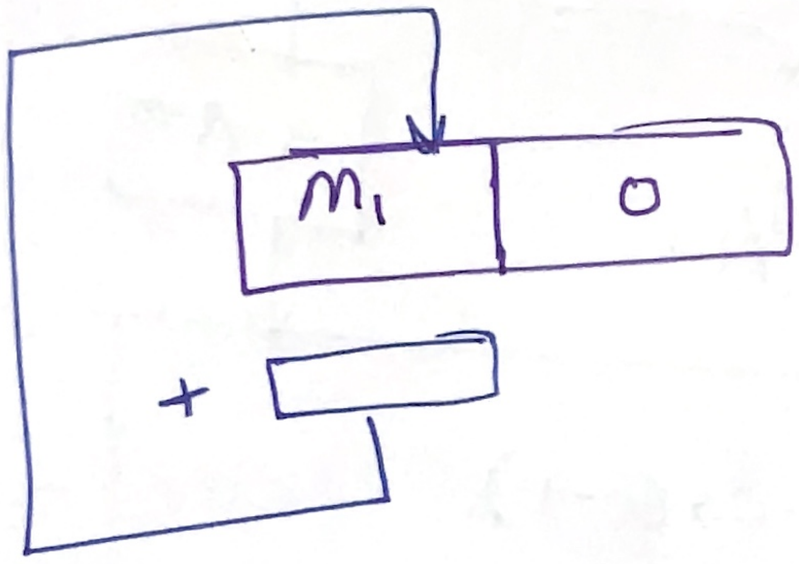
Lecture 27

Booth's multiplication.

```
  2 3 5     m1
  x 1 7     m2
-----------
1 6 4 5
2 3 5 x     (<<)
-----------
3 9 9 5     1 Addition.
```



Now based on m2
① We add something To m1
& Then it goes back To m1.
→ Then shift Right the thing here
(1645) >> and added

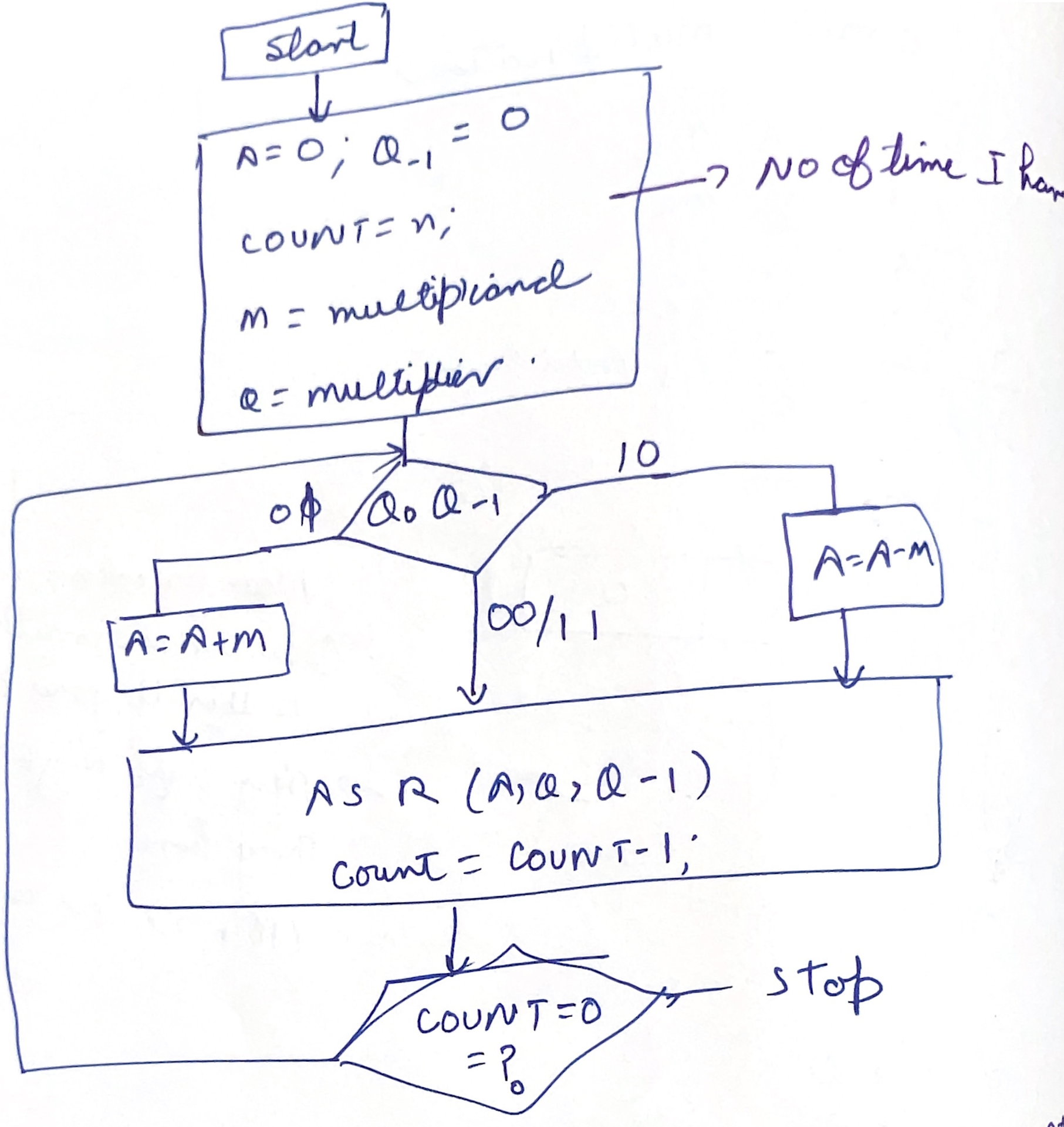
```
  0 1 0 1 1 0
  1 0 1
-----------
  0 1 0 1 1 0
0 0 0 0 0 0 -
1 0 1 1 0 - -
-----------
```
1 Add.

