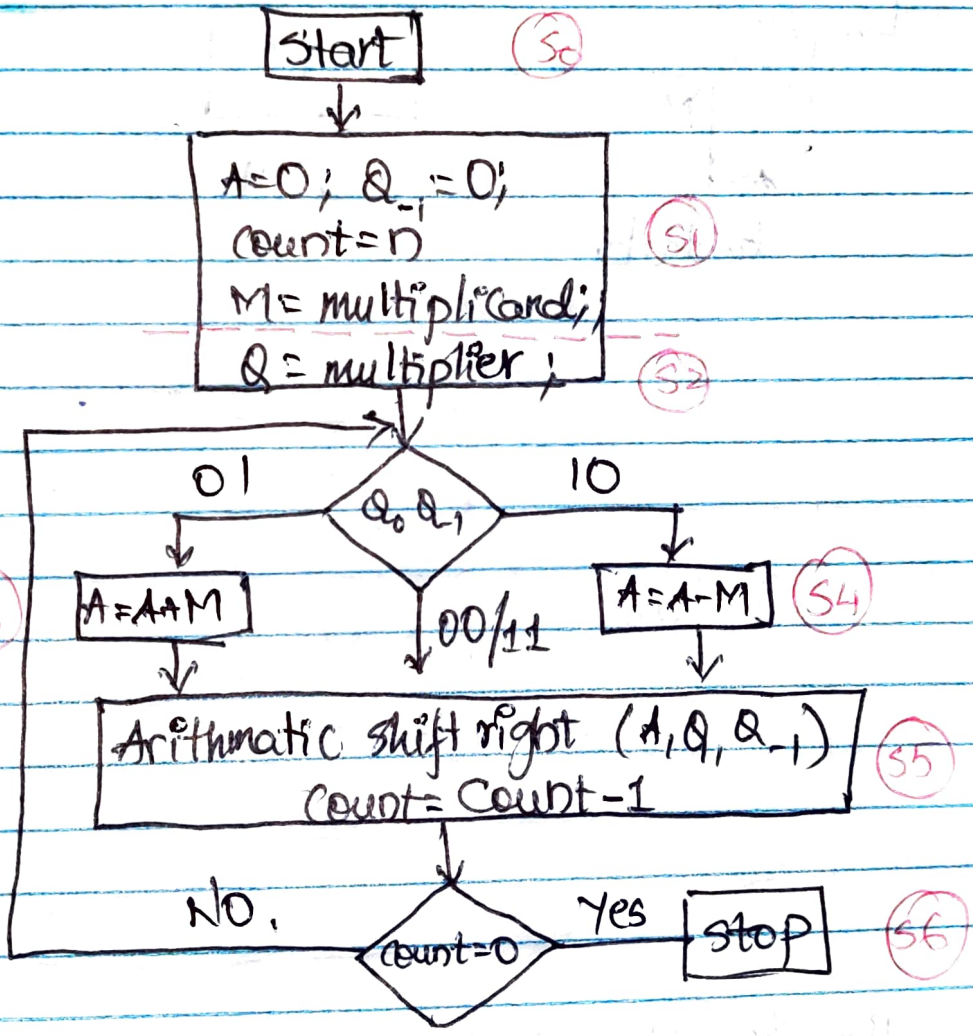
Q13. Booth's Multiplication algorithm for signed multiplication.

