```
p <= pp
process(areset, clk)
    if areset = 1                          -- initialize
        pp <= 1
        xx <= x
        nn <= n
    elsif rising_edge
        if nn /= 0
            if nn(0) = 1
                pp <= pp * xx

            xx <= xx * xx
            nn <= nn/2
```

→ OR SOME SORT OF

0 & nn(nn'left downto 1) ; nk.

write VHDL for 8-to-1 bit MUX.

ie. 8 2-bit inputs
1 2-bit output

```
entity MUX is
    port (
        IN a,b,c,d,e,f,g,h: std_logic_vector(1 downto 0);
        IN sel : std_logic_vector(2 downto 0);
    ) OUT outm: std_logic_vector(1 downto 0)
end entity;

architecture behavioral of MUX is
--signals go here
begin
    process(sel)
    --variables go here
    begin
        outm <= a when "000",
                b when "001",
                c when "010",
                d when "011",
                e when "100",
                f when "101",
                g when "110",
                h when others;
    end process;
end behavioral;
```

this is combitional so all inputs need to be here -2

with sel select

forgot this review syntax!

wrong syntax
when "000" => outm <= a; -1

write VHDL of 3-input, 8-output decoder

ex. "000" in, "00000001" out
"100" in, "10000000" out

↳ set 3-bit input as x
set bit x of output to 1 (asserts)
set set all other bits to 0 (deasserts)

```
entity decoder is
    port( x_in : std_logic_vector(2 downto 0);
          x_out : out std_logic_vector(7 downto 0)
    );
end decoder;
architecture behavioral of decoder
begin
  process(x)
  variable x_int : integer;
    begin
      x_int := to_int(unsigned(x));
      x_out <= "00000000";
      x_out(x_int) <= '1';
  end process;
end behavioral;
```

probably safer to just hardcode it

write VHDL for table:

E_N	w_1	w_0	y_0	y_1	y_2	y_3
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1
0	x	x	0	0	0	0

entity ....

architecture ...

~~if EN='0' then~~

~~y <= "0000";~~

~~else~~

~~y <=~~

if (EN = '1') then

y (to_int (unsigned(w))) = '1';

end if;

↑ again

Review Videos

Combinational logic

- ↳ outputs only depend on current inputs
- ↳ uses only basic gates, no flip-flops
  - ↳ toggle output w/ each pulse

```
entity XOR_GATE is
    port( A,B : in bit;
          x : out bit)
end XOR_GATE
```

describes i/o

```
architecture behavioural of XOR_GATE is
    signal int1, int2 : bit;
begin
    int1 <= a and not b;
    int2 <= b and not a;
    z <= int1 or int2;
end behavioural;
```

describes behaviour

← multiple assignments happen concurrently here

(could also just be "z <= a xor b")

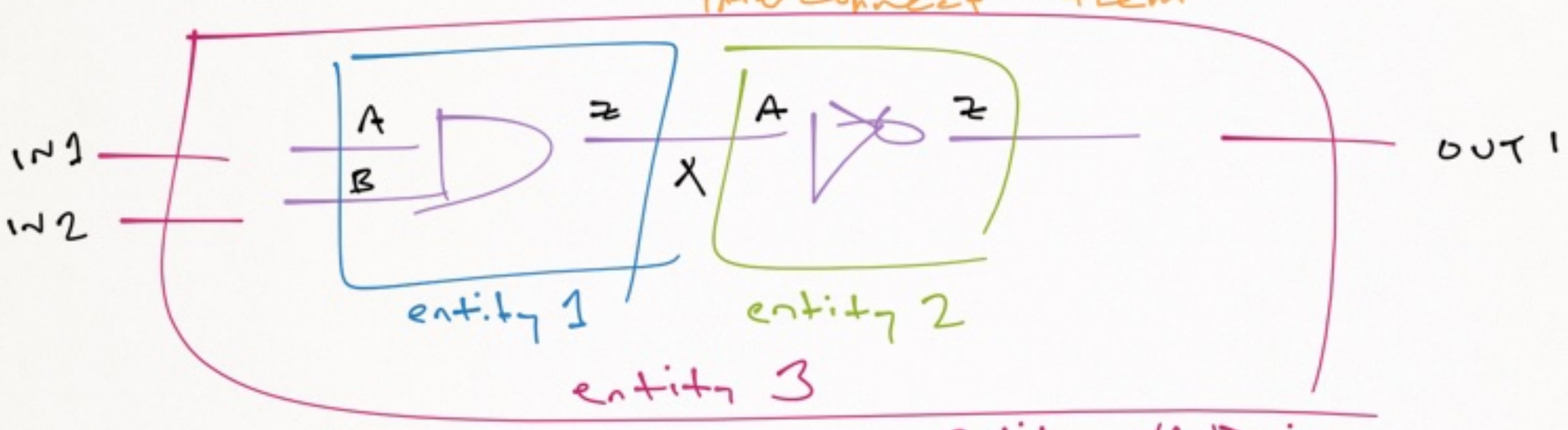
most real-life engineers design hardware using a description language, not schematics

signals
↳ only have life inside architecture

ports
↳ cannot assign inputs values
↳ cannot use outputs in logic
} use signals instead
↳ can also use inout types, but that is bad practice at this point

Structural Specification

VHDL, hierarchy is very important
↳ build things as entity levels and interconnect them




```
entity AND is
  ...
architecture of AND
  ...

entity NOT is
  ...
architecture of NOT
  ...
```

```
entity NAND is
  ...
architecture Behav of NAND is
  component AND
    port(A,B : in bit;
         Z : out bit)
  end component
```

```
component NOT is
    port(A: in bit;
         Z: out bit);
end component;

signal x: bit;

begin ← arbitrary names as long as unique
  label1: AND
      port map(IN1, IN2, X);
  label2: NOT
      port map(X, OUT1);
end Behavioural;
```

these declared in NAND entity ∄ NAND signals

this is like joining components in a schematic

also kind of like functions

using positional notation.

can also use name notation:

```
port map(A => IN1,
         Z => X,
         B => IN2)
```

better practice ↗ ∄ more robust

in industrial designs there can be ~30 levels of hierarchy

Buses

```
port(A,B : in STD_LOGIC_VECTOR(7 downto 0);
     Z : out STD_LOGIC_VECTOR(7 downto 0))
```

↳ like array

can do logic like bitwise and

```
Z <= A,B        -- here each bit is and'd w/ each corresponding bit
Z <= "01000111"
Z(5 downto 1) <= "010"
Z(7) <= '1'
```

2D-ARRAYS

```
type REGARRAY is array(7 downto 0) of bit_vector(3 downto 0)
signal R: REGARRAY;
```

Attributes

```
signal NAME : bit_vector(7 downto 0);
```

instead of

```
blah <= NAME(7);
```

you can use

```
blah <= NAME(NAME'high);
```

instead of

```
blah <= NAME(7 downto 0);
```

you can use

```
blah <= NAME(NAME'range);
```

others: left, right, high, low, range, reverse_range, event

never use BIT and BIT_VECTOR

↳ always use STD_LOGIC & STD_LOGIC_VECTOR

These also have, in addition to '1' & '0',

Z → no other driving signal

X → unknown value

U → uninitialized

```
library IEEE;                                  ← library
use IEEE.std_logic_1164.all;                   } packages for std_logic
use IEEE.std_logic_arith.all;
```

```
use IEEE.numeric_std.all;
```

→ this lets you use unsigned type which is like std_logic_vector except you can do mathematical operations & convert to ints very easily.

PROCESS

→ allows us to describe behaviour of circuit w/o describing actual hardware

```
process( sensitivity_list )  -- process runs when any
                             -- of the signals in this
                                change
  begin
    software-like statements
    executed sequentially
end process;
```

FUNDAMENTAL DIFFERENCE B/W HARDWARE/SOFTWARE

↳ hardware is elements operating in parallel

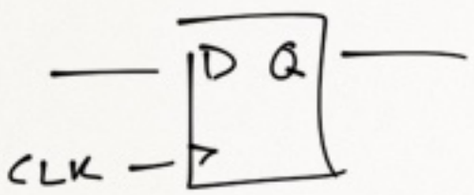
↳ software is statements operating sequentially

NOTE: can only use if/elses & case statements inside processes

assignments such as X <= A xor B inside architecture are actually processes in and of themselves

Can't make entire design a process, assignments only assigned at the end of a process.

D flip flop



↳ memory element

whenever clk goes from 0→1
the value of D is copied to Q
—> and stays until next clk rising edge

```
entity DFF is
    port(D, clk: in STD_LOGIC;
         Q : out STD_LOGIC);
end DFF;

architecture BEHAV. of DFF is
begin
    process(clk)
    begin
        if rising_edge(clk) then
            Q <= D;
        end if;
    end process;
end BEHAV.
```

not assigned a value otherwise which implies storage

When writing VHDL, you are only describing behaviour. The FPGA does not execute VHDL, the synthesis tool maps the VHDL to gates, and the FPGA uses those gates.

↳ field-programmable gate array

Asynchronous/Synchronous Resets

↓ implemented immediately on change

↓ implemented only on clock change

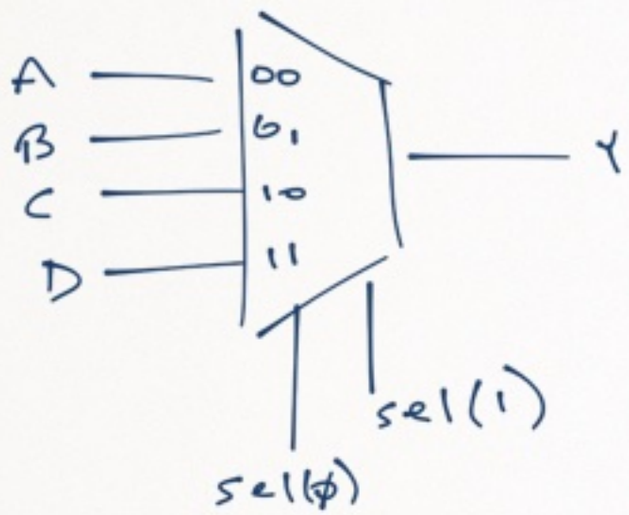
↳ process (clk, reset)

...

:P reset = '1' then

Q <= '0';

multiplexer




```
entity mux
...

architecture behav of mux is
begin
   process(sel, A,B,C,D)
   begin
      case sel is
         when "00" => Y <= A;
         when "01" => Y <= B;
         when "10" => Y <= C;
         when others => Y <= D;
      end case;
   end process;
end behav;
```

note all inputs go in sensitivity list

what should go in sensitivity list?

combinational logic ⟶ put inputs in sens. list

flip flop / sequential logic ⟶ put clk in sens. list

```
process( A,B)
  Z<=A and B and C
```

⟶ get VHDL process statement warning for this combinational error

```
process(clk,D)
  if clk='1' then
    Q<=D
```

⟶ get "inferred latch" warning for this sequential logic error

this is not a flip-flop b/c d should not change when q changes

THIS IS NON-SYNTHESIZABLE AND WILL BE A REAL PROBLEM

NOTES FROM LECTURE SLIDES

Do not create circular dependencies

ex. `X <= A and B`
`A <= X or C`
} there are assigned simultaneously so this code is nonsense

↳ also, you cannot use the values of an out signal for logic and cannot write to the value of an in signal → use internal signals

Synchronous Reset

```
process(clk)
begin
    if rising_edge(clk) then
        if reset = '1' then
            Q <= '0';
        else
            Q <= D;
```

← Do not add any more conditions in these statements
↳ it won't be synthesized correctly

Asynchronous Reset

```
process(clk, reset)
begin
    if reset = '1' then
        Q <= '0';
    elsif rising_edge(clk) then
        Q <= D;
```

this can only be 1 or Ø, can't be dependent on any other signals

again, no more logic allowed here

rising_edge(clk) == clk'event and clk='1'

↳ checking if clk='1' isn't enough

CAUTIONS:

```
architecture ...
    signal A : std_logic;
    ...
begin
    A <= B;
    A <= C;
end ...;
```

cannot do this! do not assign a value to a signal or output more than once in more than one process

same with

```
process(B)
begin
    A <= B;
end process;

process(C)
begin
    A <= C;
end process;
```

same, don't do this.