Then Booth's multiplication

n-bit multiplication.

<

skip in case of consecutive.
we inspect 2 bits Tothere Qi, Qi-1.
if same - just shift
01 → addition shift.
10 → sub & shift.

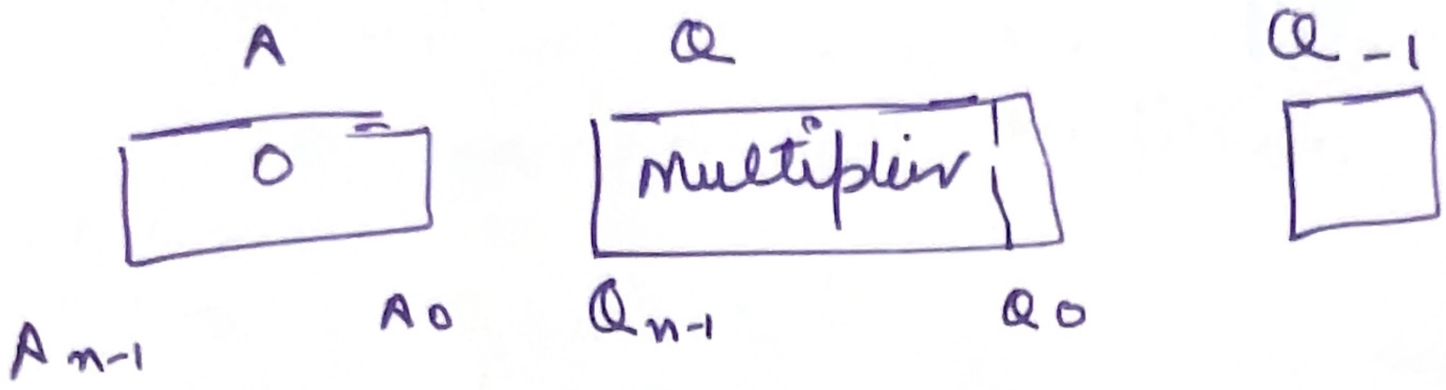


n = n-bit multi

Q = n-bit multip

A = n-bit temp reg

Q-1 = 1bit FF



4 x 5 = 20
↑ ↑ multiplier
multiplicand

(A ± M → A) then Rshift -
MSB is not change (shifted back).

1 0 1 0
8 4 2 1

-10 : M

13

-10 → (10110)2
-M → (01010)2 (+10) multiplicand

Q → (01101)2 (+13) multiplier

-10 x 13

A	Q	Q -1	
00000	01101	0	
01010	01101	0	A = A - M (1)
00101	00110	1	Add -M (1), shift (2)
00101 + (10110)			
11011	00110	1	A = A + M (3)
11101	10011	0	shift (2)
00111	10011	0	shift (3)
00011	11001	1	A = A - M (4)
00001	11100	1	shift (4)
10111	11100	0	
11011	11110	0	-130

final Res.

