  - Use a two-phase clock to clock the alternate stages.

A more complex Example

- Consider a pipeline that carries out the following stage-wise operations:
  - Inputs: Three register addresses (rs1, rs2, and rd), an ALU function (func), and a memory address (addr).
  - Stage 1: Read two 16-bit numbers from the registers specified by "rs1" and "rs2", and store them in A and B.
  - Stage 2: Perform an ALU operation on A and B specified by "func", and store it in Z.
  - Stage 3: Write the value of Z in the register specified by "rd".
  - Stage 4: Also write the value of Z in memory location "addr".

The Assumptions :-

- There is a register bank containing 16 16-bit registers.
  - 4 bits are required to specify a register address.
  - 2 register reads and 1 register write can be

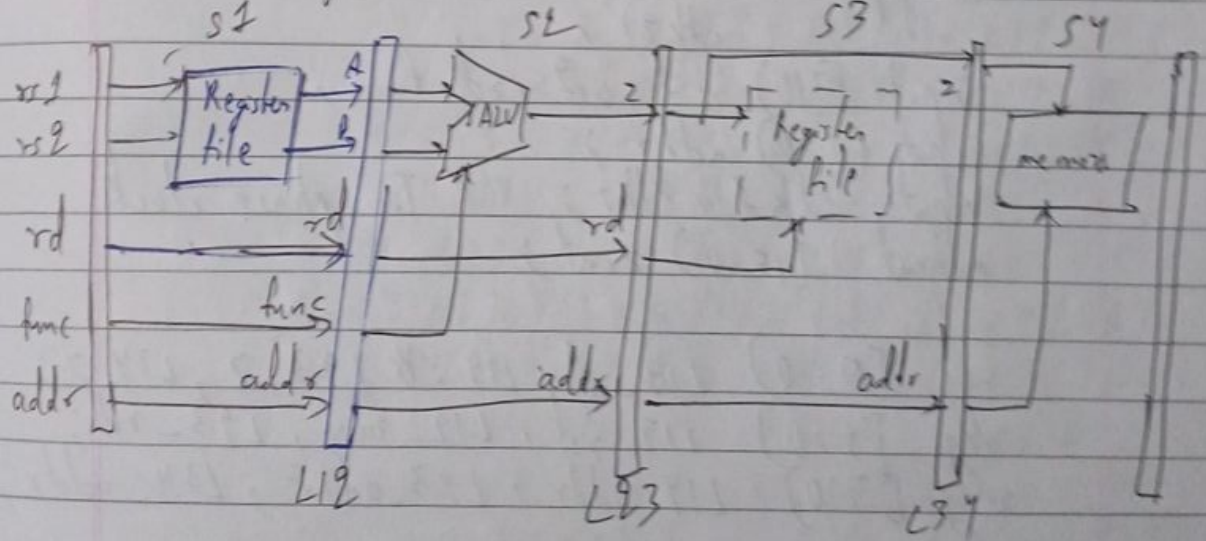
performed every clock cycle.

- Register addresses are "rs1", "rs2" and "rd".

- Assume that the memory is organized as 256 × 16.
  - 8 bits are required to specify memory addresses.
  - Every memory location contains 16 bits of data, which can be read in a single clock cycle.
  - Memory address specified by "addr".

- The ALU function is selected by a 4-bit field "func" as follows:

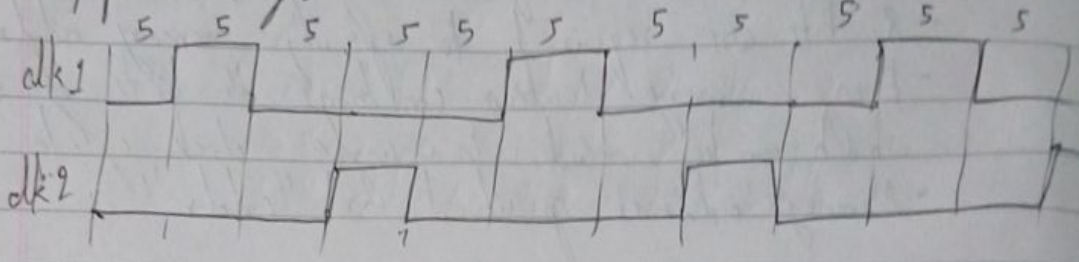
0000 : ADD	0001 : SUB	0010 : MUL
0011 : SELA	0100 : SELB	0101 : AND
0110 : OR	0111 : XOR	1000 : NEGA
1001 : NEGB	1010 : SRA	1011 : SLA
output is A	shift right	shift left