EVEN IF THE CONDITIONS ARE MUTUALLY EXCLUSIVE

```
process
    if input='1'
        A <= B;
process
    if input='0'
        A <= C;
```

just don't do it

however, you can assign a signal multiple values within a single process.

ex.
```
process(A,B,C)
begin
  A<='0';
  if C='1' then
    A<=B;
```

only the last assignment made will take effect

↳ order matters inside process

NOTE

changes to signal or output only take effect at the end of a process.

if you keep thinking like a programmer, this will fuck you over.

ex.
```
process(A,B,C)
begin
  A<='0';
  C<=A;
  if C='1' then
    A<=B;
  end if;
end process;
```

this assignment is only made at the end of the process

so at this point, C hasn't actually been given the value of A yet

so the value of C we're comparing is the old value.

remember this

↑ end of process

STATE MACHINES

```
architecture BEHAV of FSM is
begin
   process (clk)
      variable STATE : std_logic_vector(1 downto 0) := "00";
   begin
      if rising_edge(clk) then
         case STATE is
            when "00" => STATE := "01";
            when "01" => STATE := "10";
            when "10" => STATE := "11";
            when others => STATE := "00";
         end case;

         STATE_SIGNAL <= STATE;
      end if;
   end process;
end BEHAV;
```

↑ output of this entity

↑ we can initialize variables to default val. on startup

variables only have lifetime inside of a process.

they are also updated immediately inside the process (unlike signals)

for readability, you could define enumerated types:

```
type state_type(live, wait, sample, display);
variable STATE : state_types;

....

case STATE is
    when live => ...
    when wait => ..
    when sample => ..
    when display => ..
```

MOORE MODEL STATE MACHINE

```
process(clk)
   variable PRESENT_STATE := "00";
   variable NEXT_STATE;
begin
   case PRESENT_STATE is
      when "00" => if input='0' then
                      NEXT_STATE := "00";
                   else
                      NEXT_STATE := "01";
                   end if;
      when "01" => ... etc ...
   end case;

   PRESENT_STATE := NEXT_STATE;

   case NEXT_STATE is
      when "00" => z <='0';
      when "01" => z<='1';
      ...
      ...
   end case;
  end if;
end process;
```

← or you know, whatever logic drives the state machine

} or whatever this output should be.

Circuit that counts every clock cycle

```
IEEE ...
entity ... clk in, vector out
architecture behav of counter is
   signal count: unsigned(7 downto 0);
begin
   if rising_edge(clk) then
      count <= count + 1;
   end if;
end process;

outsignal <= std_logic_vector( count );
end behav.
```

VHDL FLOW

code + testbench → modelsim → synthesize → gates → sell → dolla dolla bills

this is because as it stands, no one can yet convert C or something to pure hardware

any CPSC hack can write C, it takes an ECE whiz to make hardware, which can run faster and use less power, but is often more expensive.

ex. iPads can only watch certain video types, this is because they decode straight in hardware, which means you can watch more hours.

→ but also more expensive

All Process Must Be One Of 3 Types

Type 1: PURELY COMBINATIONAL

Type 2: SEQUENTIAL w/ CLOCK

Type 3: SEQUENTIAL w/ CLOCK, ASYNC RESET

If it's not, who the hell knows if your VHDL is synthesizable

Purely Combinational

→ outputs function only of current inputs

→ no memory, no fucking around

ex.
```
process(A, B, sel)
    if sel = '0' then
        Z <= A;
    else
        Z <= B;
```

RULES

1. all inputs in sensitivity list → ex. sel not being in sens. list
2. output assigned value for all input combinations.
   → ex. no else statement

Purely Synchronous

→ each output changes only on clk edge

ex.
```
process(clk) begin
    if rising_edge(clk) then
        Z <= A and B;
    end if
```

RULES

↗ violations of this cause inferred latch after

1. only clk allowed in sensitivity list
2. only signals that change on same edge of the same process should be a part of the same process

Synchronous w/ Async Reset

→ reset condition outside of rising_edge condition

ex.
```
process( clk, reset)
begin
    if reset = '1' then
        Z <= '0';
    elsif rising_edge(clk) then
        Z < A and B;
    end if;
end process;
```

RULES

1. only clk/reset in sens. list
2. can only assign '1' or '0', can't depend on any other signals
3. need rising_edge clause

FOR REAL.

All processes must be one of these. No exceptions, don't fuck around or debugging your VHDL is going to be a world of pain and you're going to get fired.

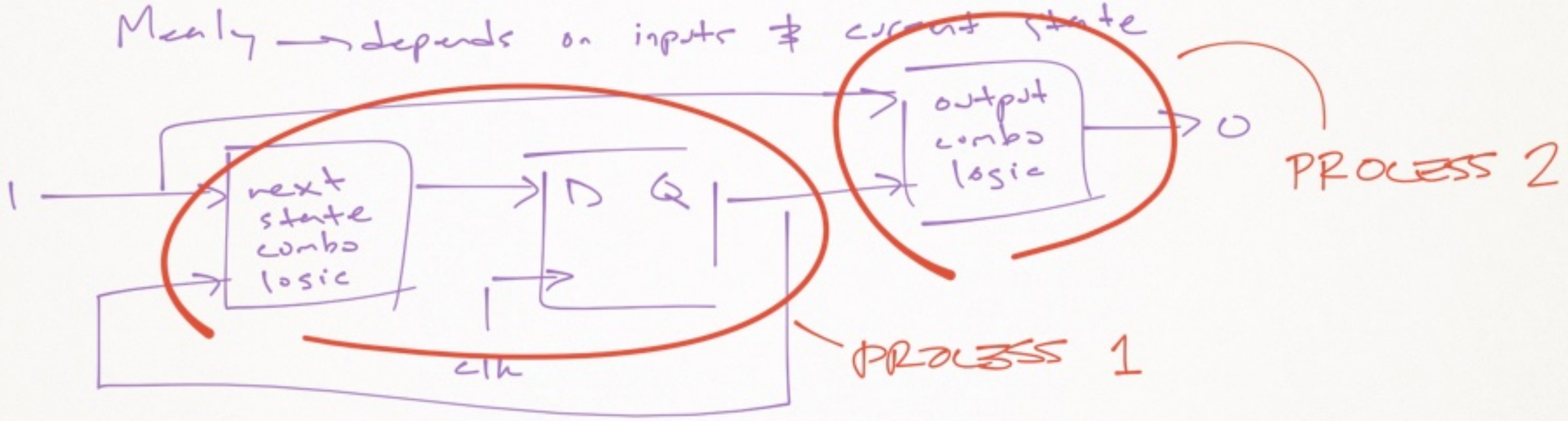
MORE ON STATE MACHINES

heh

Moore → can do in 1 process
Mealy → have to use 2 or 3 processes

Moore
↳ outputs depend only on current state
ex. if rising_edge(clk)
case PRESENT_STATE is
when "00" ...
...
...
end case;
Z <= PRESENT_STATE;

Mealy → depends on inputs & current state



MEALY → signal current_state

```
process(clk)
begin
   if rising_edge(clk)
      case current_state is
        when "00" => if whatever
                        current_state <= "01";
                     else current.state<='00';

      end case;
   end if;
end process;
```

```
process(current_state, inputs)
begin
   output <= function of inputs & current state
end process;
```

all in same architecture

NOTE:

When writing a state machine, use a variable as the current_state, because it will be updated when you assign it at the end.

if you use a signal, it will be updated at the end of the process and it won't be up to date when you set the output.

see slide set 5 pg. 10

IN SHORT: TRY NOT TO USE √ SIGNALS TO COMMUNICATE WITHIN PROCESS, USE A VARIABLE.

Variables updated immediately signals not.

signal

```
Z <= A or B
X <= Z and B
```

↑ not updated

variable

```
Z := A or B;
X <= Z and A;
```

↑ updated

weird feature of VHDL

```
W <= X
X := Z
```

even though W is assigned at the end of the process and the variable is changed immediately, W gets the old value of X, before it's given Z

ex.
```
X := '0';
W <= X;
X := '1';
```
} W gets '0' because order is accounted for

BUT JUST DON'T WRITE CODE LIKE THIS

what if no sensitivity list?

```
process
begin
   ...
end process;
```

basically an infinite loop.
↳ re-executes immediately upon finishing

an equivalent to

```
process(A)
   ...
end process;
is
process
  wait until A' event
   ...
end process;
```

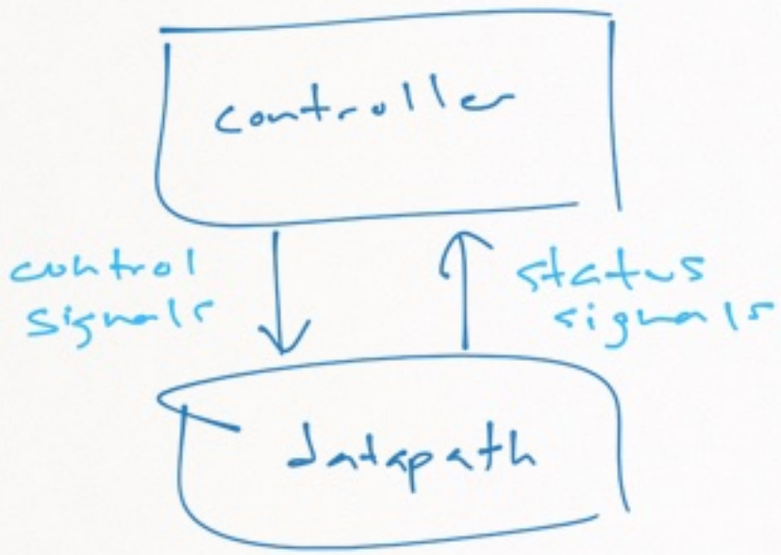
wait isn't synthesizable though.

Usually wait is used for testbenches

```
process
begin
   ... do something cycle 1
   wait until rising-edge(clk)
   ... do something cycle 2
   wait until rising-edge(clk)
   ... etc
```

level-sensitive latches are like flip-flops, but go off clk='1' not rising_edge(clk)

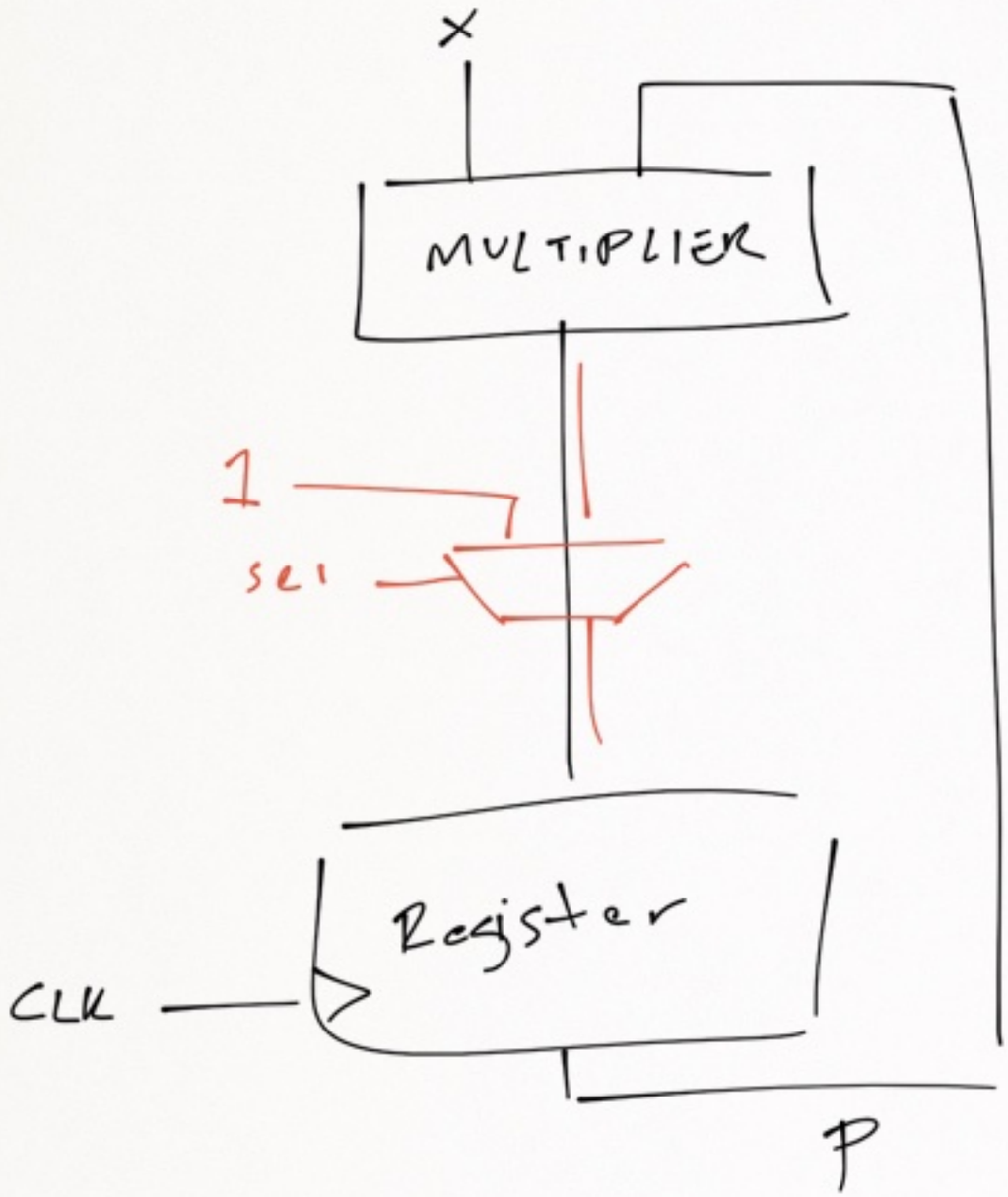
systems have 2 parts
→ 1. controller
→ 2. datapath



Problem Build controller that calculates X^A where X and A are both inputs.