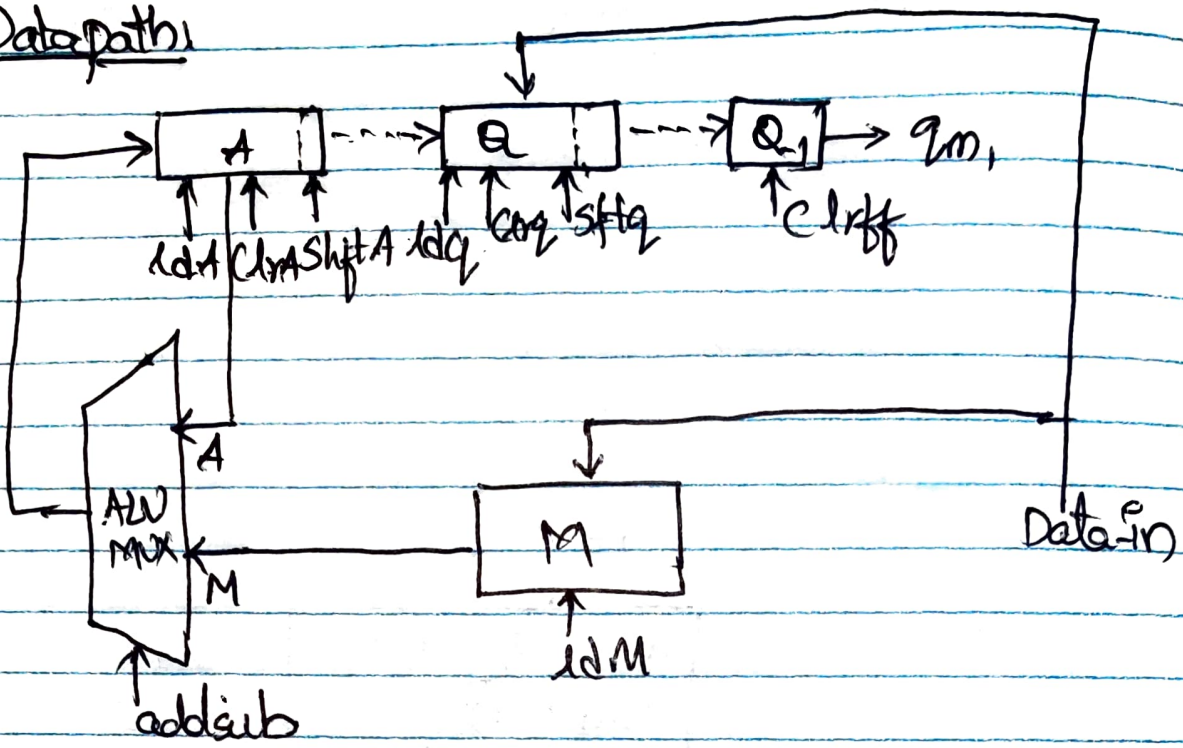
Rule is to simplify multiplication process by
- → inspecting (Q_i, Q_{i-1}) bits at the same time
- → if bits are (01) we add and shift
- → if bits are (10) we substract and shift
- → Q_{-1} is assumed to be 0.

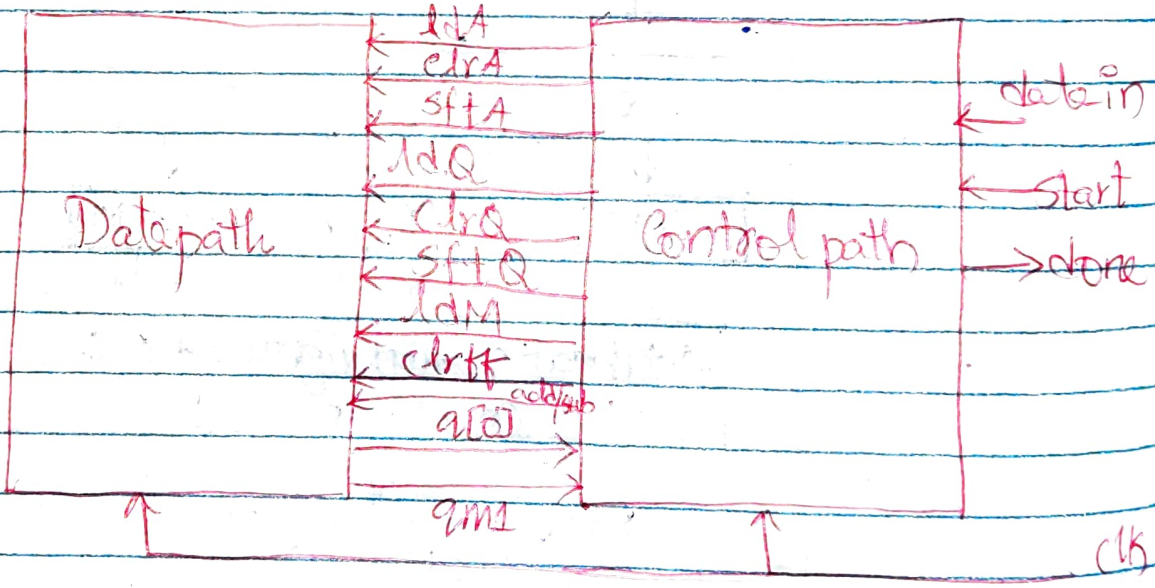
This process reduces number of additions/substractions.