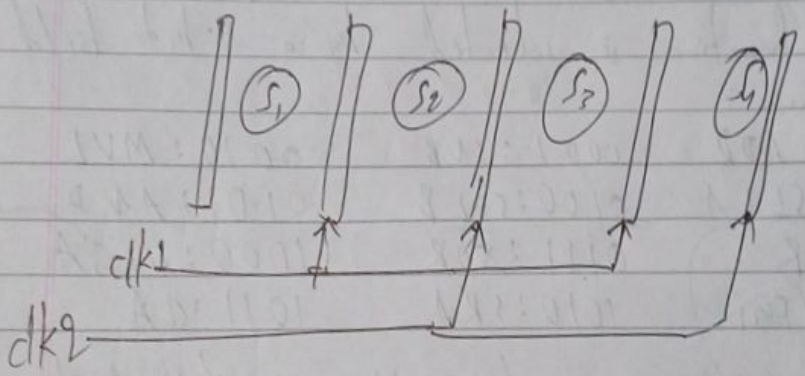


Clocking issue in Pipeline

- It is important that the consecutive stages be applied suitable clocks for correct operation.

- Two options:
  a) Use master/slave flip-flops in the latches to avoid race condition.
  b) Use non-overlapping two phase clock for the consecutive pipeline stages.






```
module pipe_ex2 (zout, rs1, rs2, rd, func, addr, clk1,
                 clk2);
  input [3:0] rs1, rs2, rd, func;
  input [7:0] addr;
  input clk1, clk2; // Two phase clock
  output [15:0] zout;

  reg [15:0] L12_A, L12_B, L23_2, L34_2;
  reg [3:0] L12_rd, L12_func, L23_rd;
  reg [7:0] L12_addr, L23_addr, L34_addr;

  reg [15:0] regbank [0:15]; // Register Bank
  reg [15:0] mem [0:255]; // 256 x 16 memory
```

```
assign Zout = L34_Z;

always @ (posedge clk1)    // stage 1
begin
    regbank[L23_rd] <= #2 L23_Z;
    L34_Z <= #2 L23_Z;
    L34_addr <= #2 L23_addr;
end
begin
    L12_A <= #2 regbank[rs1];
    L12_B <= #2 regbank[rs2];
    L12_rd <= #2 rd;
    L12_func <= #2 func;
    L12_addr <= #2 addr;
end
always @ (negedge clk2)    // stage 2
begin
    case (func)
        0 : L23_Z <= #2 L12_A + L12_B;
        1 : L23_Z <= #2 L12_A - L12_B;
        2 : L23_Z <= #2 L12_A * L12_B;
        3 : L23_Z <= #2 L12_A;
        4 : L23_Z <= #2 L12_B;
        5 : L23_Z <= #2 L12_A & L12_B;
        6 : L23_Z <= #2 L12_A | L12_B;
        7 : L23_Z <= #2 L12_A ^ L12_B;
        8 : L23_Z <= #2 ~ L12_A;
        9 : L23_Z <= #2 ~ L12_B;
        10 : L23_Z <= #2 L12_A >> 1;
        11 : L23_Z <= #2 L12_A << 1;
    default : L23_Z <= #2 16'hxxxx;
end case
```

```
        L23_rd <= #2 L12_rd;
        L23_addr <= #2 L12_addr;
    end

    always @(posedge clk1)          // stage 3
        begin
            regbank[L23_rd] <= #2 L23_z;
            L34_z <= #2 L23_z;
            L34_addr <= #2 L23_addr;

    always @(negedge clk2)          // stage 4
        begin
            mem[L34_addr] <= #2 L34_z;
        end
endmodule

// testbench
module pipe2_test;
    wire [15:0] z;
    reg [3:0] rs1, rs2, rd, func;
    reg [7:0] addr;
    reg clk1, clk2;
    integer k;
    pipe_ex2 MYPIPE(z, rs1, rs2, rd, func, addr,
                    clk1, clk2);
    initial
        begin
            clk1 = 0; clk2 = 0;
            repeat (20)
                begin
                    #5 clk1 = 1; #5 clk1 = 0;
```

```
        #5 clk2 = 1; #5 clk2 = 0;
    end
end
initial
    for (k=0; k<16; k=k+1)
        MYPIPE.regbank [k] = k; // initialize registers
initial
begin
    #5 rs1=3; rs2=5; rd=10; func=0; addr=125; // ADD
    #20 rs1=3; rs2=8; rd=11; func=2; addr=126; // MUL
    #20 rs1=10; rs2=5; rd=14; func=1; addr=128; // SUB
    #20 rs1=7; rs2=3; rd=13; func=11; addr=129; // SLA
    #20 rs1=10; rs2=5; rd=15; func=2; addr=129; // SUB
    #20 rs1=11; rs2=13; rd=16; func=0; addr=130; // ADD

    #60 for (k=125; k<131; k=k+1)
        $display ("Mem [%3d] = %3d", k, MYPIPE.mem[k]);
end

initial
    begin
        $dumpfile ("pipe2.vcd");
        $dumpvars (0, pipe2_test);
        $monitor ("Time = %3d, F=%3d", $time, 2);
        #300 $finish
    end
endmodule
```