Algorithm:
```
P = 1
CNT = A - 1
while ( CNT >= 0 )
    P = P * X
    CNT = CNT - 1
```

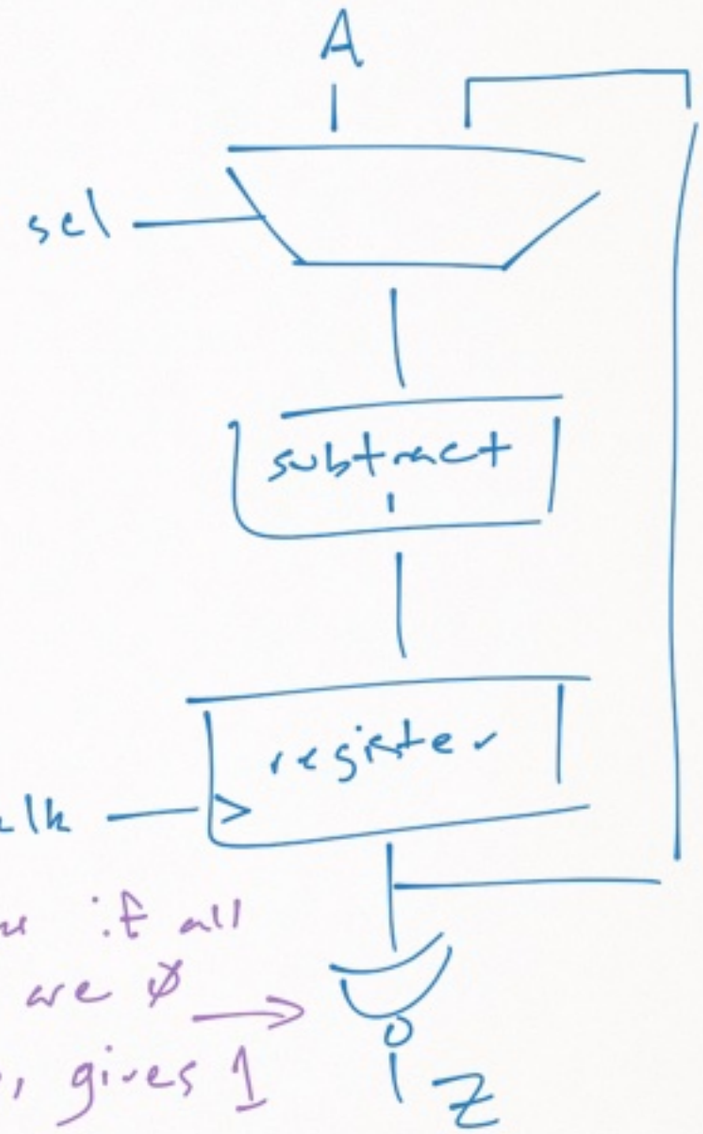
DATAPATH



This will run indefinitely

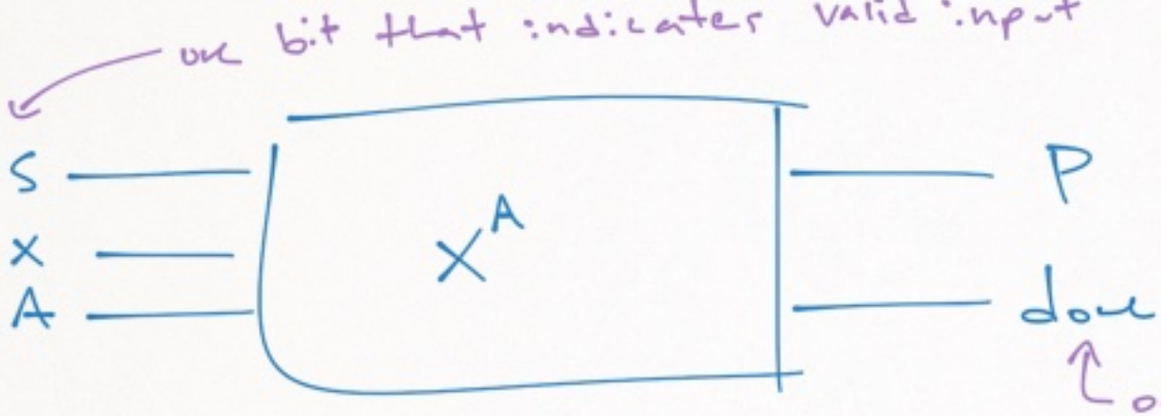
This is to initialize P on first cycle.

This is to count down multiplying for A cycles



Exponential-er

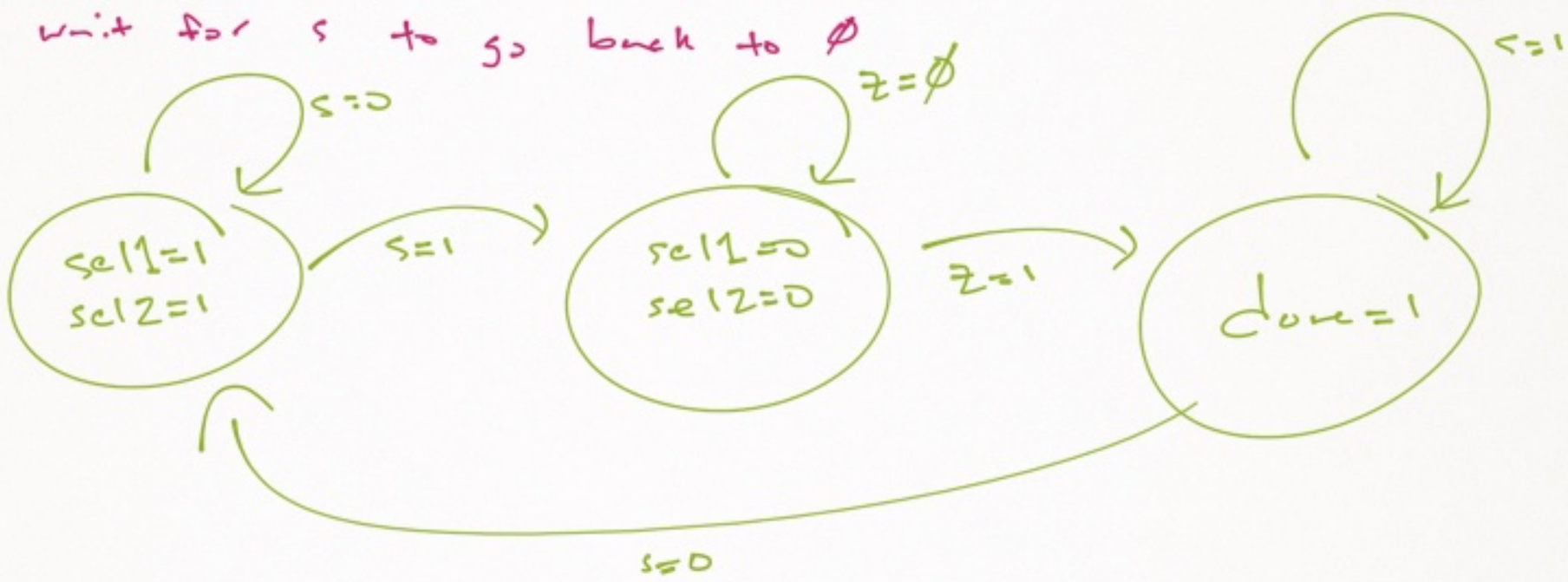
STATE MACHINE

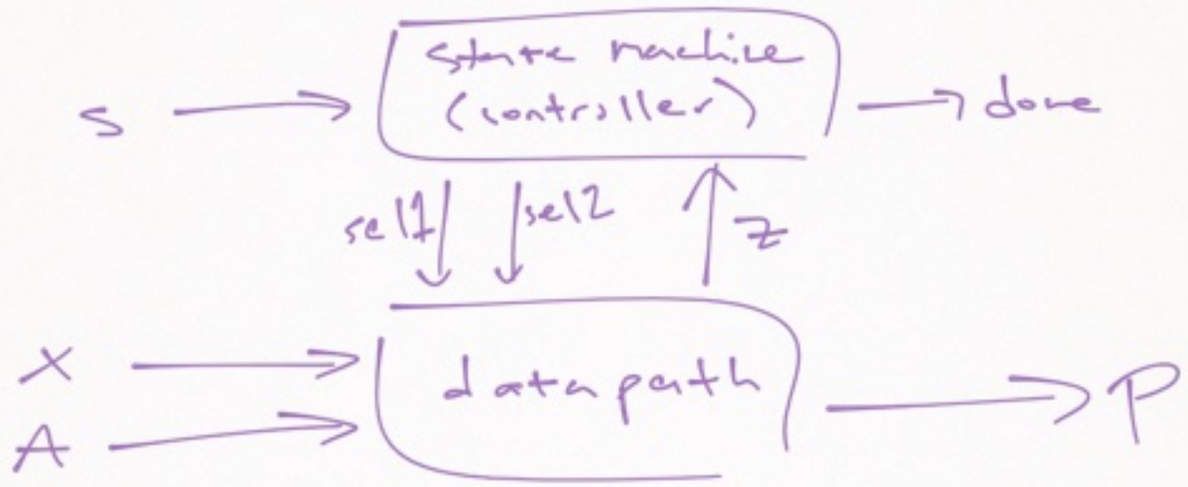


when S goes to 1
- initialize p to 1
- initialize CNT to A

} (i.e. sel1 & sel2 = 11)

- what until CNT = 0 (i.e. Z = 1)
- set done to 1
- wait for S to go back to ∅





datapath VHDL

```
process(clk)
  variable temp : unsigned(15 downto 0);
begin
  if clk='1' then
    if sel1='1' then
      P_int <="0000001";
    else
      temp := X * P_int;
      P_int <= temp(7 downto 0);
    end if;
  end if;
end process

P <= P_int;
```

```
process(clk)
begin
  if clk='1' then
    if sel2='1' then
      CNT <= A-1;
    else
      CNT <= CNT-1;
  end if;
end process;

process(CNT)
begin
  if CNT="00000000" then
    z<='1';
  else
    z<='0';
  end if;
end process;
```

state machine VHDL (controller)

```
process(clk)
begin
   if clk='1' then
      case  curr_state is
        when "00" =>
             if s='0' then
                curr_state <="00";
             else
                curr_state <="01";
             end if;
        when "01" => . . .
                     . . .
      end case;
   end if;
end process;

process(curr_state)
begin
   case curr_state is
      when "00" =>
           sel1<='1'; sel2 <='1'; done<='0';
      when "01" =>
           sel1='0'; sel2<='0'; done<='0';
      when others =>
           sel1 ='0'; sel2<='0'; done<='1';
   end case;
end process;
```

QED. hardware is more work then software.

SLIDE SET 7,
PAGE 20

USE THIS
SPACE FOR
THE BIT
COUNTING
EXAMPLE

Types

std_logic
std_logic_vector
boolean
character
string
int (32b signed)
real (floating point) NOT SYNTHESIZABLE

do more notes here

Practice

combination lock

problem w/ code?

```
...
process(sel, a)
begin
    case (sel) is
        when "000" => q(0) <= a;
        when "001" => q(1) <= a;
        ...
        when others => q(7) <= a;
    end case;
end process;
```

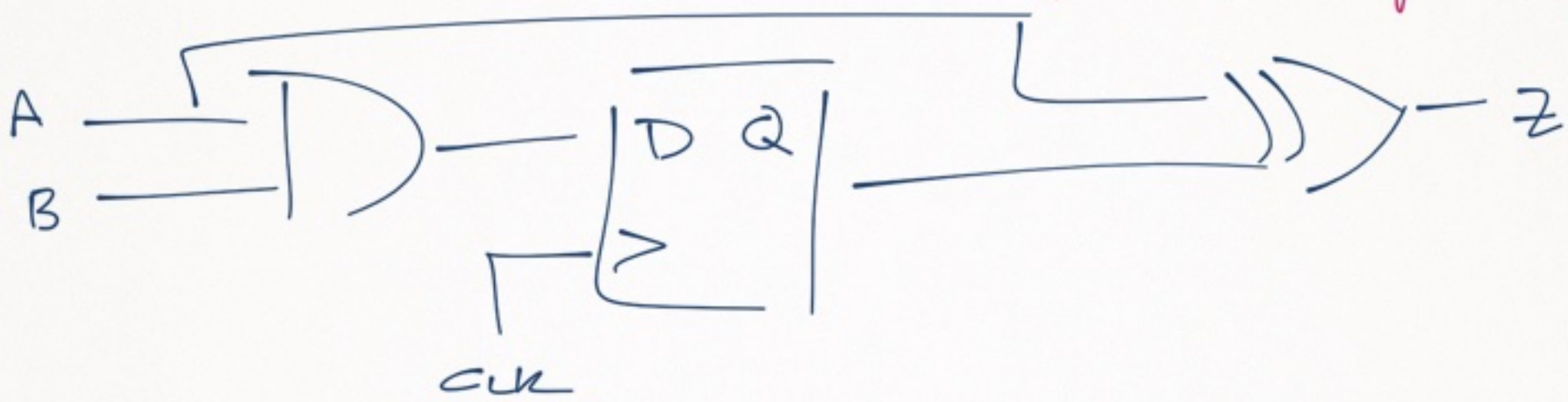
this is a purely combinational process, but the logic doesn't drive every other bit of q

WRITE VHDL FOR JK FLIP-FLOP

J	K	Q
0	0	previous value of Q
0	1	0
1	0	1
1	1	inverse of previous value of Q.

```
process(clk)
variable QQ in std_logic := '0';
begin
    if rising_edge(clk) then
      if J='0' then
        if K='1' then
          QQ:='0';
      else
        if K='1' then
          QQ:= not Q;
        else
          QQ:='1';
        end if;
      end if;
      Q<=QQ;
end process;
```

Write VHDL for circuit (if you think you can do it in one process, it's probably unsynthesizable.)