1 1 1 0 1
+ 0 1 0 1 0
1 0 1 1 1

1 1
1 1 1 0 1
+ 0 1 0 1 0
0 1 0 0 0 1
1 1 0 0 0

1 1 1 0 1
+ 0 1 0 1 0
0 0 1 1 1

$(-31) \times (20)$

multiplicand ↖ multiplier

→ multiplicand $_{10}$ multiplier

20 → 10100

$M = -31$

$= 11111$

00000

$+ 1$

00001

$M = 100001$

$-M = 011110$

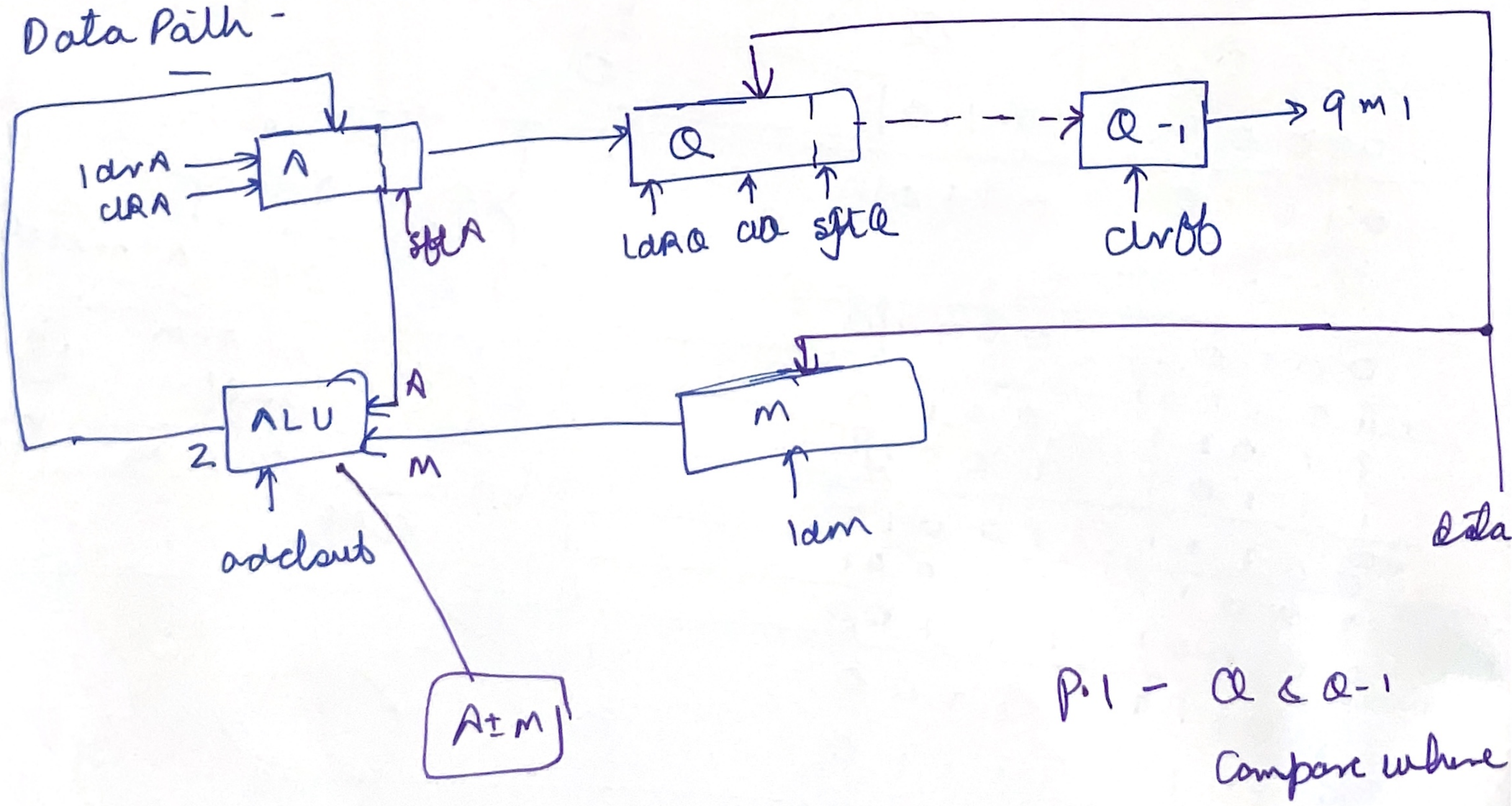
$+1$

$-M = 011111$

A	Q	Q-1	
000000	011100	0	
000000	001110	0	- (1)
000000	000111	0	- (2)
011111	000111	0	(3) A = A - M
001111	100011	1	
000111	110001	1	(4) Shift
000011	111000	1	(5) Shift
100100	111000	1	Add.
110010	011100	0	

only 2 Add/Sub — if same (Q, Q-1) are just need Shift.