Pipeline implementation of a processor

- We shall first look at instruct, set architecture of a program Reduced instruction set architecture (RISC) i.e MIPS 32.

MIPS 32

- MIPS registers:
  a) 32, 32-bit general purpose registers (GPRs), R0 to R31.
     - Register R0 contains a constant 0, cannot be written
  b) A special-purpose 32-bit program counter (PC).
     - Points to the next instruction in memory to be fetched and executed.
- No flag registers (zero, carry, sign, etc).
- Very few addressing modes (register, immediate, register indexed, etc).
  - Only load and store instructions can access memory.
- We assume memory word size is 32-bits (word addressable).

Instruction Subset being considered

- Load & Store instructions

```
LW  R2, 124(R8)     // R2 = Mem[R8 + 124]
SW  R5, -10(R25)    // Mem[R25 - 10] = R5
```

- Arithmetic & Logic instructions

```
ADD  R1, R2, R3     // R1 = R2 + R3
ADD  R1, R2, R0     // R1 = R2 + 0
SUB  R12, R10, R8   // R12 = R10 - R8
AND  R20, R1, R5    // R20 = R1 & R5
OR   R11, R5, R6    // R11 = R5 | R6
MUL  R5, R6, R7     // R5 = R6 * R7
SLT  R5, R11, R12   // If R11 < R12, R5 = 1; else R5 = 0
```

```
ADDI  R1, R2, 25    // R1 = R2 + 25
SUBI  R5, R1, 150   // R5 = R1 - 150
SLTI  R2, R10, 10   // if R10 < 10, R2 = 1; else R2 = 0
```

- Branch instructions

```
BEQZ  R1, Loop      // Branch to Loop if R1 = 0
BNEQ  R2 R5, Label  // Branch to Label if R5 != 0
```

- Jump Instruction

```
J  Loop             // Branch to Loop unconditionally
```

- Miscellaneous instruction

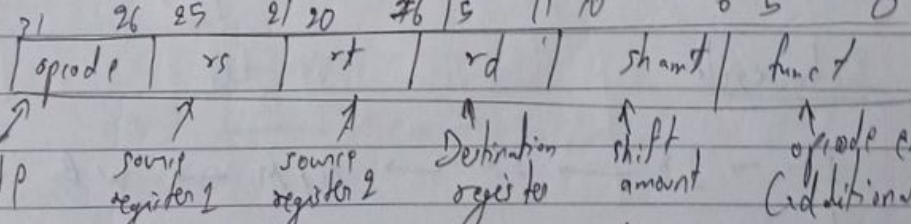
```
HLT                 // Halt execution
```