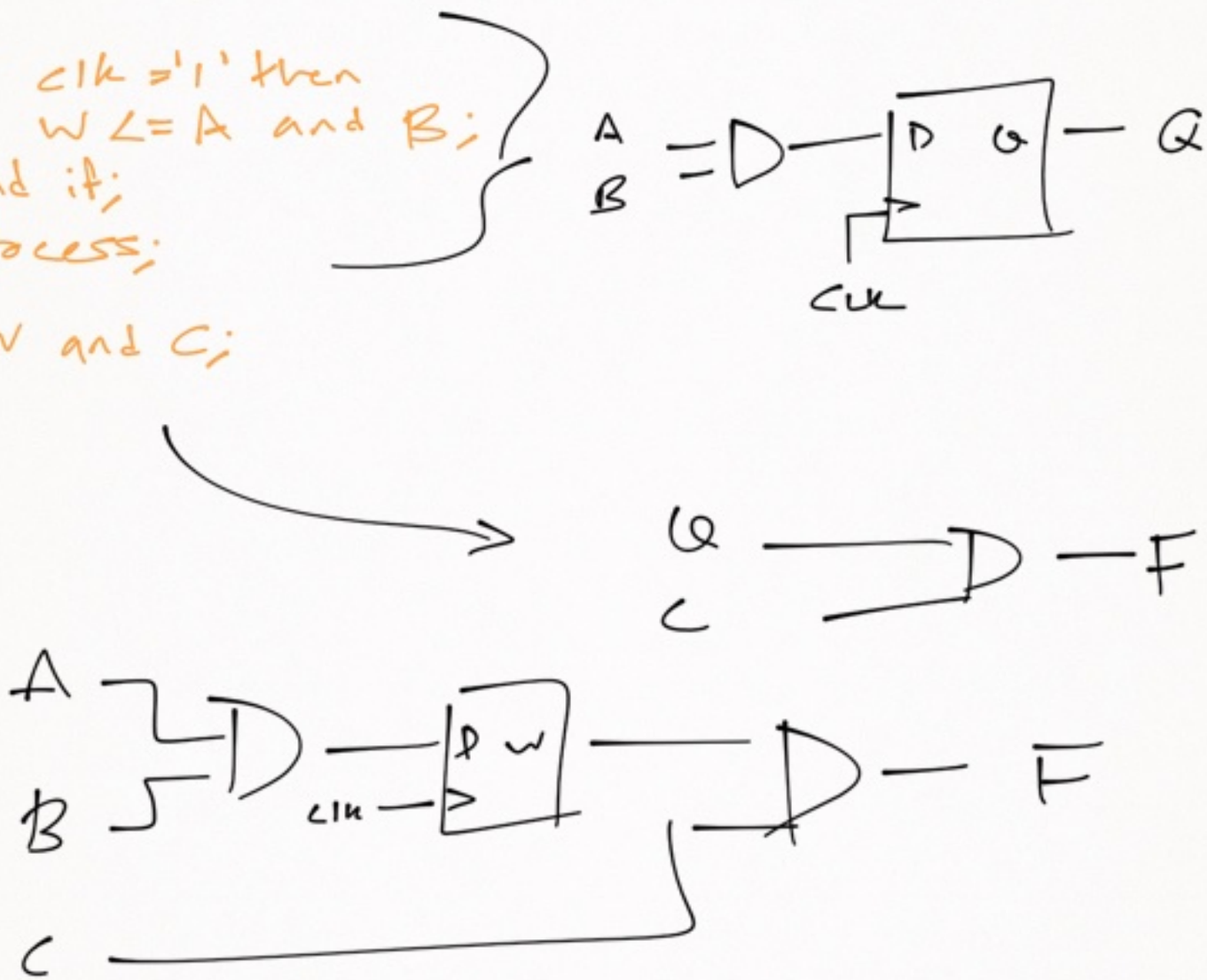
define Q as signal, blah blah

```
process(clk) --flipflop
begin
    if rising_edge then
        Q<=D: A and B;
    end if;
end process;
```

~~process(A, Q)~~
~~begin~~
    Z<=Q xor A;
~~end process;~~

Consider code, describe & draw circuit

```
A,B,C,CLK : in bit
F : out bit

signal W : bit

process(clk)
begin
    if clk='1' then
        W <= A and B;
    end if;
end process;

F <= W and C;
```

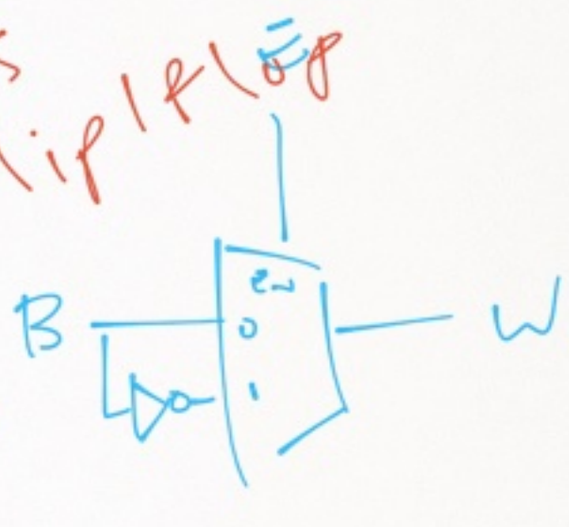


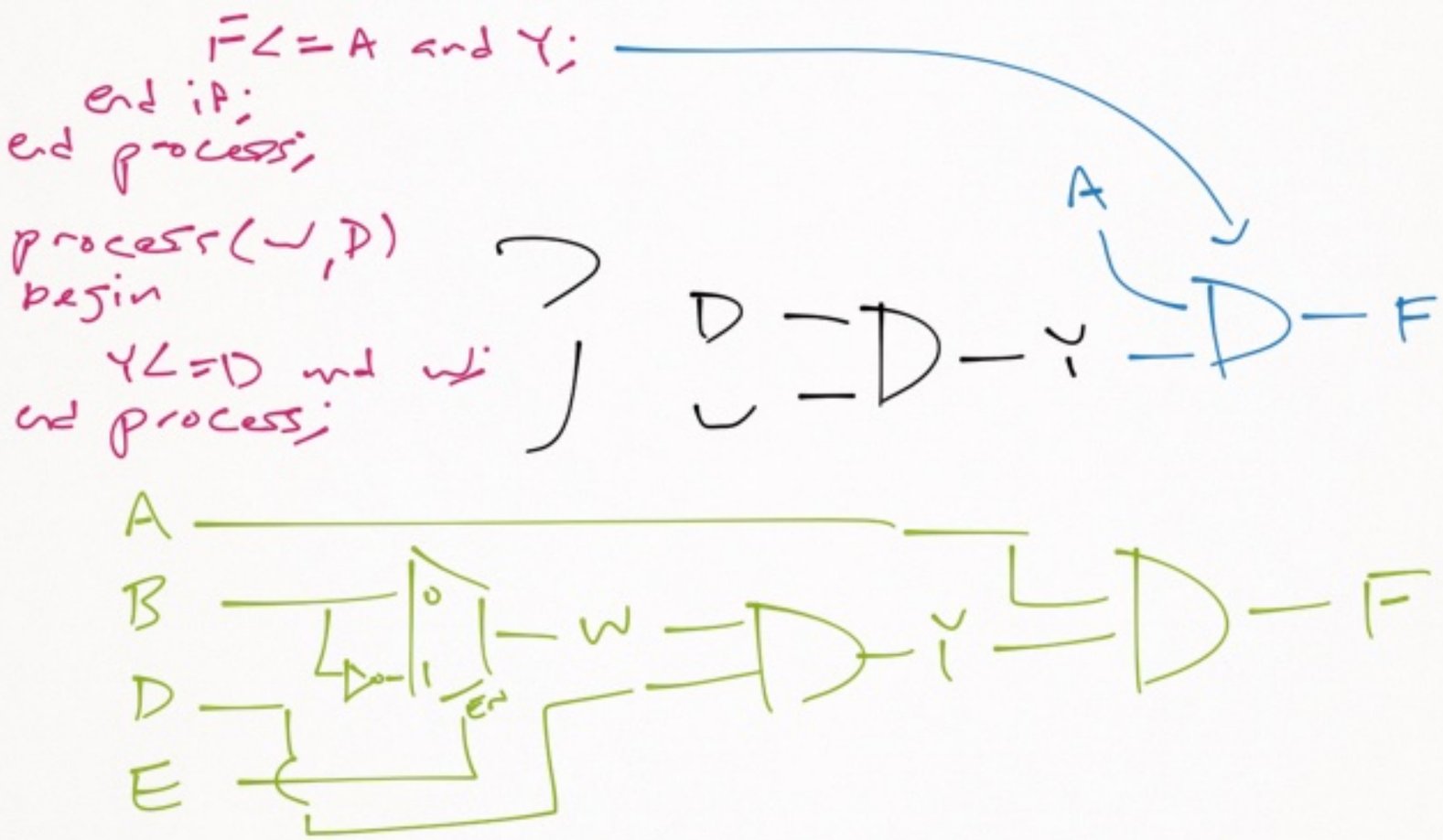
Draw schematic for following code

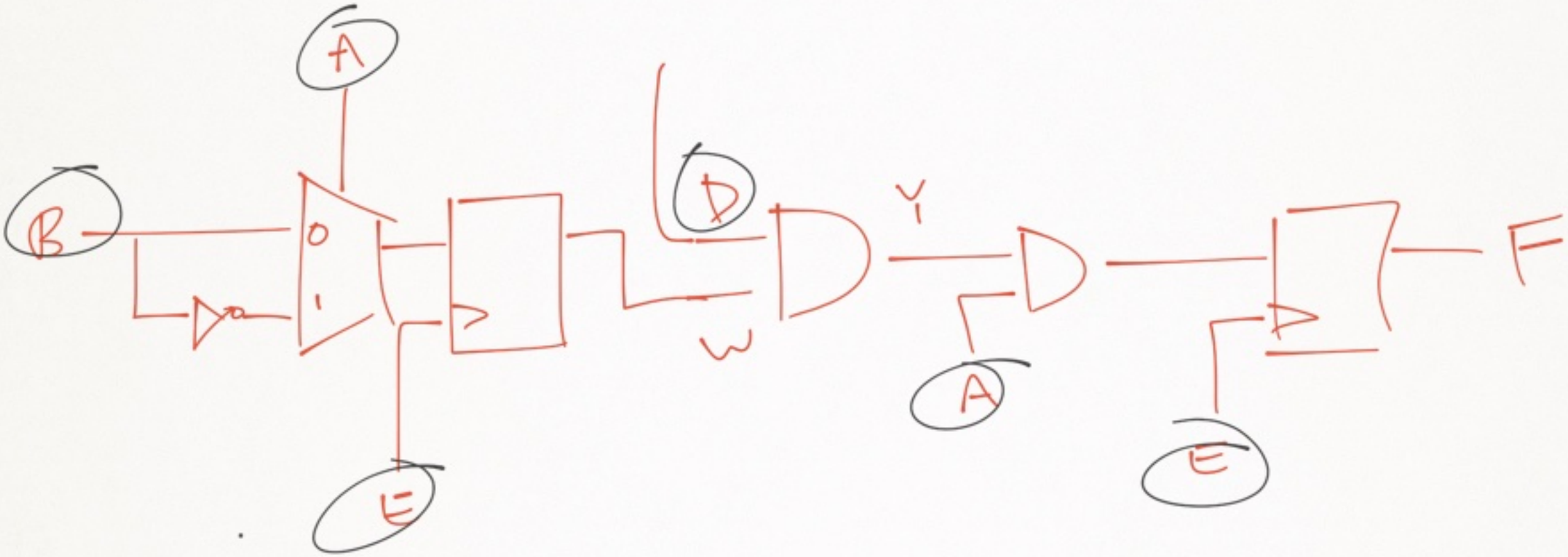
```
A,B,D,E : in bit
F : out bit

signal W,Y : bit

process(E)
begin
    if E='1' then
        case A is
            when 0 => W <= B;
            when 1 => W <= not B;
        end case;
        F <= A and Y;
    end if;
end process;

process(W,D)
begin
    Y <= D and W;
end process;
```