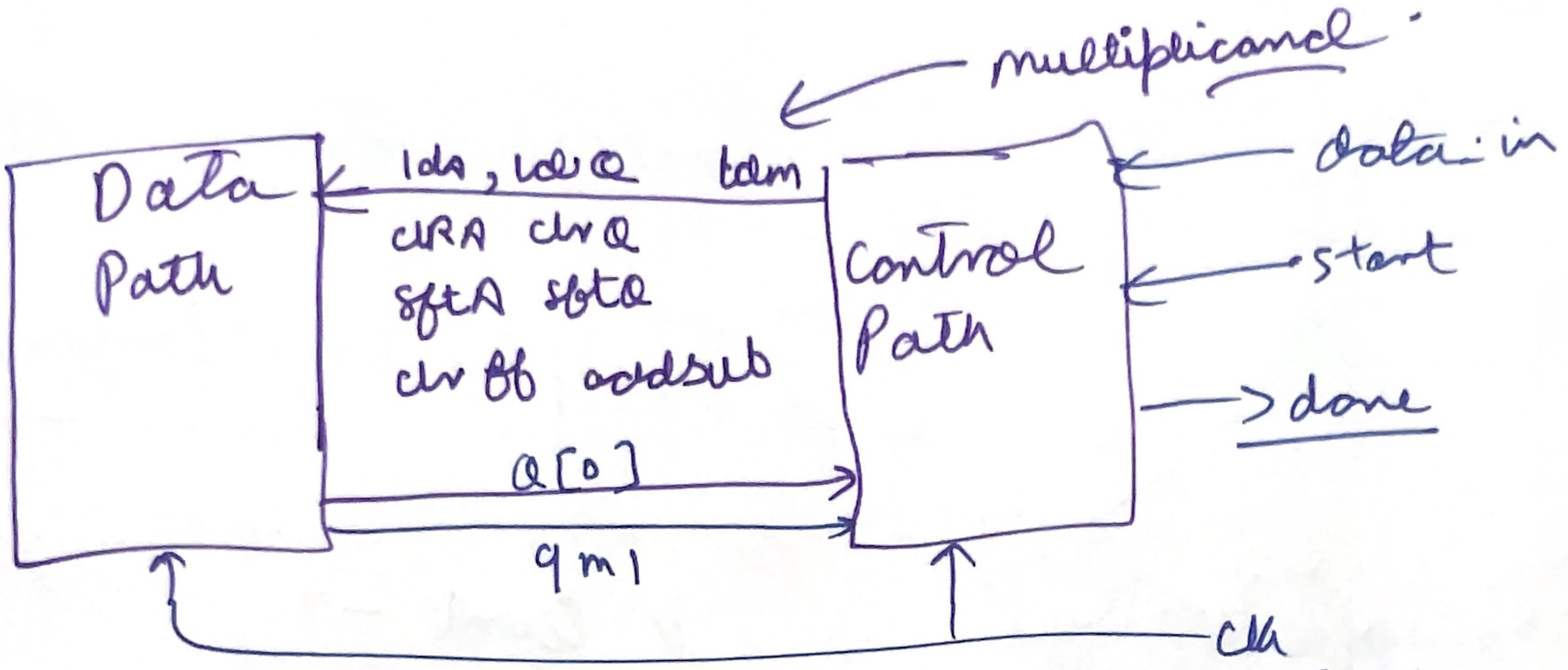
Data Path -



P.1 - Q & Q-1

Compare where

=

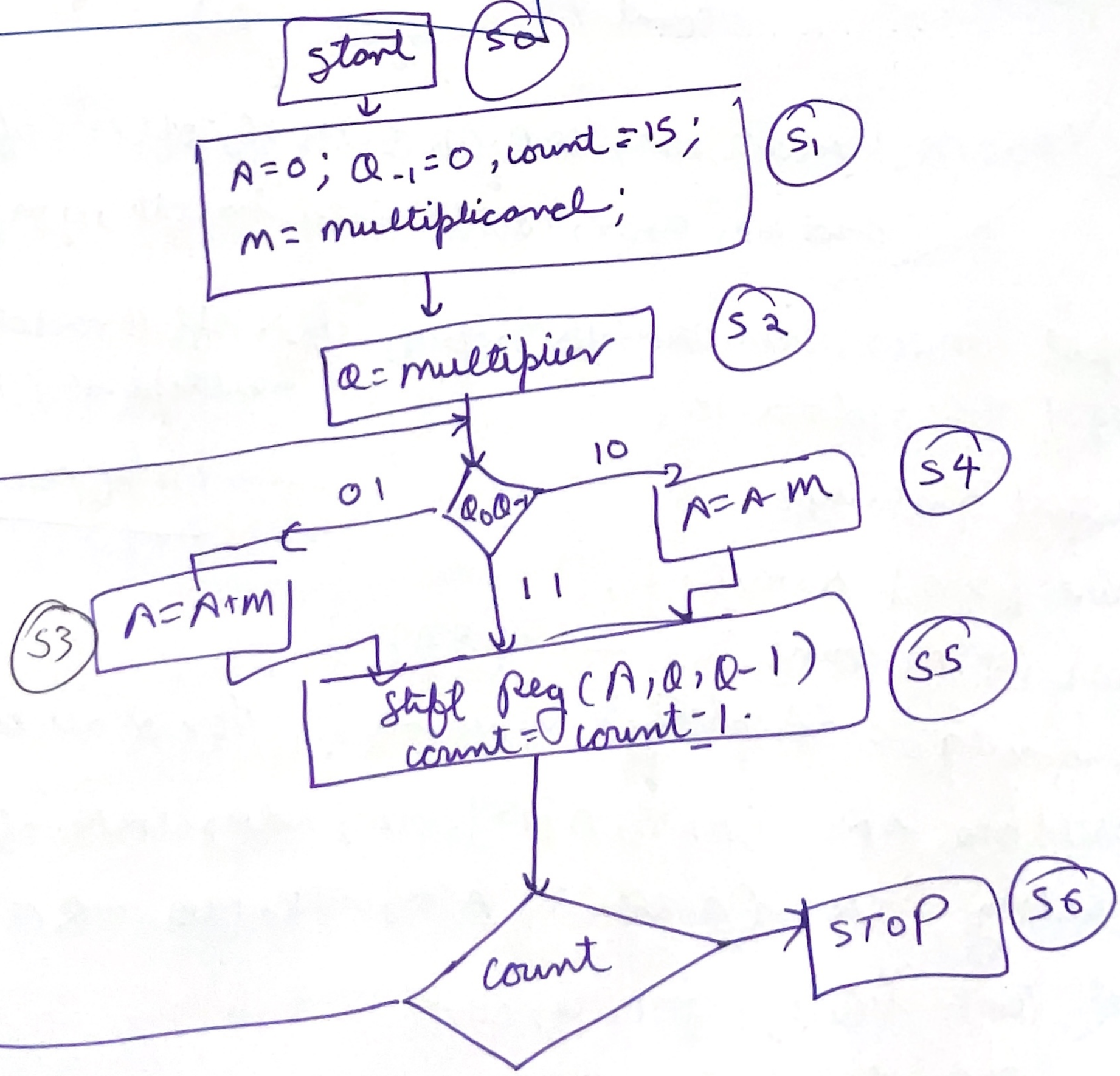


Q[0] is the first of multiplier