MIPS instruction encoding

- Instructions are of 3 type - r, i, j

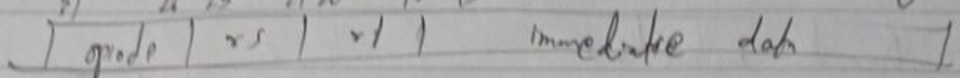
a) r-type instruction

- Two source and one destination
- In addition, for shift instructions, the number of bits to shift can also be specified (we are not considering such instructions here)



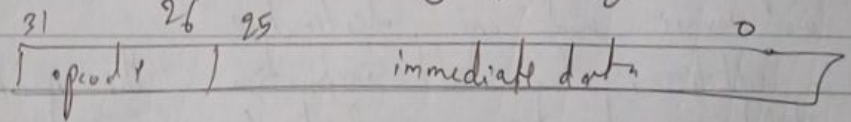
	opcode
ADD :	000000
SUB :	000001
AND :	000010
OR :	000011
SLT :	000100
MUL :	000101
HLT :	111111

b) I-type instruction Encoding
- contains 16-bit immediate data field
- supports one source and one destination register



LW : 001000
SW : 001001
ADDI : 001010
SUBI : 001011
SLTI : ~~001100~~ 001100
~~SLTI~~ SNEQ 2 : 001101
BEQ2 : 001110

c) J-type instruction
- contains a 26-bit jump address field
- extended to 28 bit by padding two 0's on right



J : 010000

Instru cycle

IF → ID → EX → MEM → WB

IR ← mem[PC];
NPC ← PC(+1;
(next PC)

ID

A ← Reg[rs];
B ← Reg[rt];
Imm ← (IR15)^16 ## IR15...0 // sign extension
Imm1 ← (IR25)^6 ## IR25...0 // sign extension

EX → ALU operation.

Mem reference

ALUOut ← A + Imm;

Reg-Reg

ALUOut ← A func B;

Reg-imm

ALUOut ← A fun Imm;

Branch

ALUOut ← NPC + Imm;
cond ← (A op 0);

ex BEQZ R2, Label
(op is ==)

MEM

Load inst.

PC ← NPC;
(Load mem data) LMD ← Mem[ALUOut];

store

PC ← NPC;
Mem[ALUOut] ← B;

other inst

PC ← NPC;

Branch

if (cond) PC ← ALUOut;
else PC ← NPC;