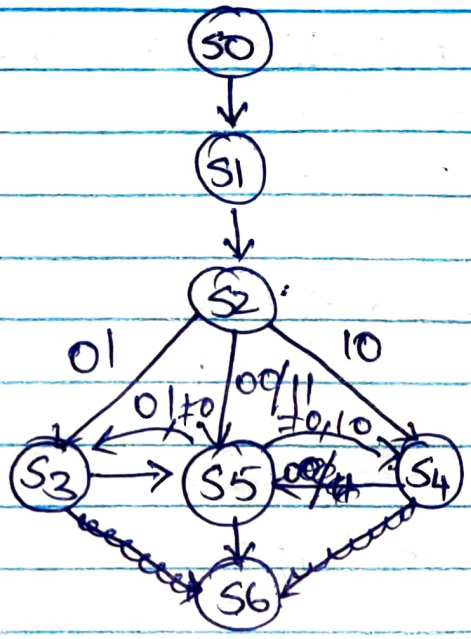
Flowchart for booth's multiplier algorithm.

Datapath:





from the flow chart the state transitions are defined as follows:




```
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, shftQ, addsub, clk;
input [15:0] data_in;
wire [15:0] A, M, Q, z;
wire [4:0] count;
assign eqz = ~|count;

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  C4(count, decr, ldcnt, clk);
endmodule
```

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

shiftreg (dataout, 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) data_out <= 0;
      else if (ld) data_out <= data_in;
      else if (sft)
                dataout <= {sin, dataout[15:1]};
      end
endmodule
```

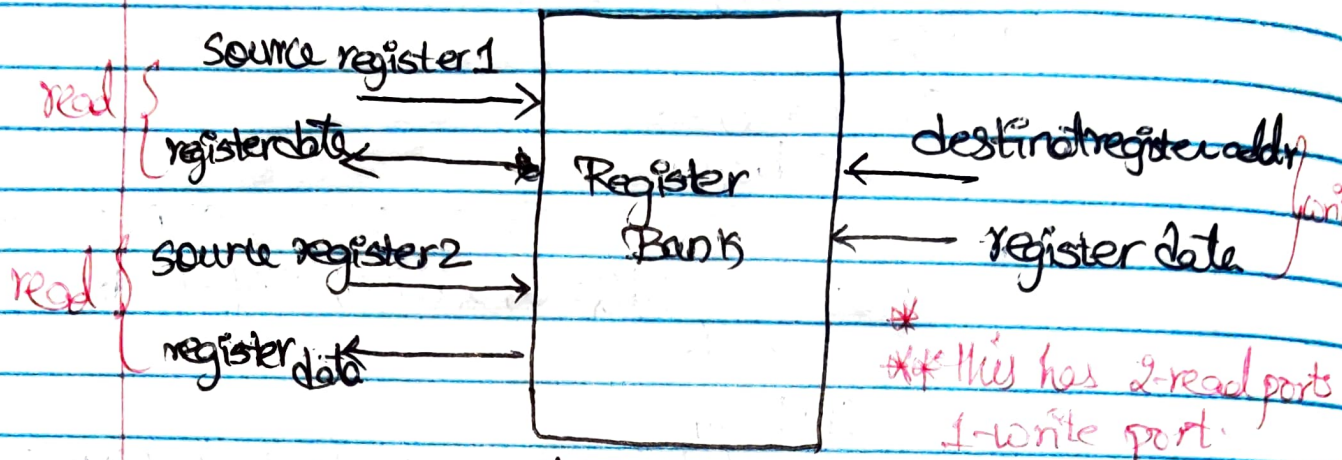
```
module ALU (out, in1, in2, addsub);
  input [15:0] in1, in2;
  input addsub;
  output reg [15:0] out;
  always @(*)
      begin
        if (addsub == 0)
            out = in1 - in2;
        else
            out = in1 + in2;
        end
endmodule