wrong, needs flipflop

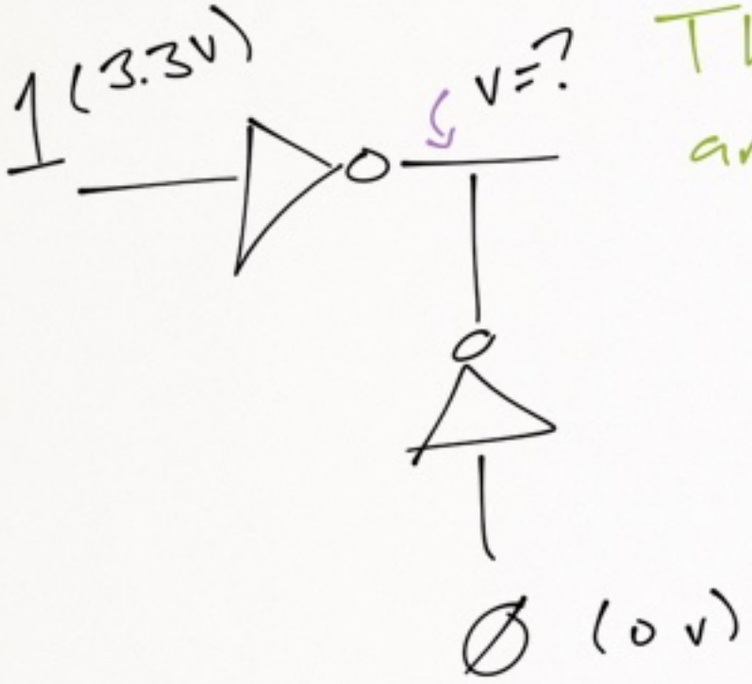
2






A
B
0
1
D
W
Y
A
E
E
F

Tri-States



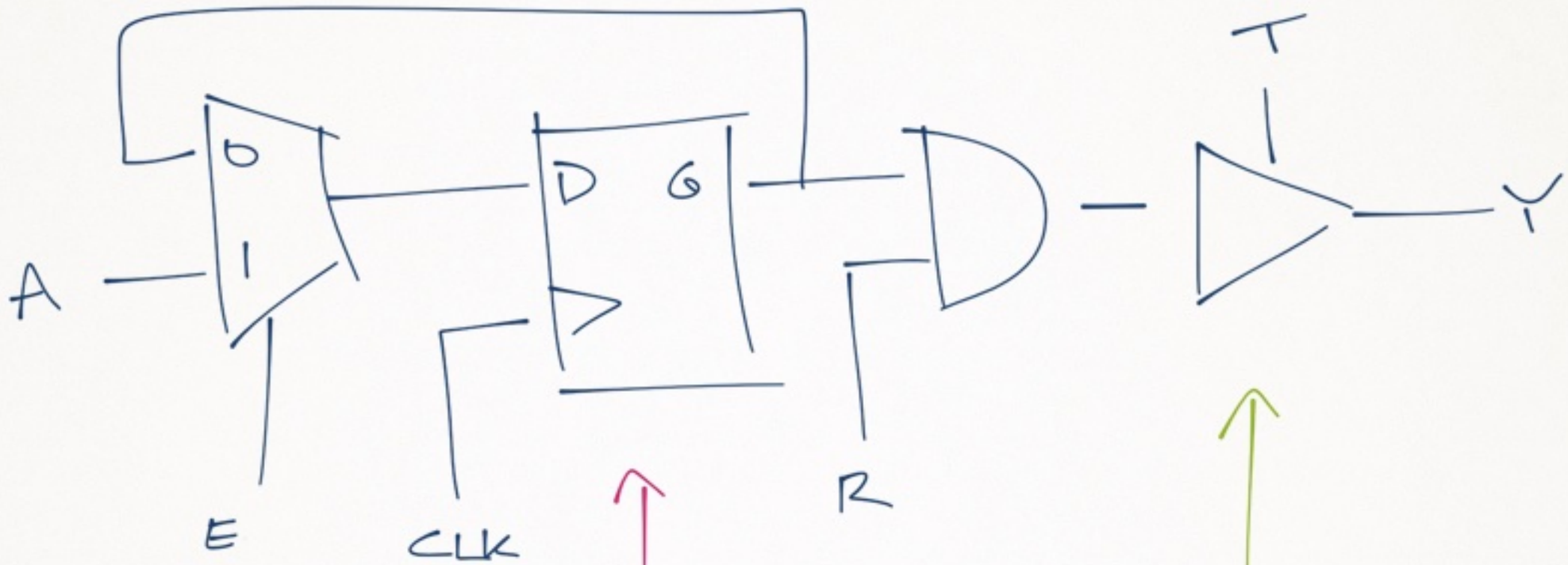
This is "fighting" and it damages chip

```
architecture
    r <= not a;
    r <= not b;
end
```

will cause "multiple drivers" error

if enable is 1
    out is driven by in
else
    out is not driven
    ↳ given value of 'Z' for high impedance

```
process(A, enable)
begin
  if enable = '1' then
    Q <= A;
  else
    Q <= 'Z';
```



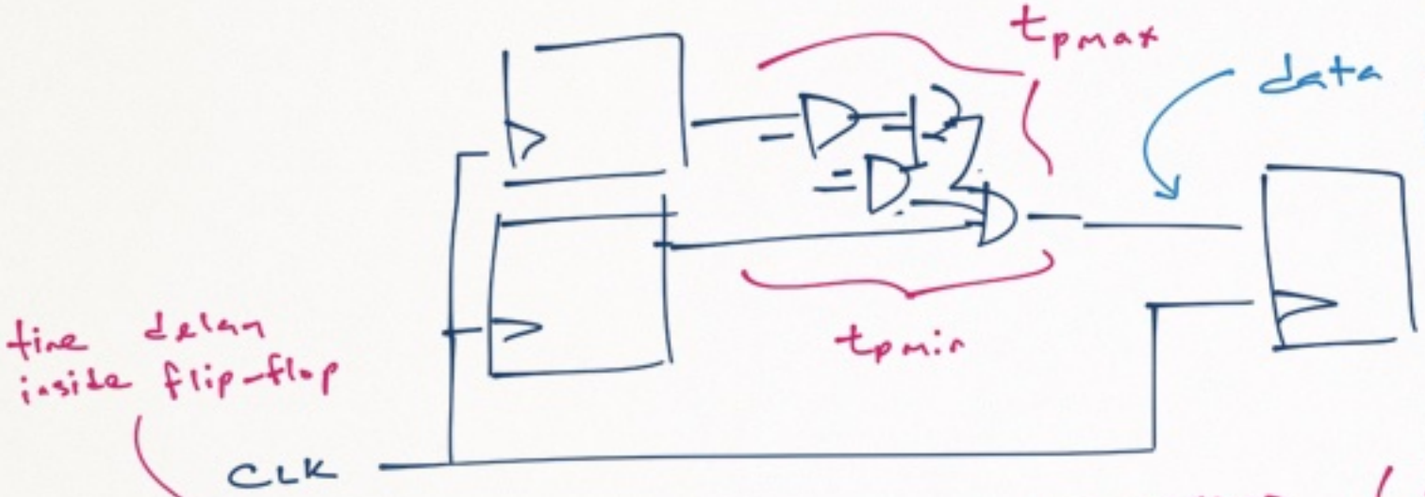

```
case E is
    when '1' => D <= A;
    when others => D <= Q;
end case;
```

```
if rising_edge(clk) then
    Q <= D;
end if;
```

```
if T='1' then
    Y <= R and Q;
else
    Y <= 'Z';
end if;
```

OCT 20 2014

WHAT'S THE FASTEST CLK WE CAN RUN CIRCUIT AT?



time delay inside flip-flop

PERIOD (time b/w clk edges)

1) $t_{clk\text{-}to\text{-}q} + t_{p_{max}} + t_{setup} \leq T$ ← if this isn't true, increase T (slow clk)

time going from clk_i to clk_{i+1}

maximum propagation delay (longest path)

2) ~~$t_{clk\text{-}to\text{-}q}$~~ $+ t_{p_{min}} \geq t_h$

minimum propagation delay (fastest path)

CAN'T FIX THIS BY CHANGING T, CAN ONLY INCREASE t_{pmin} (ex. —— to —▷o-▷o—)

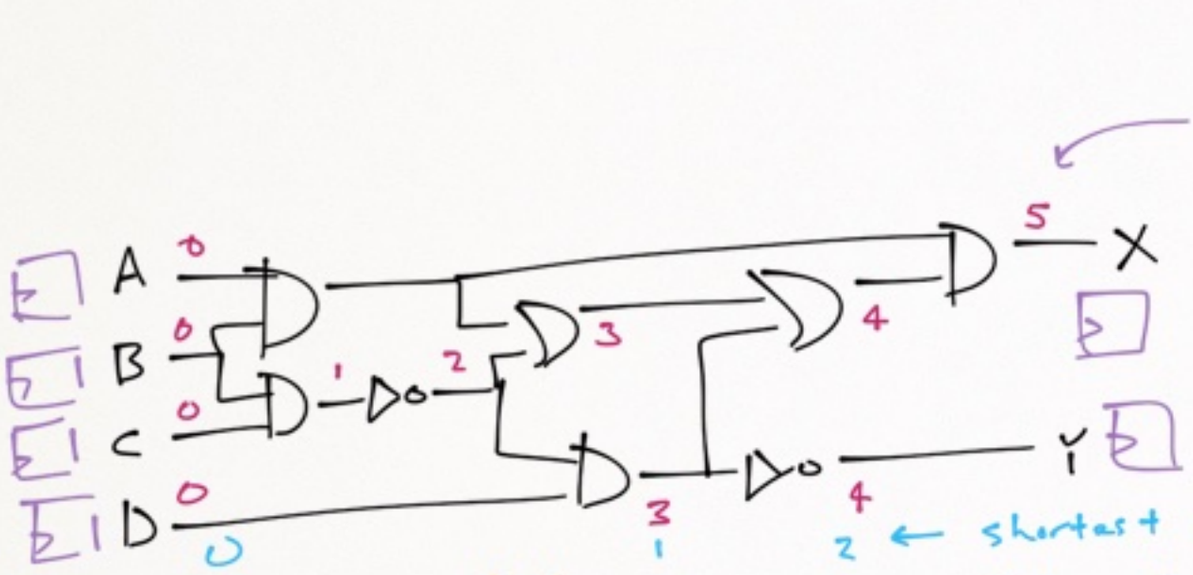
1) SETUP TIME
→ data races b/w clk_i to clk_{i+1}

1) & 2), IF THESE EQUATIONS ARE SATISFIED, YOUR CIRCUIT WILL WORK CORRECTLY

2) HOLD TIME
↳ b/w i & i
↳ close to ∅
↳ cannot easily fix

→ a time where you don't want the data to change, might be removed before read, can't fix by going faster

Find max freq. circuit can run at and what happens when breaks down



CRITICAL PATH ← largest delay is 5ns
so fastest frequency is < 200MHz
< b/c accounting for setup and clk-to-q time

← shortest delay

outputs like X might change multiple times in 5ns

NOW IMAGINE PUTTING FLIP-FLOPS ON ALL INPUTS + ALL OUTPUTS

If something works at a certain f, it will always run at lower f

overclocking PC usually violates equation 1), could actually result in incorrect arithmetic
↳ can't trust it

CRITICAL PATH
→ LONGEST PATH, USUALLY REGISTER TO REGISTER

QUARTUS
TIMEQUEST TIMING ANALYZER → SLOW 1200mV 85C MODEL (running hot, slightly pessimistic)
→ WORST-CASE TIMING PATHS

CARRY LOOKAHEAD ADDER (CLAs)