WB

Reg-Reg

Reg[rd] ← ALUOut

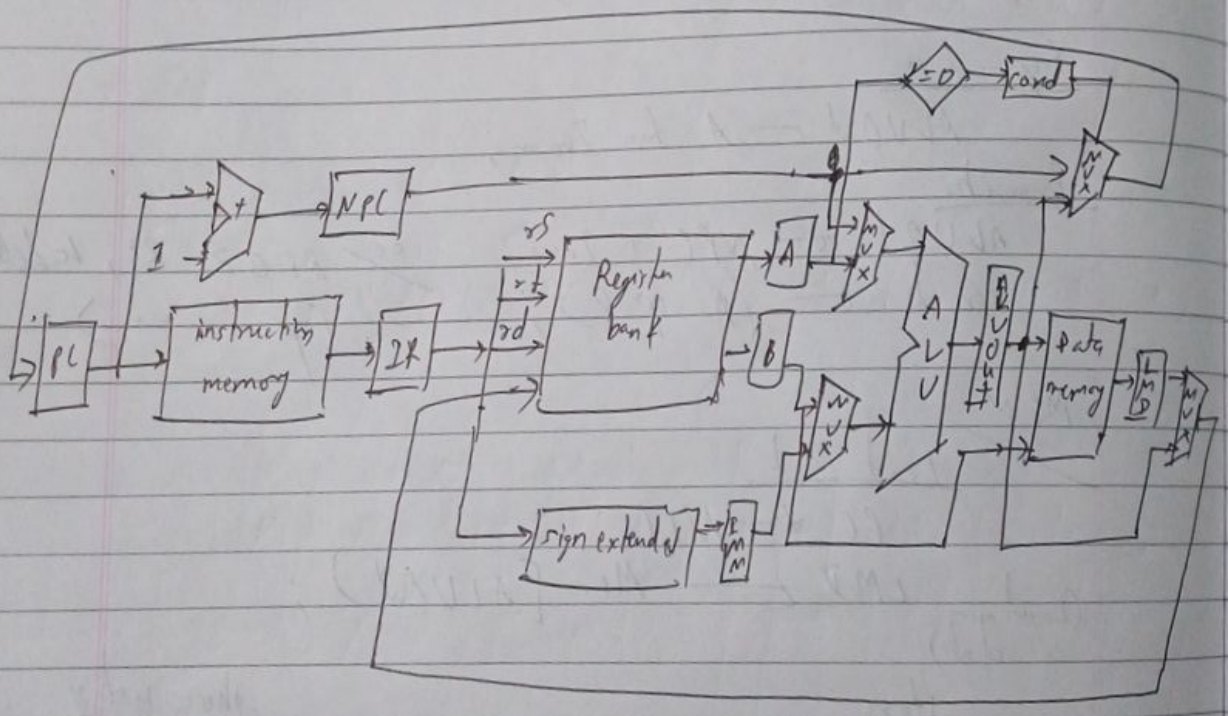
Reg-imm

Reg[rt] ← ALUOut;

Load instr

Reg[rt] ← LMD;

Data Path
(non-pipelined)

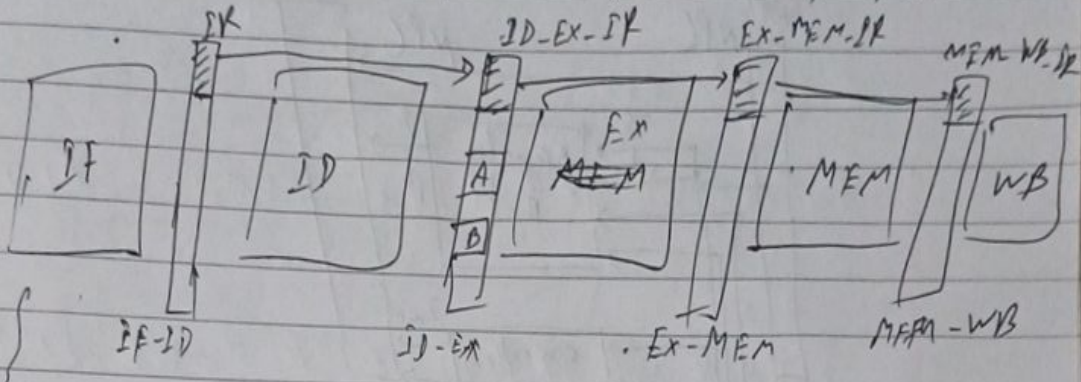


Pipelined

- We should be able to start a new instr in every clock cycle
- Each stage should be completed within 1 time period.

· Convention used

· The latches → IF-ID, ID-EX, EX-MEM, MEM-WB

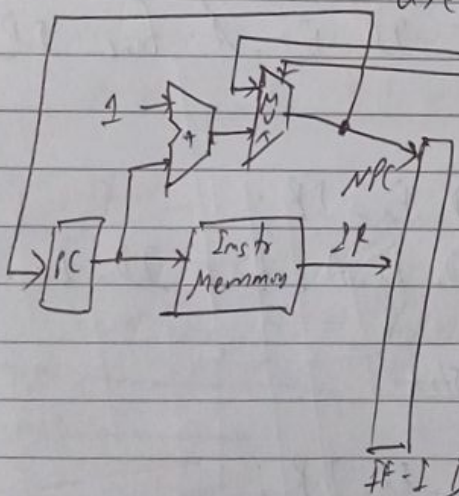


IR ← Mem[PC]

a) Micro operations for stage IF

IF_ID_IR ← Mem[PC]

IF_ID_NPC, PC ← (if ((EX_MEM_IR[Opcode] == branch) & EX_MEM_cond) {EX_MEM_ALUOut} else {PC+1});