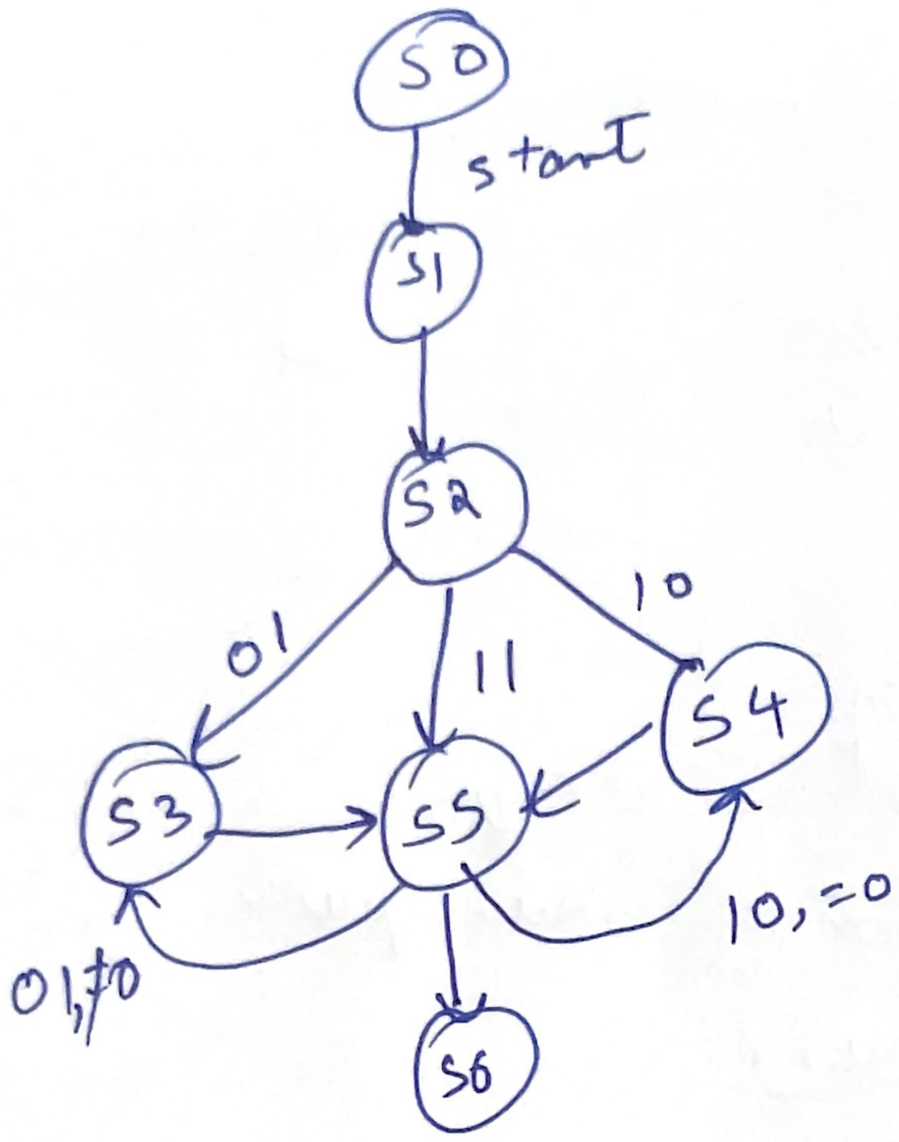
qm1 - q minus 1

Coursera Computer Architecture

Advance Computing



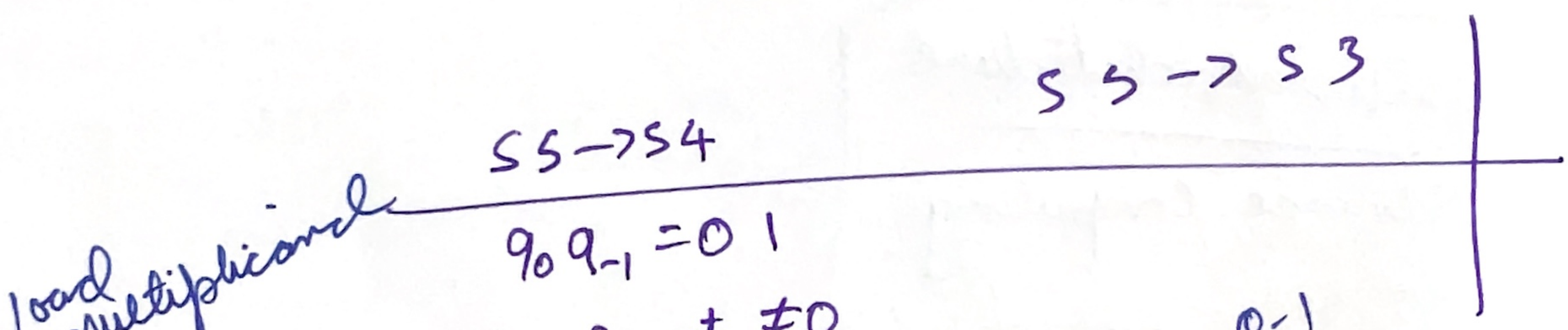
Count = 15 why.