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)
            datata_out <= data_out - 1;
      end
    endmodule;
```

```
module controller (ldA, clrA, sftA, ldQ, clrQ, sftQ, ldM,
                   clrff, addsub, start, decr, ldcnt, done
                   clk, q0, qm1);
  input clk, q0, qm1, start;
  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'b011, 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 <= S3;
            else if ({q0, qm1} == 2'b10) state <= S4;
            else state <= S5;
        S3: state <= S5;
        S4: state <= S5;
        S5: #1 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.
```

```
always @(state)
  begin
    case (state)
      s0: begin
            clrA=0; ldA=0; sftA=0; clrQ=0; sftQ=0;
              ldQ=0; ldM=0; clrff=0; done=0
          end
      s1: begin
            clrA=1; clrff=1 ldcnt=1; ldM=1
          end
      s2: begin
            clrA=0; clrff=0; ldcnt=0; ldM=0;
            ldQ=1;
          end
      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
endmodule
```

Modelling register banks



an example for smaller register banks.

```
module register_banks_v1 (rddata1, rddata2, wrData, sr1, sr2,
        dr, write, clk);
    input clk, write;
    input [1:0] sr1, sr2, dr;
    input [31:0] wrdata;
    output reg [31:0] rdData1, rdData2;
      reg [31:0] R0, R1, R2, R3;
    always@(*)
        begin
            case (sr1)
              0: rdData1 = R0;
              1: rdData2 = R1;
              2: rdData1 = R2;
              3: rdData1 = R3;
            default: rdData1 = 32'hxxxxxx;
            endcase
    end
```

```
always@(*)
    begin
        case(SR2);
        0: rdData2 = R0;
        1: rdData2 = R1;
        2: rdData2 = R2;
        3: rdData2 = R3;
        default: rdData2 = 32'hxxxxxxxx;
    endcase
end
always@(posedge clk)
    begin
        if(write)
            case(dr)
                0: R0 <= wrData;
                1: R1 <= wrData;
                2: R2 <= wrData;
                3: R3 <= wrData;
            default:
            endcase
        end
endmodule
```

This can be implemented with assign statements and conditional operators. ** Model during simulation. Regbank_v2. Register banks V2 is a simple assign statement for scaled version of 4x32 to 32x32. Implement during projects.

```
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;
   integer k;

  reg [31:0] regfile [0:31];        //check for [31:0] aswell.

  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 begin
          if (write)
            regfile[dr] <= wrData;
          end

    end
endmodule
```

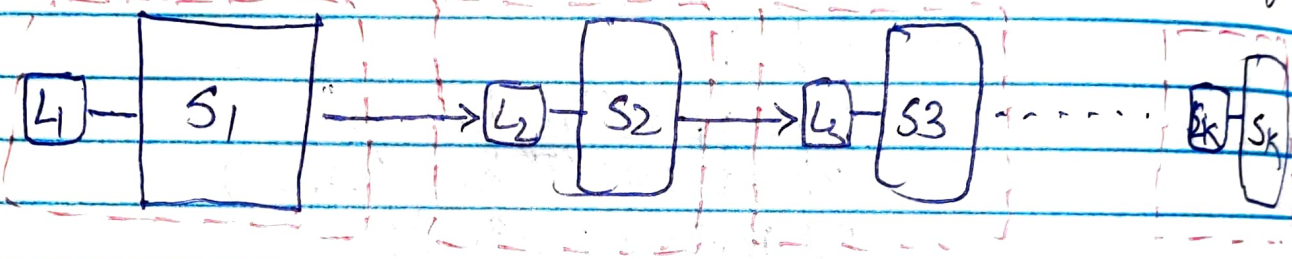
```
module regfile_test;
    reg [4:0] sr1, sr2, dr;
    reg [31:0] wrData;
    reg write, reset, clk;
    wire [31:0] rdData1, rdData2;
    integer k;
    regbanks_v4 reg (rdData1, rddata2, wrData, sr1, sr2, dr, write,
                     reset, clk);
    initial clk = 0;
    always #5 clk = !clk;
    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 ; write=1;
          #10 write=0;
        end
      #20
        for (k=0; k<32; k=k+2);
        begin
          sr1=k; sr2=k+1;
        #5
        $display ("reg[%2d] = %d,
```