GROUP OF 4 BITS WE WANT TO ADD TOGETHER

propagate$_i$ = a_i xor b_i
generate$_i$ = a_i and b_i

$$t_{clk-to-q} + t_{p-max} + t_{su} < T$$

~~$t_{clk-to-q}$~~ + $t_{p-min} > t_h$

ASSUME TIME GIVEN IS MAX, ASSUME MIN TIME IS ZERO

$F_1 = \emptyset$

$F_2 = 1$

$F_3 = F_1 + F_2 = 1$

$F_4 = F_2 + F_3 = 2$

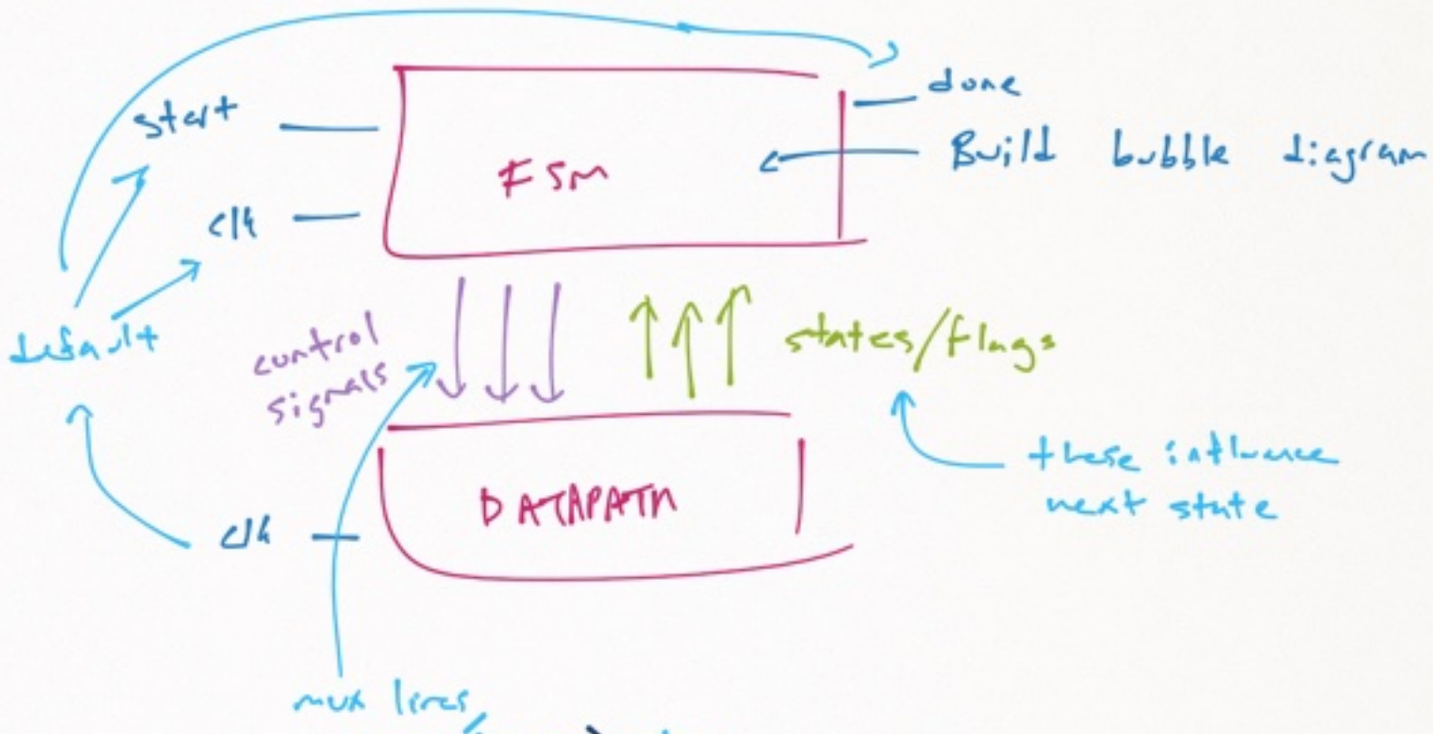
$F_5 = 1 + 2 = 3$

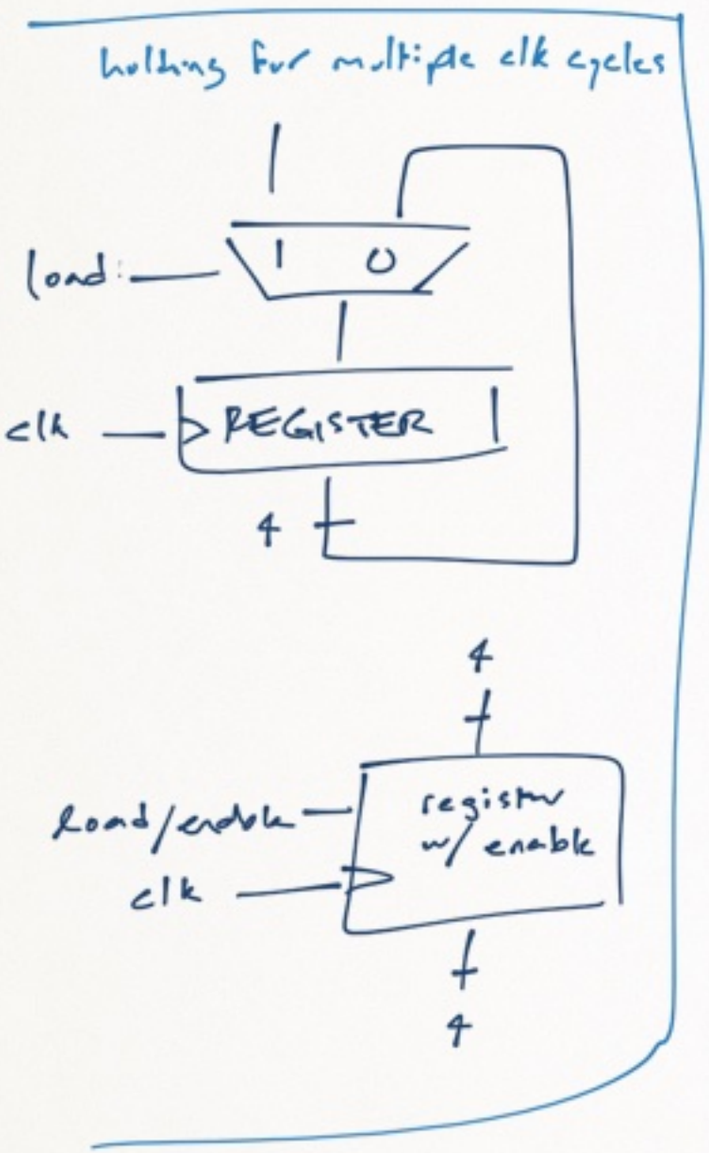
$F_6 = 2 + 3 = 5$

$n \geq 3$

counter
gates
register
adder
tri-state
mux
comparator

BREAK INTO DATAPATH + STATE MACHINE



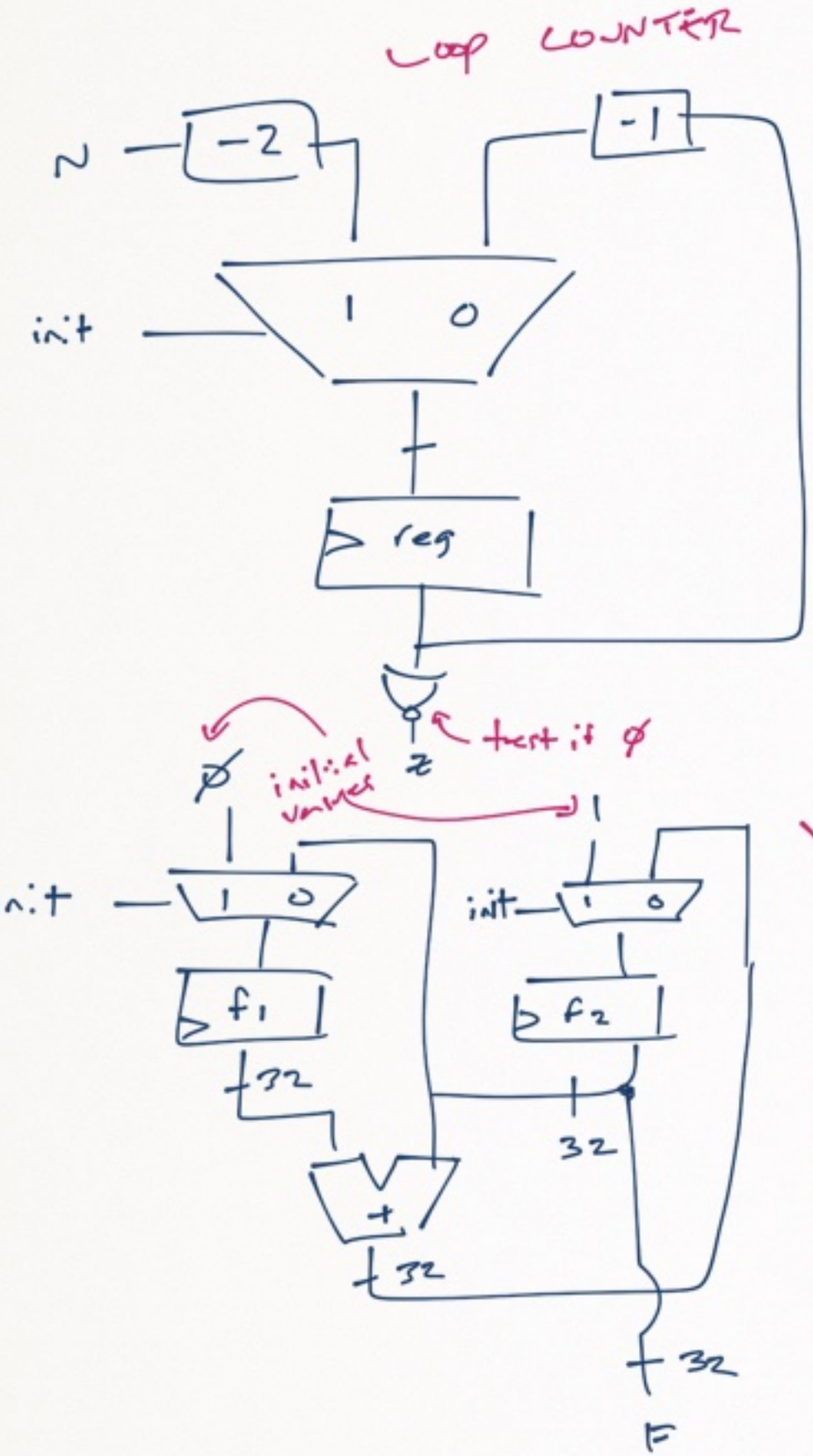


FIND ALGORITHM

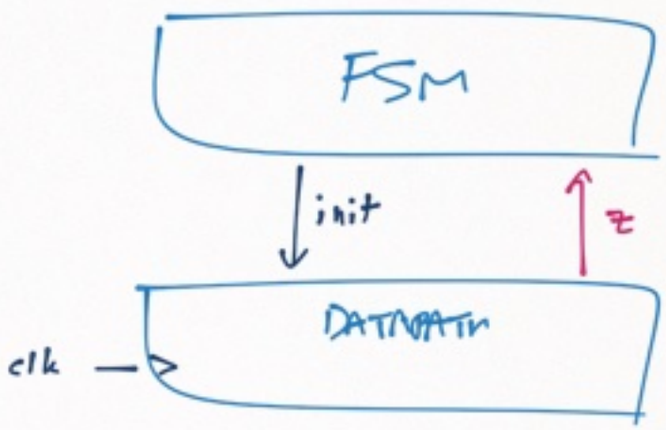
```
count = N-2
F1 = ∅ ; F2 = 1
while count != 0
    F = F1 + F2
    F1 = F2
    F2 = F
    count--
output F
```