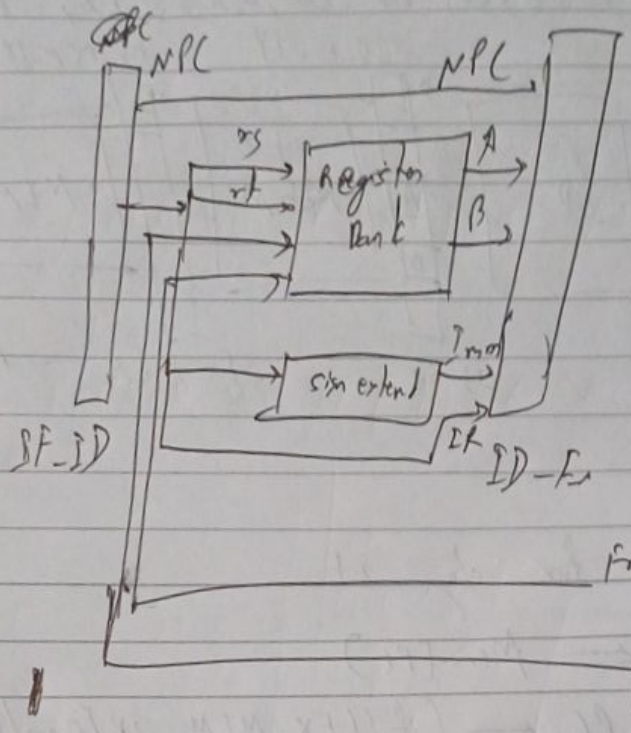
b) ID

ID_EX_A ← Reg[IF_ID_IR[rs]];

ID_Ex_B ← Reg[IF_ID_IR[rt]];

ID_Ex_NPC ← IF_ID_NPC;

ID_Ex_IR ← IF_ID_IR;
ID_Ex_Imm ← sign-extend (IF_ID_IR$_{15-0}$);



c) Ex

R-R ALU

Ex_MEM_IR ← ID_Ex_IR;
Ex_MEM_ALUOut ← ID_Ex_A func ID_Ex_B;

R-M ALU

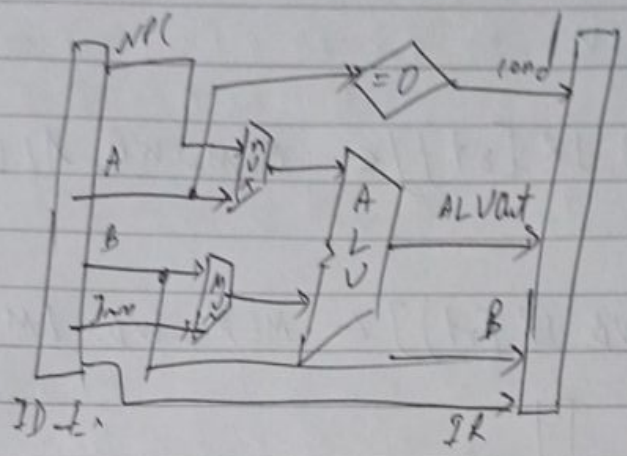
EX_MEM_IR ← ID_Ex_IR;
Ex_MEM_ALUOut ← ID_Ex_A . func ID_Ex_Imm;

R-M ALU Load/Store

Ex_MEM_IR ← ID_Ex_IR;
Ex_MEM_ALUOut ← ID_Ex_A + ID_Ex_Imm;
EX_MEM_B ← ID_Ex_B;

Branch

Ex_Mem_ALUOut ← ID_Ex_NPC + ID_Ex_Imm;
Ex_Mem_cond ← (ID_Ex_A == 0);



2) MEM

ALU

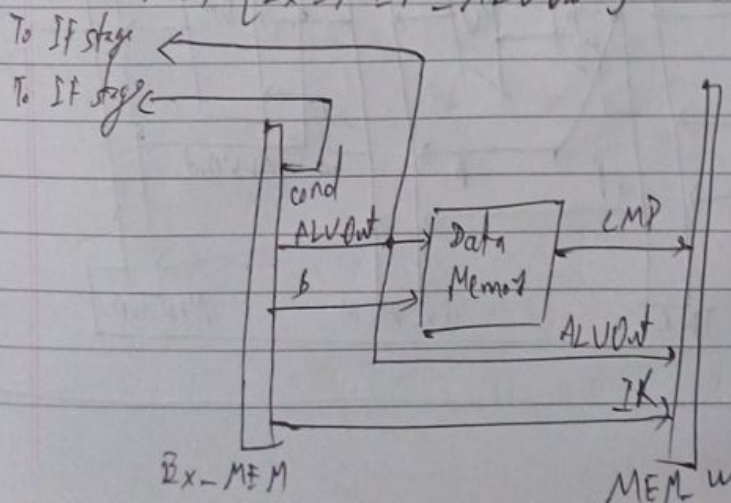
MEM_WB_IR ← Ex_MEM_IR;
MEM_WB_ALUOut ← Ex_MEM_ALUOut;