```
        reg[%2d] = %d", sr1, rdData1, sr2, rdData2);
    end
    #200 $finish;
end
endmodule
```

Pipelining:

Overlapping of computational stages.

Requires latches to store data from previous stage



$L_1 \cdots L_k$ are latches to store each stage o/ps

$S_1 \cdots S_k$ are combinational circuits for computation

linear pipline

non-linear pipline.

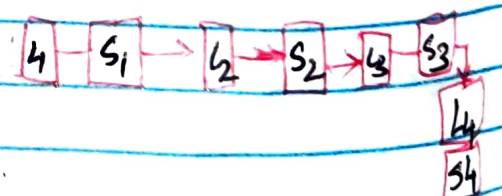
Reservation table

→ X-axis time steps

→ Y-axis stages

fourstage linear pipeline reservation table

	1	2	3	4
S_1	X			
S_2		X		
S_3			X	
S_4				X



Non linear reservation table is complex structure due to feedback and feedforward combinations.

Speed up & efficiency:

τ = clock period of pipeline
t_i = time delay of S_i the stage
d_L = latch delay
maximum stage delay $\tau_m = \max\{t_i\}$
$\Rightarrow \tau = \tau_m + d_L$
pipeline frequency $f = \frac{1}{\tau}$

Total time to process N dataset is

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

since $k-1$ is for filling $k-1$ slots.
and N stage outputs for all slots.

for a non-pipelined processor

$$T_1 = N \cdot k \cdot \tau$$

since N dataset and k timeslots.

$$\text{Speed up} = \frac{T_1}{T_k} = \frac{k(N)k\tau}{[((k-1)+N)\tau]} = S_k.$$

as $N \to \infty$ T_1 for S_k is k
i.e $N \to \infty$ $S_k = k$ speed $\simeq$ total stage delay

$$\text{efficiency } E_k = \frac{S_k}{k} = \frac{N}{k+(N-1)} \quad [\text{performance}]$$

$$\text{throughput } \frac{N}{T_k} = \frac{N}{[k+(N-1)]\tau} \quad [\text{no. of operations per unit time}]$$

Clock skew: max. delay difference between arrival of clocks.

Jitter: maximum delay difference between arrival of clock signal at same latch.

Logic delay: Maximum delay of slowest stage of pipeline

Setup time: time taken for clock to become stable at the input of latch before it can be captured

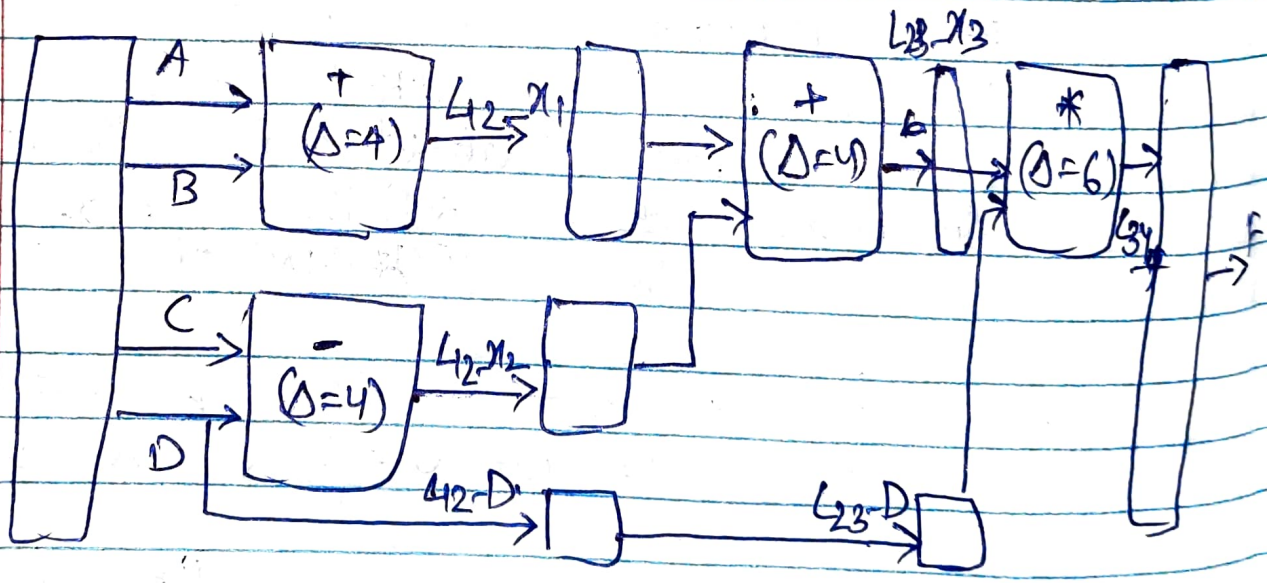
A simple pipelining example with 3 stages.

Given A, B, C, D are n-bit inputs

a) S1 : $x_1 = A + B$; $x_2 = C - D$;

b) S2 : $x_3 = x_1 + x_2$;

c) S3 : $F = x_3 * D$;