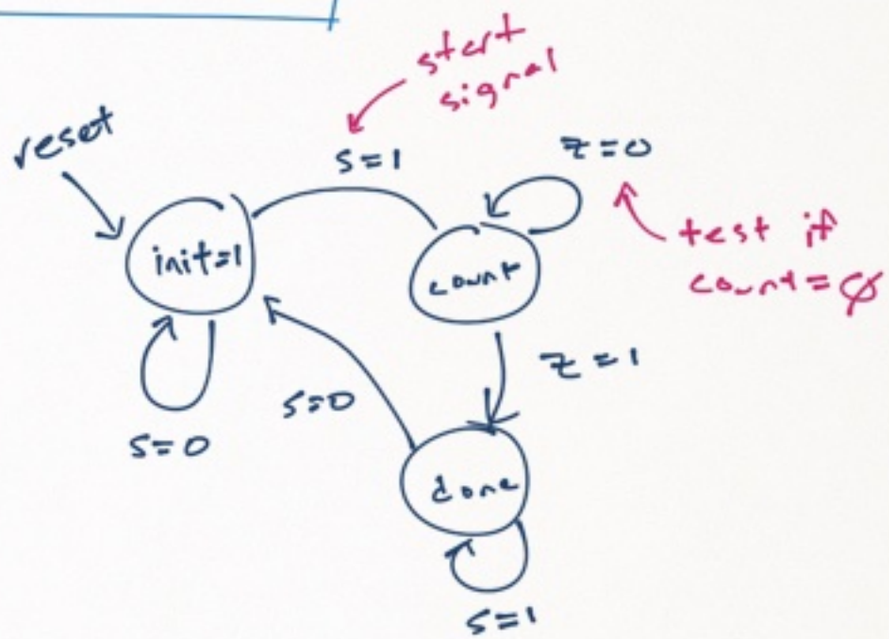
now find datapath from algorithm

DATAPATH



STATE MACHINE

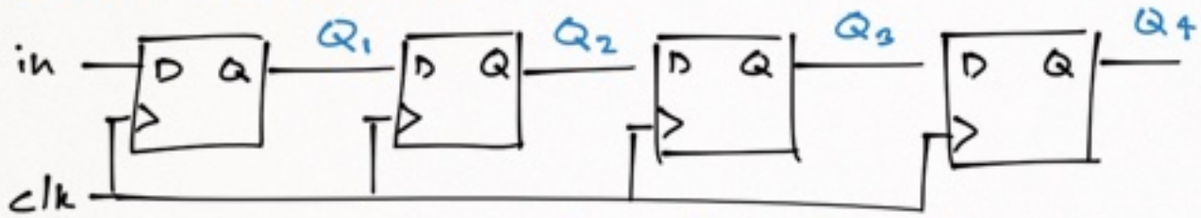




MIDTERM #2 REVIEW

COUNTERS

SHIFT REGISTERS

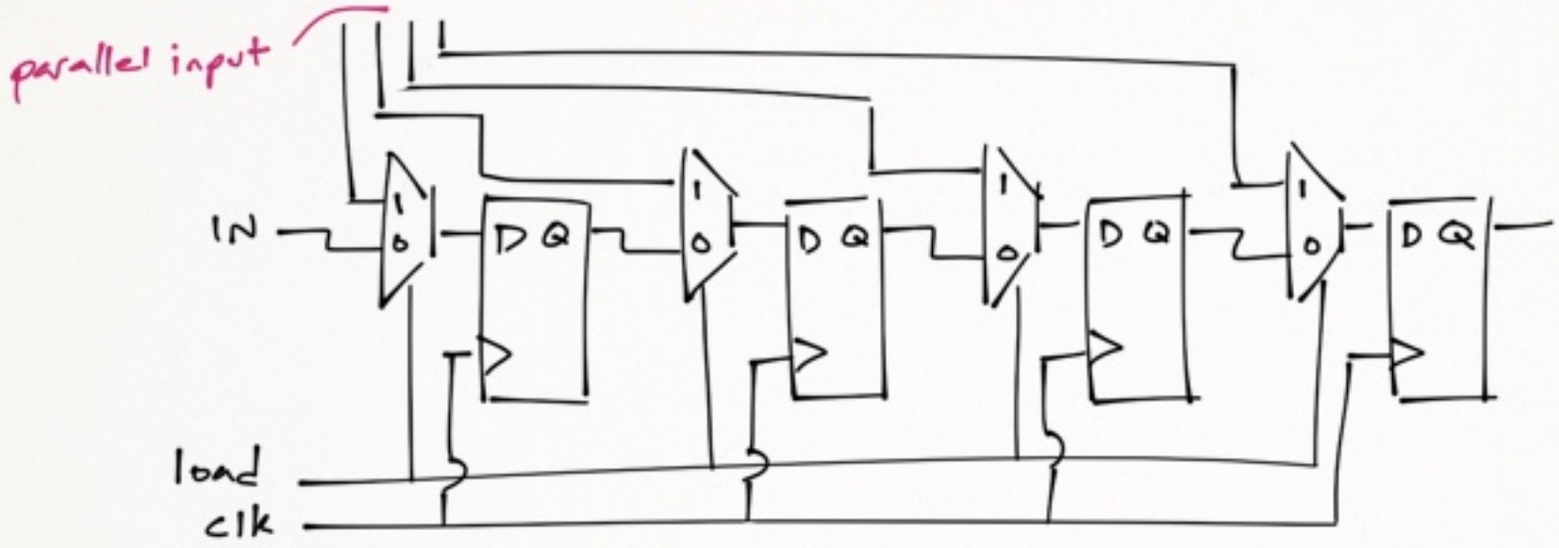


each cycle shifts the contents of this register by one bit.

	IN	Q_1	Q_2	Q_3	Q_4
CYCLE ϕ	1	0	0	0	0
CYCLE 1	0	1	0	0	0
CYCLE 2	1	0	1	0	0
CYCLE 3	1	1	0	1	0
CYCLE 4	0	1	1	0	1
CYCLE 5	0	0	1	1	0

```
entity shift is
    port( clk, in_signal  :  in std_logic;
          out_signal : out std_logic )
end shift;

architectural behavioural of shift is
begin
    process(clk)
        variable int_value : std_logic_vector(3 downto 0);
    begin
        if rising_edge(clk) then
            int_value := in_signal & int_value( 3 downto 1);
            out_signal <= int_value(0);
        end if;
    end process;
end behavioural;
```

SHIFT REGISTER WITH PARALLEL INPUT



IF LOAD = 1 THEN LOADS 4 BIT PARALLEL INTO THE REGISTER

IF LOAD = Ø THEN OPERATES NORMALLY

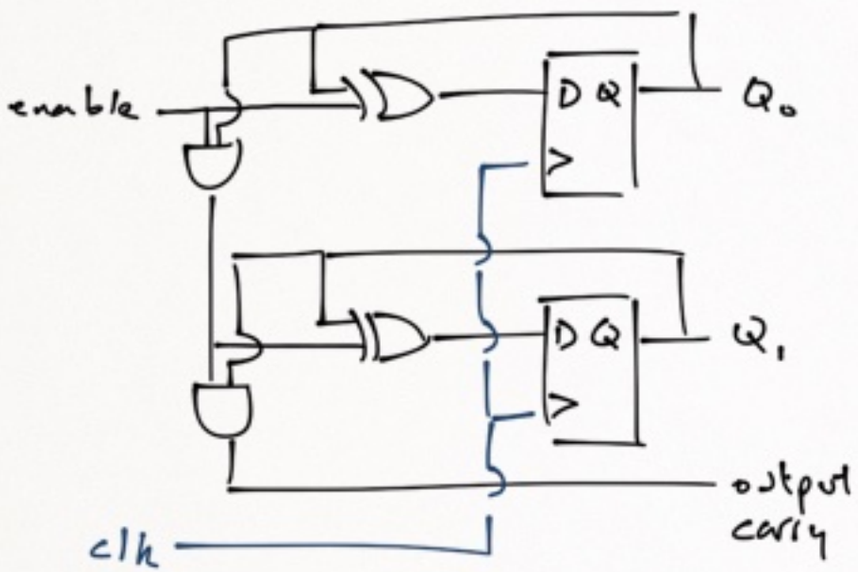
```
entity pshift is
    port( clk, insig, load : in std_logic;
          parallel_in : in std_logic_vector(3 downto Ø);
          outsig : out std_logic)
end p_shift;

architecture behavioural of p_shift is
begin
   process(clk)
      variable int_value : std_logic_vector(3 downto Ø);
   begin
      if rising_edge(clk) then
         if load = '1' then
            int_value := parallel_in;
         else
            int_value := insig & int value(3 downto 1);
         end if;
         outsig <= int_value(0);
      end if;
   end process;
end behavioural;
```

DIFFERENT TYPES OF COUNTERS

1. BINARY COUNTER
2. BCD COUNTER
3. RING/ONE-HOT COUNTER
4. JOHNSON/MOBIUS COUNTER
5. LINEAR FEEDBACK SHIFT REGISTER

COUNTERS ARE USED TO CREATE TIME DELAYS OR AS A CONTROLLER STEPPING THROUGH INSTRUCTIONS.

BINARY COUNTER (2-BIT)




```
entity count4 is
    port(clk, enable : in std_logic;
         count       : out std_logic_vector(3 downto 0));
end count4;

architecture behavioural of count4 is
begin
   process( clk )
      variable int_value : unsigned(3 downto 0);
   begin
      if rising_edge( clk ) then
          if enable = '1' then
              int_value := int_value + 1;
          end if;
      end if;
      count <= int_value;
   end process;
end behavioural;
```

FINISH

TIMING

$$t_{clk\text{-}to\text{-}q} + t_{path\text{-}max} + t_{setup} <= T$$

↑ PERIOD

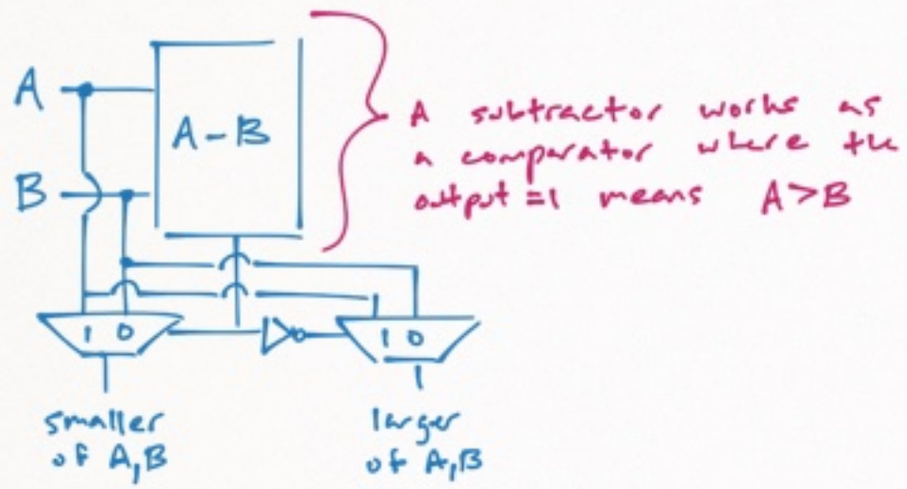
$$t_{p\text{-}min} >= t_{hold}$$

longest path from flip-flop to flip-flop or from flip-flop to itself = CRITICAL PATH

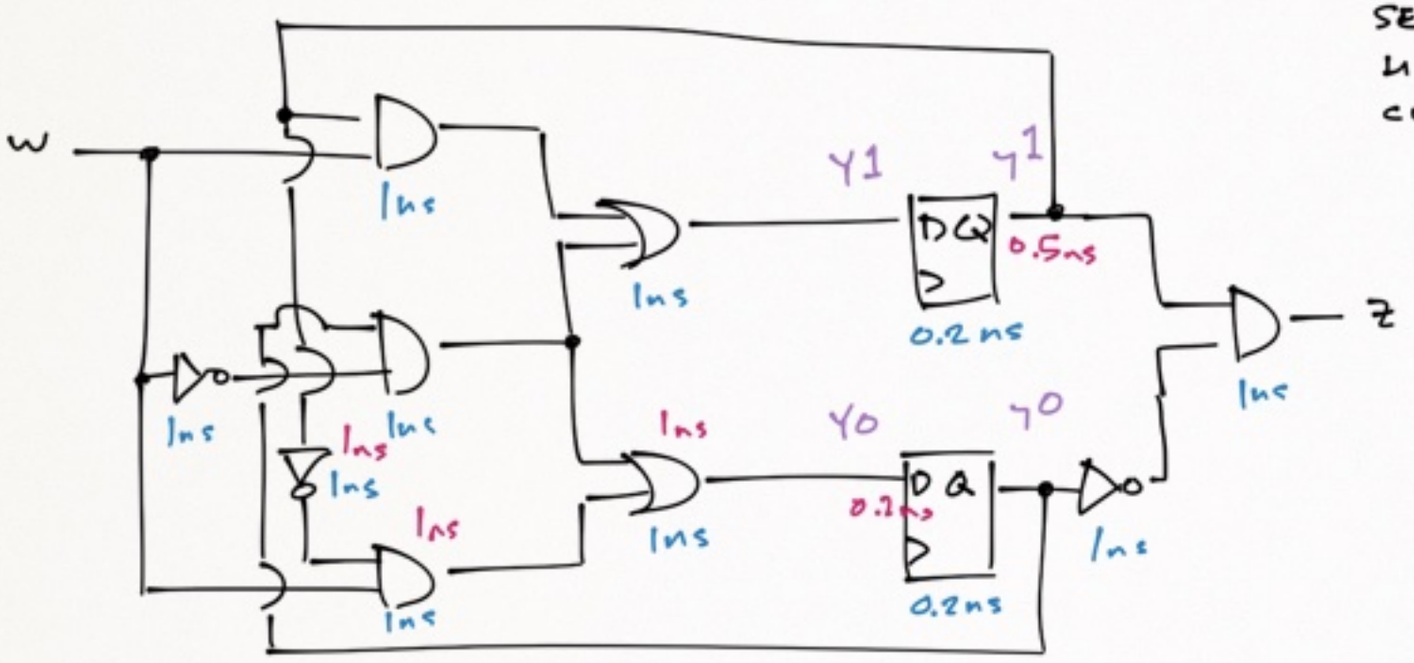
$$\text{max frequency} = \frac{1}{\text{DELAY OF CRIT. PATH}}$$

DATAPATH

SORTER



WRITTEN ASSIGNMENT #3



DELAY OF LOGIC GATES – 1ns
SETUP TIME OF EACH FLIP-FLOP – 0.2ns
HOLD TIME FOR EACH FLIP-FLOP – 0ns
CLK-TO-Q DELAY OF EACH FLIP-FLOP – 0.5ns

WHAT'S MAX FREQUENCY OF CLK?

SUPPOSE HOLD TIME IS NOT ZERO
WHAT'S LARGEST VALUE FOR HOLD-TIME
THIS CIRCUIT WILL OPERATE AT?

WRITE VHDL FOR CIRCUIT USING
SINGLE PROCESS.