Load

MEM_WB_IR ← Ex_MEM_IR;
MEM_WB_LMD ← Mem[Ex_MEM_ALUOut];

Store

MEM_WB_IR ← Ex_MEM_IR;
Mem[Ex_MEM_ALUOut] ← Ex_MEM_B;



e) WB

R-R ALU

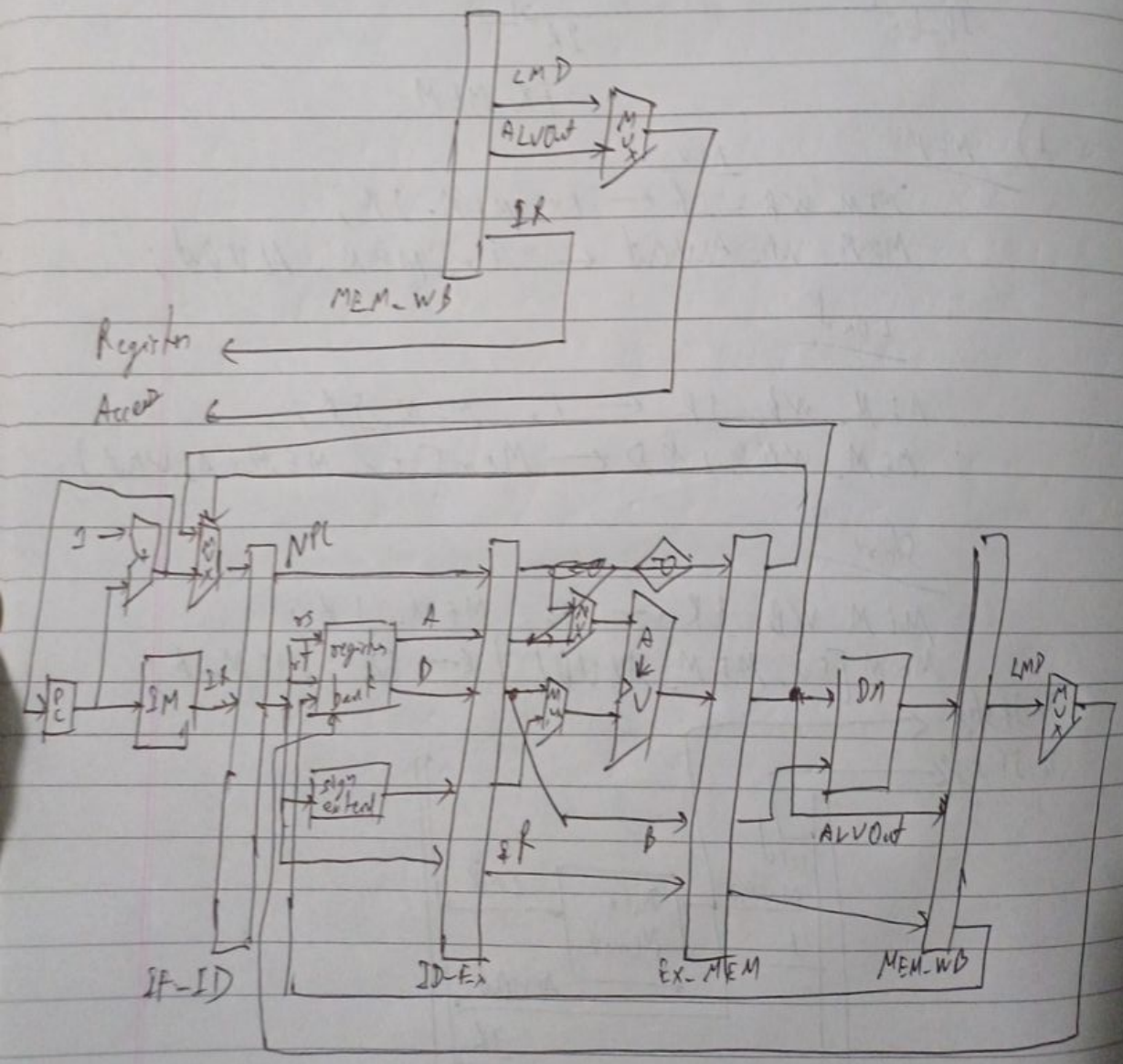
Reg[MEM_WB_IR[rd]] ← MEM_WB_ALUOut;