// multiplicand → Q-1, count

// multiplier (Q) Comparison

// Count →




```
module BOOTH(ldA, ldQ, ldm, clrA, clrQ, clrff, sftA, sftQ,
             addSub, decr, ldcnt, data_in, clk, qm1, eq2);
input ldA, ldQ, ldm, clrA, clrQ, clrff, sftA, sftQ, addSub, clk;
input [15:0] data_in;          // multiplier / multiplicand
output qm1, eq2;
wire [15:0] A, m, Q, Z;
wire [4:0] count;
assign eq2 = ~|count;          // nor of all bits
shiftreg AR (A, Z, A[15], clk, ldA, clrA, sftA);
shiftreg QR (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.
```

Q0 → Q-1

O/P of ALU

```
module shift reg (data-out, datain, 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) data_out <= 0;
            else if (ld)
                data_out <= data-in;
            else if (sft)
                data_out <= { s_in, data-out[15:1]};
        end
endmodule.
```

LSB = 0

MSB : 1

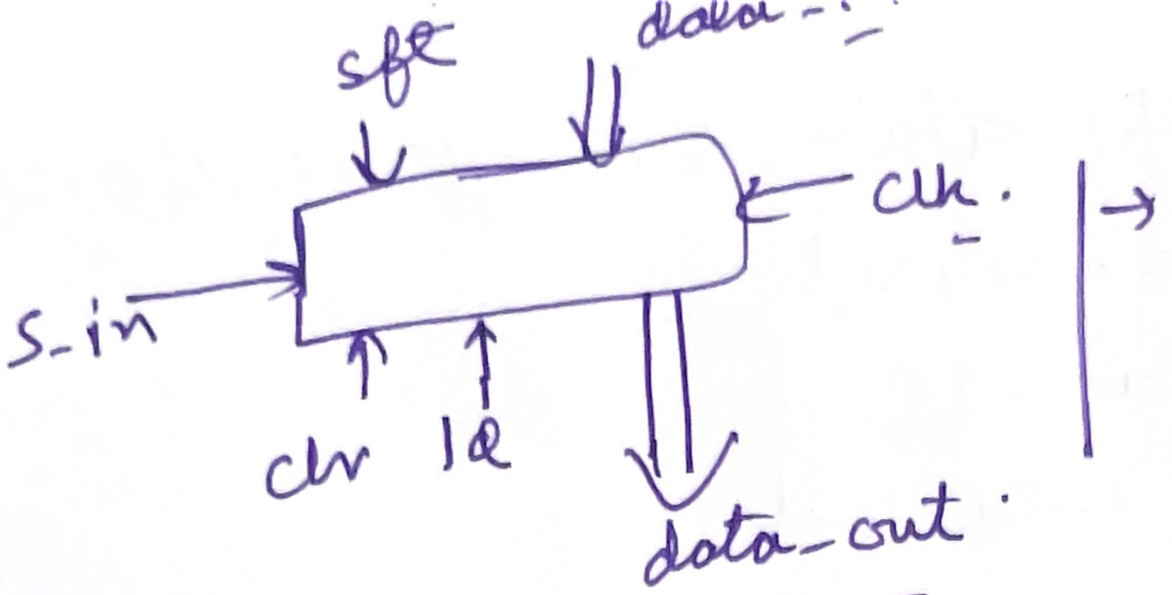
Shifting

```
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; // Active load.
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
```

else

open



C2S & DLI
3, 4

CFET

ASAP7 → ARM

36

ADC

PLL

22:37

YHB

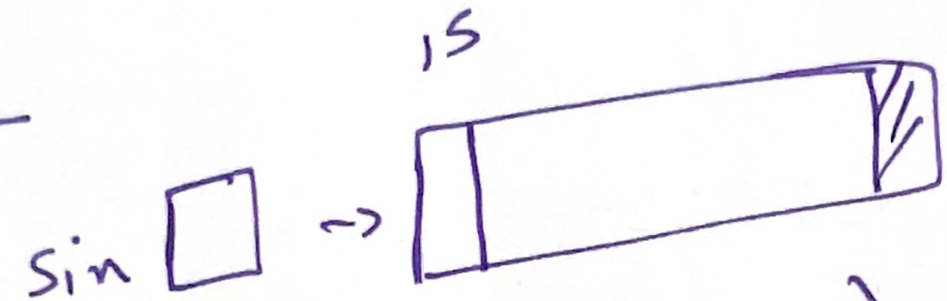
C → Xcelli

S → VCS

St

SRA

STA



CMOS

→ Pipeline.
→ Arbiter.

Verilog.

↳ Synthesis → 2nd

↳ DC (Design compiler)

C → Genus.

input reg A

input reg A, B, C.

Opensource tools.

Counter.V

tb_Counter.V

{ iverilog.
↳ Gtkwave

.lib

{ Yosys
.v

PDK

Skywater
^ 130nm

2022

EDC
Linting
XDC
P&R

Pre Sta

Area

Power.

GPU

CUDA

macros.