```
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, L23_D, L23_X3, L23_D,
        L34_F;
    assign F = L34_F;
    always @(posedge clk) begin
        L12_X1 <= #4 A+B;          //Stage 1
        L12_X2 <= #4 C-D;
        L12_D <= D;
        L23_X3 <= #4 L12_X1 + L12X2; //Stage 2
        L23_D <= L12_D;
        L34_F <= #6 L23_X3 * L23_D; //Stage 3
    end
endmodule
```

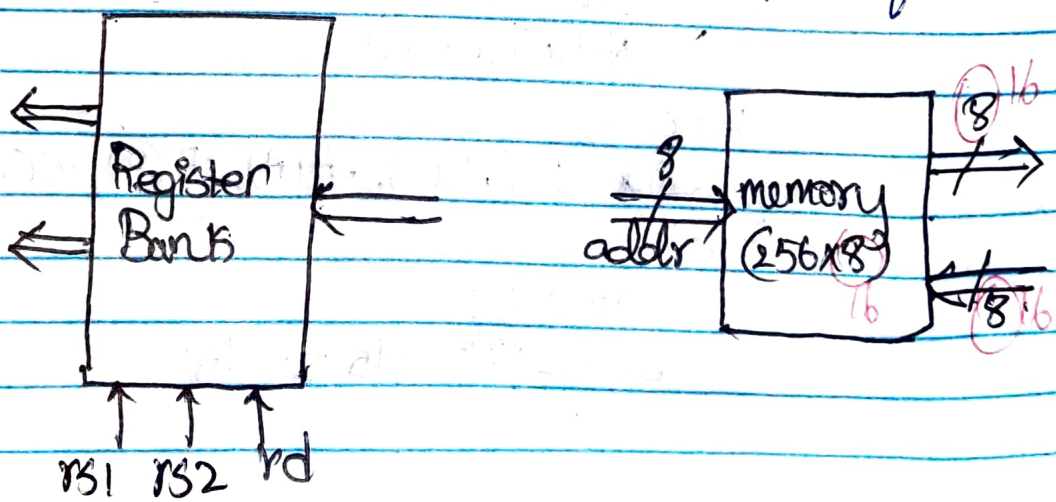
Stages can be divided into seperate procedural blocks

```
module pipe_test;
    parameter N=10;
    wire [N-1:0] F;
    reg [N-1:0] A, B, C, D;
    reg clk;
    pipe_ex p (.F(F), .A(A), .B(B), .C(C), .D(D), .clk(clk));
    initial
        clk = 0;
    always #10 clk = ~clk;
```