R-M ALU

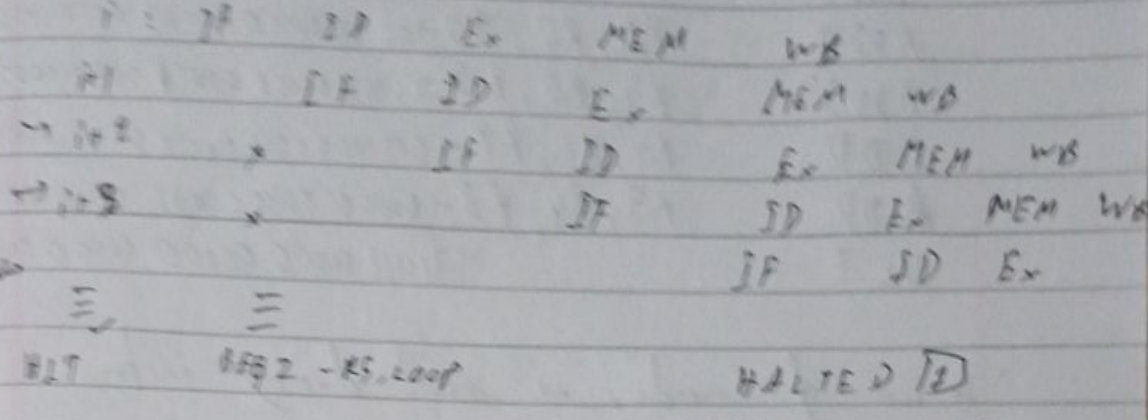
Reg[MEM_WB_IR[rt]] ← MEM_WB_ALUOut;

Load

Reg[MEM_WB_IR[rt]] ← MEM_WB_LMD;



Verilog implementation



- Two special 1-bit variables are used:
  - HALTED: set after a HLT instruction executes and reaches the WB stage.
  - TAKEN_BRANCH: Set after the decision to take a branch is known. Required to disable the instructions that have already entered the pipeline from making any state changes.

~~module pipe_MIPS32 (clk1, clk2);~~
~~input clk1, clk2; // Two phase clock~~
reg [31:0] PC, IF_ID_IR, IF_ID_

Ex: Add 3 nos 10, 20, 30 stored in processor registers.

- initialize register R1 with 10
- " " R2 with 20
- " " R3 with 25
- Add 3 nos & store the sum in R4, R5.

value = 0

ADDI R1, R0, 10 → 001010 00000 00001 0000000000001010
ADDI R2, R0, 20 → 001010 00000 00010 0000000000010100
ADDI R3, R0, 25 → 001010 00000 00011 0000000000011001
ADD R4, R1, R2 → 000000 00001 00010 00100 00000 000000
ADD R5, R4, R3 → 000000 00100 00011 00101 00000 000000
HLT → 111111 00000 00000 0000000000000000

- In case of hazards, we insert dummy instr.

Ex2/ Load a word stored in memory location 120, add 45 to it, and store the result in memory location 121.

ADDI R1, R0, 120 → 001010 00000 00001 0000000001111000
LW R2, 0(R1) → 001000 00001 00010 0000000000000000
ADDI R2, R2, 45 → 001010 00010 00010 0000000000101101
SW R2, 1(R1) → 001001 00001 00010 0000000000000001
HLT → 111111 00000 00000 0000000000000000

Ex3/ Compute the factorial of number N stored in memory location 200. The result will be stored in memory location 198.

- The steps:
  - initialize register R10 with the memory address 200.
  - Load contents of memory location 200 into register R3.
  - initialize register R2 with the value 1