Process dev. kit

Optm_fsm.
Optm_muxes.
Set_don't use XOR

$I_D = \mu_n C_{ox}$

tox

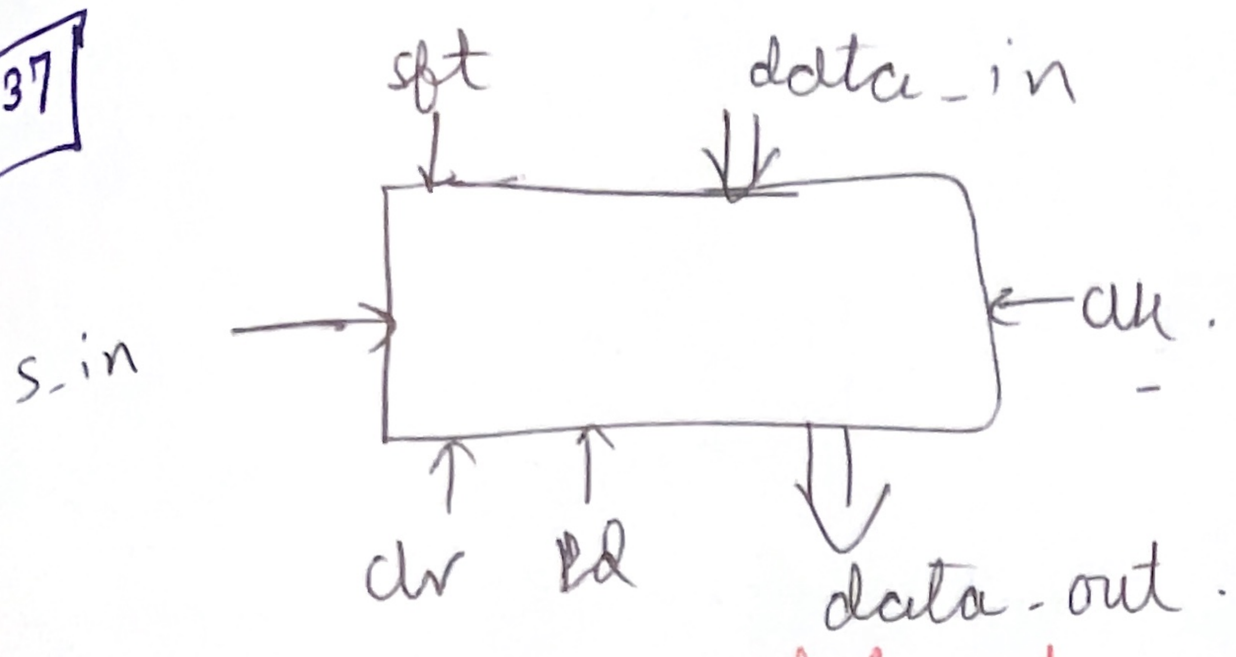
GLS.

Weetk

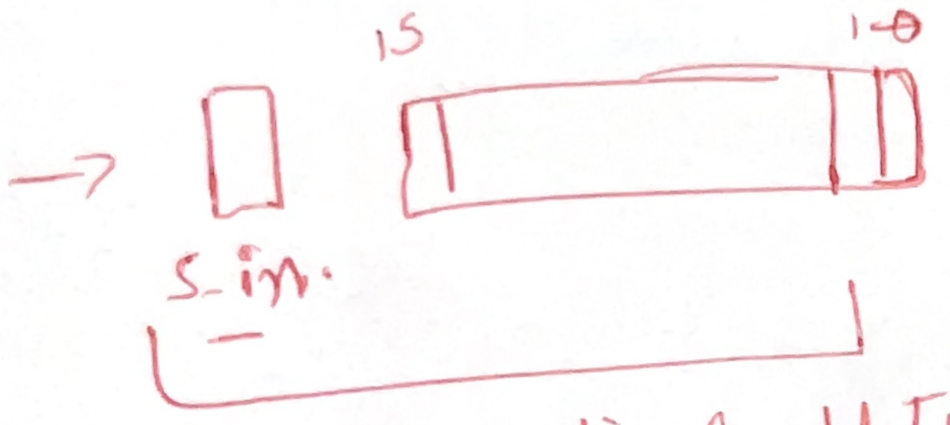
tox

Validator.

12:37



Shift Reg.



so this complete thouble be shifted right.

data_out <= {s_in, data_out[15:1]};

```
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, ldQ, clrQ, sftQ, Bm,
                    clrff, addsub, start, decr, ldcnt, done,
                    clk, q0, qm1 );

input clk, q0, qm1, start;                          // q0 to be pushed.
output reg ldA, clrA, sftA, ldQ, clrQ, sftQ,
           ldm, clrff, addsub, decr, ldcnt, done;

reg [2:0] state;
Parameter s0 = 3'b000, s1 = 3'b001, s2 = 3'b010, s3 = 3'b0
          s4 = 3'b100, s5 = 3'b101, s6 = 3'b110;

always @(Posedge clk)
  begin
    case (state)
      s0 : if(start) state <= s1;
      s1 :  state <= s2;
      s2 : #2 if ({q0, qm1} == 2'b01) state<=s
              else if ({q0, qm1} == 1'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,qm1} == 2'b10 && !eq2)
                   state <= s4;

              else if (eq2)  state <= s6;
      s6 : state <= s6;
      default : state <= s0;
    endcase
  end
```

// Blocking assignments whenever state changes.

always @ (state)
begin
case (state)
s0 : begin clrA = 0; ldA = 0; sftA = 0; clrQ = 0; ldQ = 0;
sftQ = 0; ldm = 0; clrff = 0; done = 0; incQ

s1 : begin clrA = 1; clrff = 1; ldcnt = 1; ldcnt = 1; ldm = 1

incQ

s2 : begin clrA = 0; clrff = 0; ldcnt = 0; ldm = 0; ldQ = 1; incQ
→
s3 : begin ldA = 1; addsub = 1; ldQ = 0; sftA = 0; sftQ = 0;
decr = 0; end.

s4 : begin ldA = 1, addsub = 0; ldQ = 0; sftA = 0; sftQ = 0;
decr = 0; end.

s5 : begin sftA = 1; sftQ = 1; ldA = 0; ldQ = 0; decr = 1; end

s6 : done = 1;
default : begin clrA = 0; sftA = 0; ldQ = 0; sftQ = 0; end

endcase
end.

s3 → A, M, load A, load M, Addsub.
load Q.

Write TBin a similar way. → gain delay in some places.