```
initial begin
  #5 A=10; B=12; C=6; D=3;
  #20 A=10; B=10; C=5; D=3;
  #20 A=20; B=11; C=1; D=4;
 #20 A=15; B=10; C=8; D=2;
 #20 A=8 ; B=15; C=5; D=0;
  #20 A=30; B=1 ; C=2 ; D=4;
 end
initial begin
    $dumpfile ("pipeline.vcd");
    $dumpvars (0, pipe_test );
  $monitor ($time, "F=%d", F);
  $300 Finish;
 end
endmodule
```

In this example due to common clock for all latches there are chances of race condition. This can be avoided using ***Master Slave Flip Flops***

Consider following example for subpart of MIPS implementation



Stage wise information

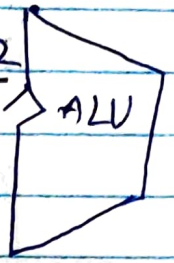
Inputs: 3 register addresses (rs1, rs2, rd), an ALU function and memory address (addr)

stage 1: Read two 16-bit numbers from RS1 & RS2 and store in A & B (temp regs)

stage 2: perform ALU on A & B by func and store in Z (temp regs)

stage 3: Write the value of Z in register specified by rd.

stage 4: Write value of Z in memory location "addr"



Assumptions for design

Register bank with 16- 16-bit registers

Memory is organised as 256x16

8-bits are required to specify memory address

each location contains 16-bits data

ALU function (func) is selected by four bit variable

0000: Add 0001: SUB 0010: MUL

0011: SELA 0100: SELB 0101: AND

0110: OR 0111: XOR 1000: NEGA

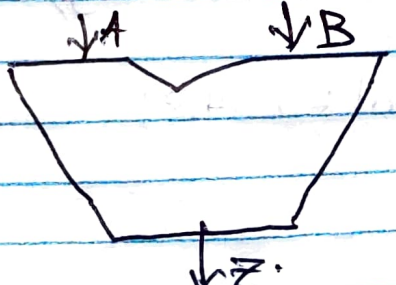
1001: NEGB 1010: SRA 1011: SLA

ADD: A+B SELB: Z=B NEGA: Z = ~A

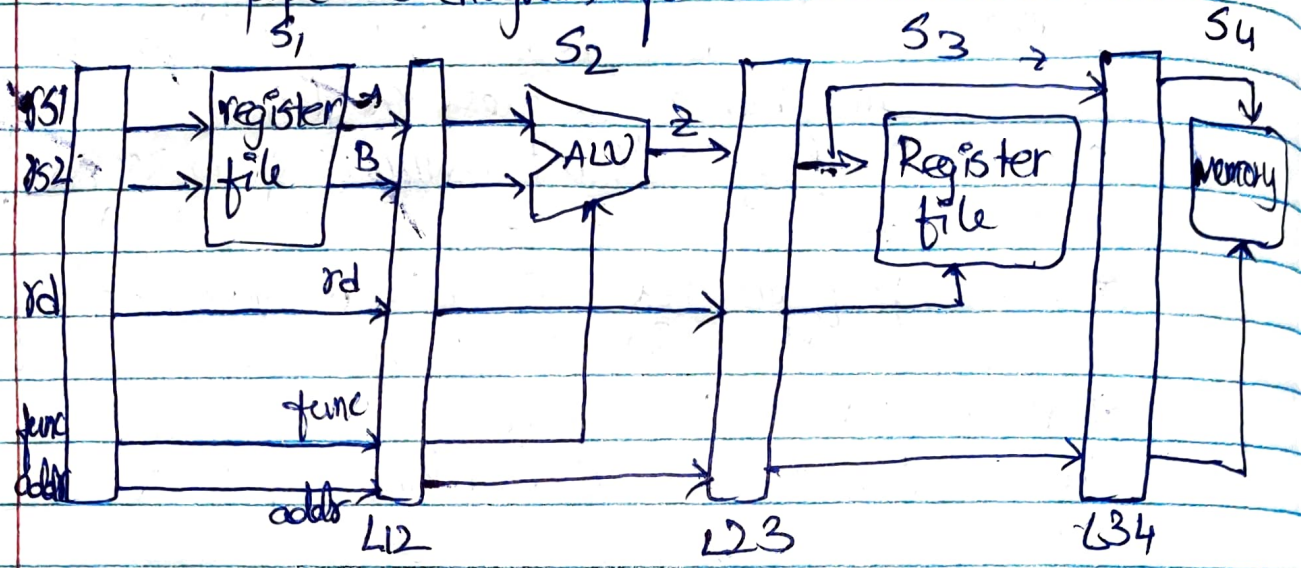
SUB: A-B AND: A & B NEGB: Z = ~B

MUL: A*B OR: A | B SRA: Shift Right A

SELA: Z=A XOR: A^B SLA: Shift Right A



pipeline diagram for above details:



** Applying two clocks non-overlapping and alternate ffs are activated. While designing in Verilog try with the Master Slave ffs instead of two phase clocking.