Synthesizable Verilog:

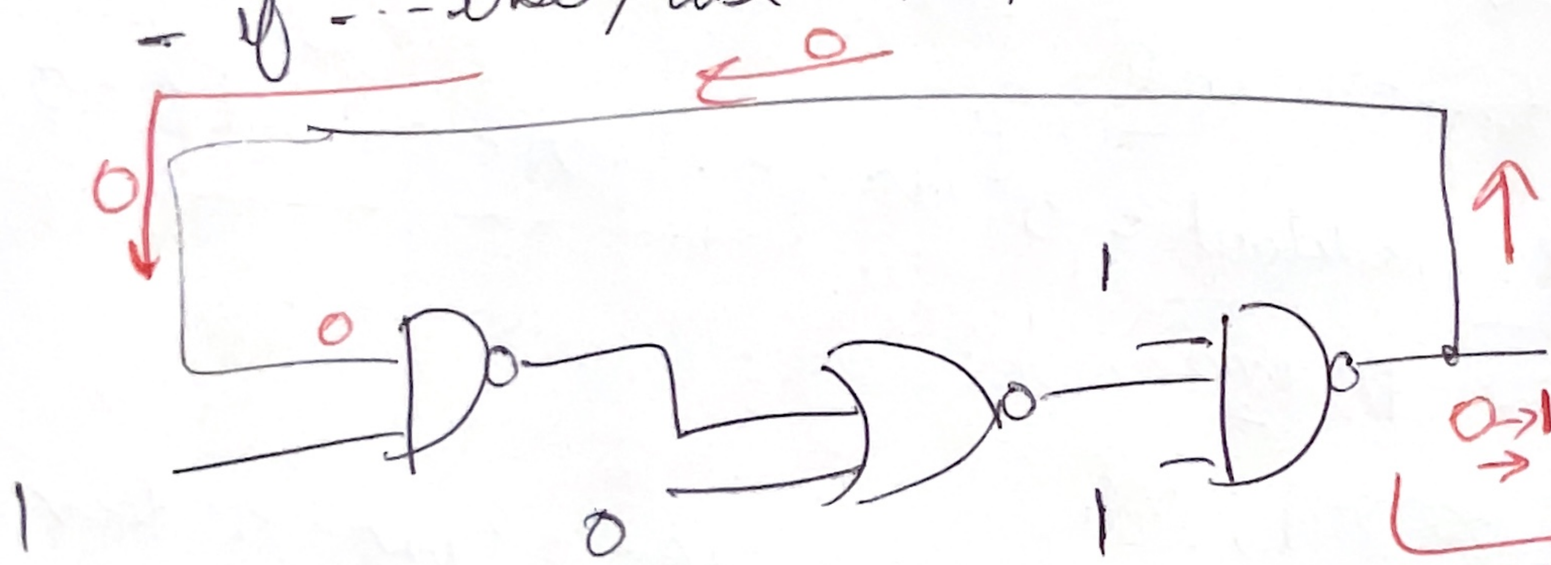
→ Supported by simulation tools

O/P of Combinational logic ckt at time t should depend only upon the inputs at time t.

Rules to be followed

- Avoid technology dependent modeling.
- There Must be not any feedback in the combinational ckt.

- if - - else / case all I/P must be



Feedback is not Needed.

Not a stable O/P.

Sequential:

- if we forget Combination (all) in if else / case it will make a latch, (sequential).

Styles for synthesizable Combinational logic.

- Netlist of Verilog built in primitives like gate instances
- Combinational UDP.

Continuous assignments

Functions

Behavioural statements

Task without event / delay control

Synthesis Tool May Do optimizations.

```
module carry (cout, a, b, c)        -> Cont Assign.
    input a, b, c;
    output cout;
    assign cout = (a & b) | (b & c) | (c & a);
endmodule
```

using Procedural

```
module mux (B, in0, in1, sel);
    input in0, in1, sel;
    output regB;                          I/P in always block.
    always @(in0 or in1 or sel)  { // will use a latch
        if (sel) B = in1;
        else B = in0;
endmodule
```

function → a single o/p.
Tasks → More than 1 o/P value.

always

```
module fulladder (s, cout, a, b, cin);
    input a, b, cin;
    output reg s, cout;
    always @ (a or b or cin)
        FA (s, cout, a, b, cin);
task FA;
    output sum, carry;
    input A, B, C;
    begin
        #2 sum = A ^ B ^ C;
        carry = (A & B) | (B & C) | (C & A);
    end
endtask
endmodule
```

Fxn vs Task.

Fxn
Can Call Another Fxn

Fxn will be executed in a delay.

— Cannot Contain any delay, event / timing Control statement.

Return a single Value.

Task
Can call a Fxn and Task

Can Control event, delay / timing Control statement

O/P may Return More than Value.

Atleast I/I/P argument.

— To Avoid for Combinational

① Edge dependent event control
② Combinational feedback loops.
③ Procedural loops with timing are Not allowed
④ No delay control's.
⑤ No of times I & M looping is allowed (condi
⑥ fork . . join
wait
disable.

Non Synthesizable Constructs.

initial
Delay in assignments
time construct
// === & !==
// fork join

Force release Variable

active high
active low signals — (-b) for active low.

(-clk) (bus_clk)

Instruction.

OPCODE	REGS	OPERAND

```
reg [5:0] OPCODE;
reg [0:5] RFGS;
reg [15:0] OPERAND;
wire [27:0] INSTR
assign INSTR = {OPCODE, REGS, OPERAND};
```

Unit 6