```
module pipe_ex2(Zout, rs1, rs2, rd, func, addr, clk1, clk2);
  input [3:0] rs1, rs2, rd, func;
  input [7:0] addr;
  input write, clk1, clk2;
  output [15:0] Zout;
  reg [15:0] L12A, L12B, L23Z, L34Z;
  reg [3:0] L12rd, L23rd, L12func;
  reg [7:0] L12addr, L23addr, L34addr;

  reg [15:0] regbank [0:15];
  reg [5:0] mem [0:255];

assign Zout = L34.Z;

//Stage 1 of pipeline
```

```
always @(posedg clk1)          //clk 1
  begin
    L12A <=#2 regbank[rs1];
    L12B <=#2 regbank[rs2];
    L12rd <=#2 rd;
    L12rd <=#2 func;
    L12addr<=#2 addr;
  end.
//stage 2 of pipeline
  always @(posedge clk2)       //clk 2.
    begin
      Case (func)
        0: L23Z <=#2 L12A + L12B;
        1: L23z <=#2 L12A - L12B;
        2: L23z <=#2 L12A * L12B;
        3: L23z <=#2 L12A;
        4: L23z <=#2 L12B;
        5: L23z <=#2 L12A & L12B;
        6: L23z <=#2 L12A | L12B;
        7: L23z <=#2 L12A ^ L12B;
        8: L23z <=#2 ~L12A;
        9: L23z <=#2 ~L12B;
        10: L23z <=#2 L12A >>1;
        11: L23z <=#2 L12A <<1;
      default: L23_z <=#2 16'bXXXX;
    endCase
    L23rd <=#2 L12rd;
    L23addr<=#2 L12addr;
  end
```

```
//Stage 3 of pipelining
always @(posedge clk1);
  begin
    Regbank[L23rd] <= #2 L23z;
    L34z <= #2 L23z;
    L34addr <= #2 L23addr;
  end
//Stage 4 of pipeline
always @(negedge clk2);
  begin
    mem[L34addr] <= #2 L34z;
  end
endmodule
//testbench
module pipetest;
  wire [15:0]z;
  reg [3:0] rs1, rs2, rd, func;
  reg clk1, clk2;
  reg [7:0] addr;
  integer k;

  pipe_en2 pipe(.zout(z), .rs1(rs1), .rs2(rs2), .clk1(clk1), .clk2(clk2)
          .rd(rd), .func(func), .addr(addr));
  initial
    begin
      clk1=0; clk2=0;
      repeat (20)
        begin
          #5 clk1=1 ; #5 clk1=0;
```

```
        #5 clk2 = 1; #5 clk2 = 0;
    end
initial begin
    for (k=0; k<16; k=k+1)
        pipe.regbank[k] = k;

initial begin
 #5 rs1=3; rs2=5; rd=10; func=0; addr=125;
#20 rs1=3; rs2=8; rd=12; func=2; addr=126;
#20 rs1=10; rs2=5; rd=14; func=1; addr=128;
#20 rs1= 7; rs2=3; rd=13; func=1; addr=127;
#20 rs1=10; rs2=5; rd=15; func=1; addr=129;
#20 rs1=12; rs2=13; rd=16; func=0; addr=130;

#60 for (k=125; k<131; k=k+1)
        $display ("Mem[%3d] = %3d", k, pipe.mem[k]);
end
initial begin
    $dumpfile ("pipe2.vcd");
    $dumpvars (0, pipe_test);
    $monitor ($time, "F = %3d", Z);
    #300 $finish;
end
endmodule
```