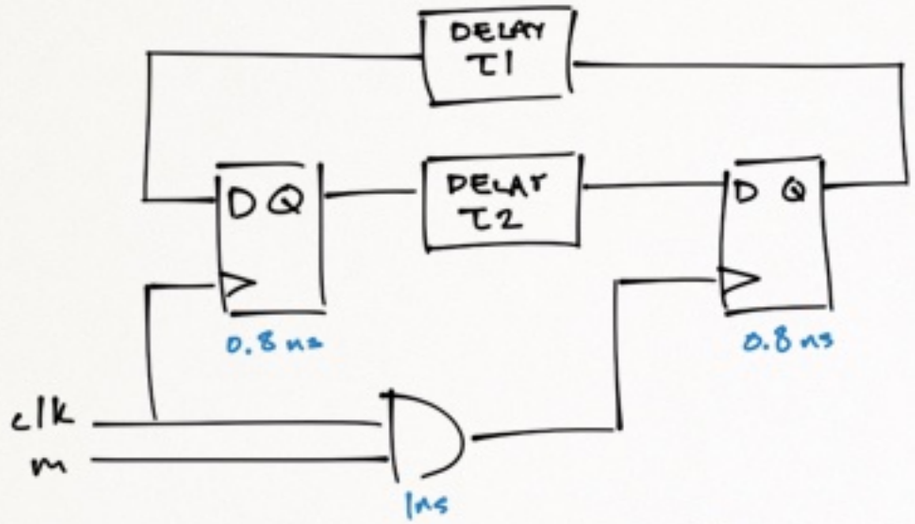
LONGEST PATH = 3.7ns

$\therefore$ MAX FREQUENCY = $\frac{1}{3.7ns} = \frac{1}{3.7} 10^9$ Hz

SHORTEST PATH = 2.0ns

$\therefore$ CIRCUIT WON'T FUNCTION IS HOLD TIME < 2ns

```
architecture behavioural of q1 is
   signal y: std_logic_vector(1 downto 0) := "00";
begin
   process(clk)
      variable next_state : std_logic_vector(1 downto 0);
   begin
      if rising_edge(clk) then
         next_state(1) := [present_state(1) and w] or [present_state(0) and not w];
         next_state(0) := [not present_state(1) and w] or [present_state(0) and not w];
         y <= next_state;
         z <= next_state(1) and not next_state(0);
      end if;
   end process;
end behavioural;
```



FF SETUP TIME = 0.3 ns
FF HOLD TIME = 0.2 ns
FF clk-to-q TIME = 0.5ns
AND-GATE TIME = 1ns

@ 100MHz MAX T1 = ?
MAX T2 = ?
MIN T1 = ?
MIN T2 = ?

PERIOD

MAX T1 = 10ns − 0.8ns − 1ns = 8.2 ns

MAX T2 = 10ns − 0.8ns + 1ns = 10.2ns

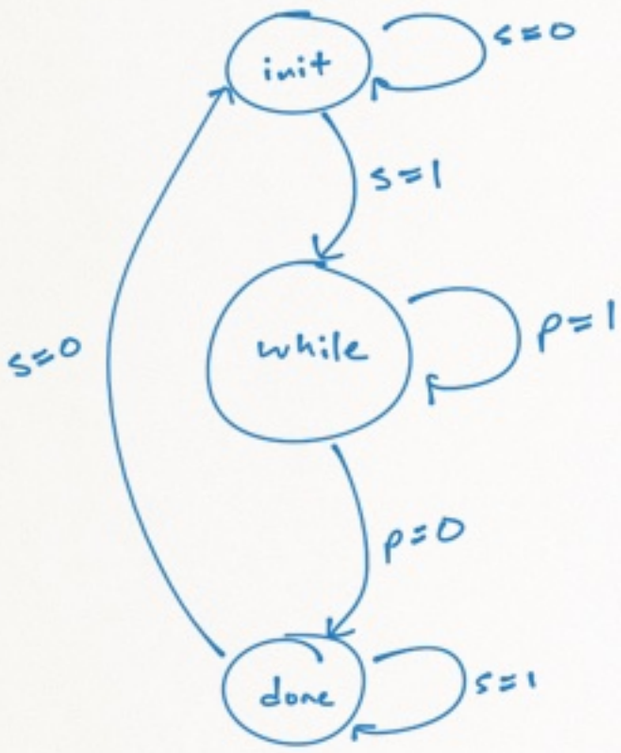
MIN T1 =

MIN T2

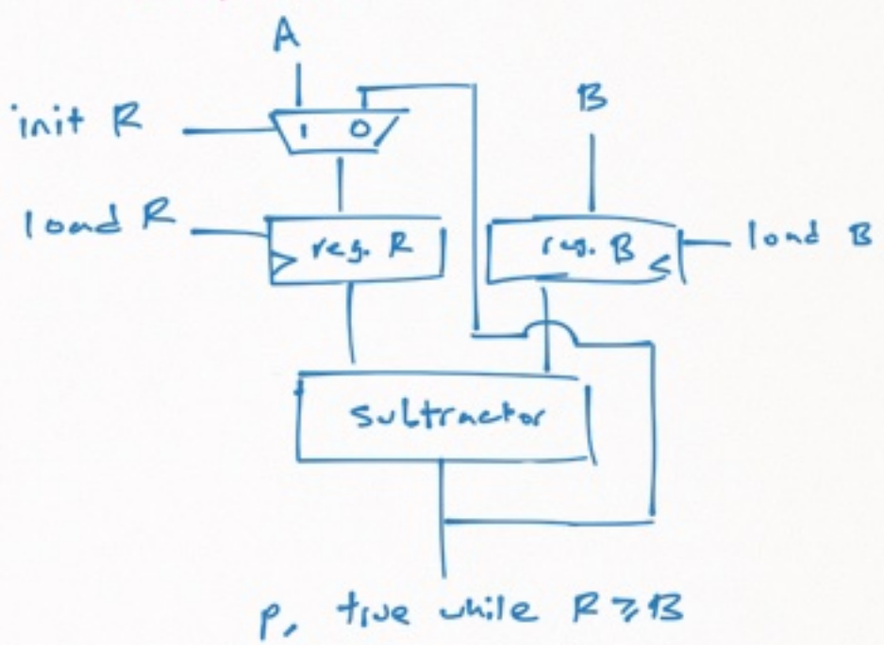
#5 IMPLEMENT CIRCUIT FOR ALGORITHM. INPUT S, OUTPUT DONE

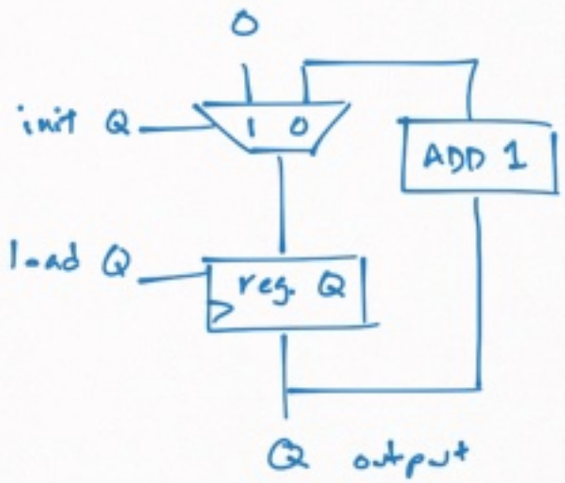
```
Q=0
R=A
while R≥B:
      R=R-B
      Q=Q+1
```

FSM



DATAPATH





20 LINES VHDL

100-200 LINES C

HOW TO CONNECT VGA TO C

→ INTERFACING TO A CPU

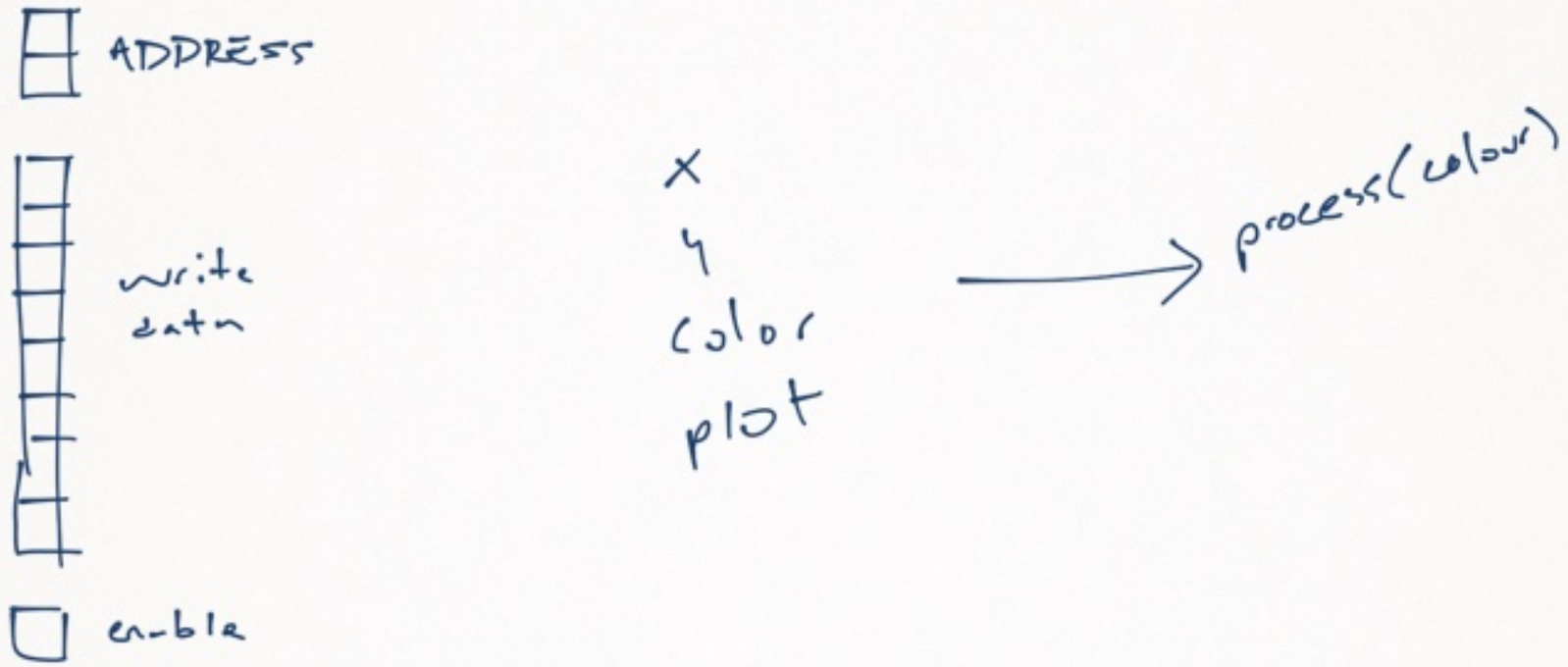
"CPU CENTRIC" VIEWPOINT, NOT "VHDL CENTRIC" VIEWPOINT

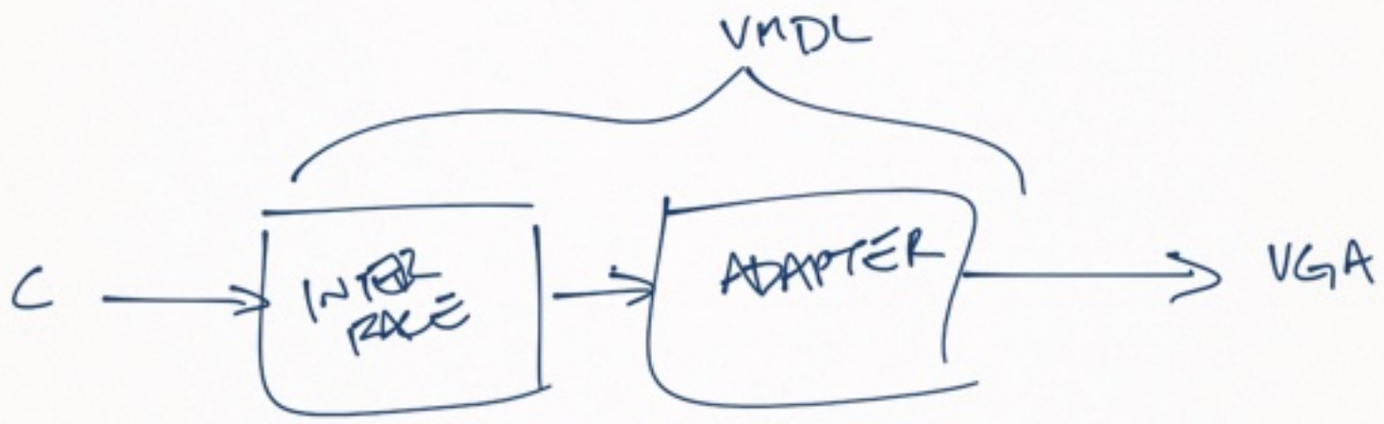
ADDRESS ~64 wires
DATA ~8-256 wires
CONTROL ~5 wires

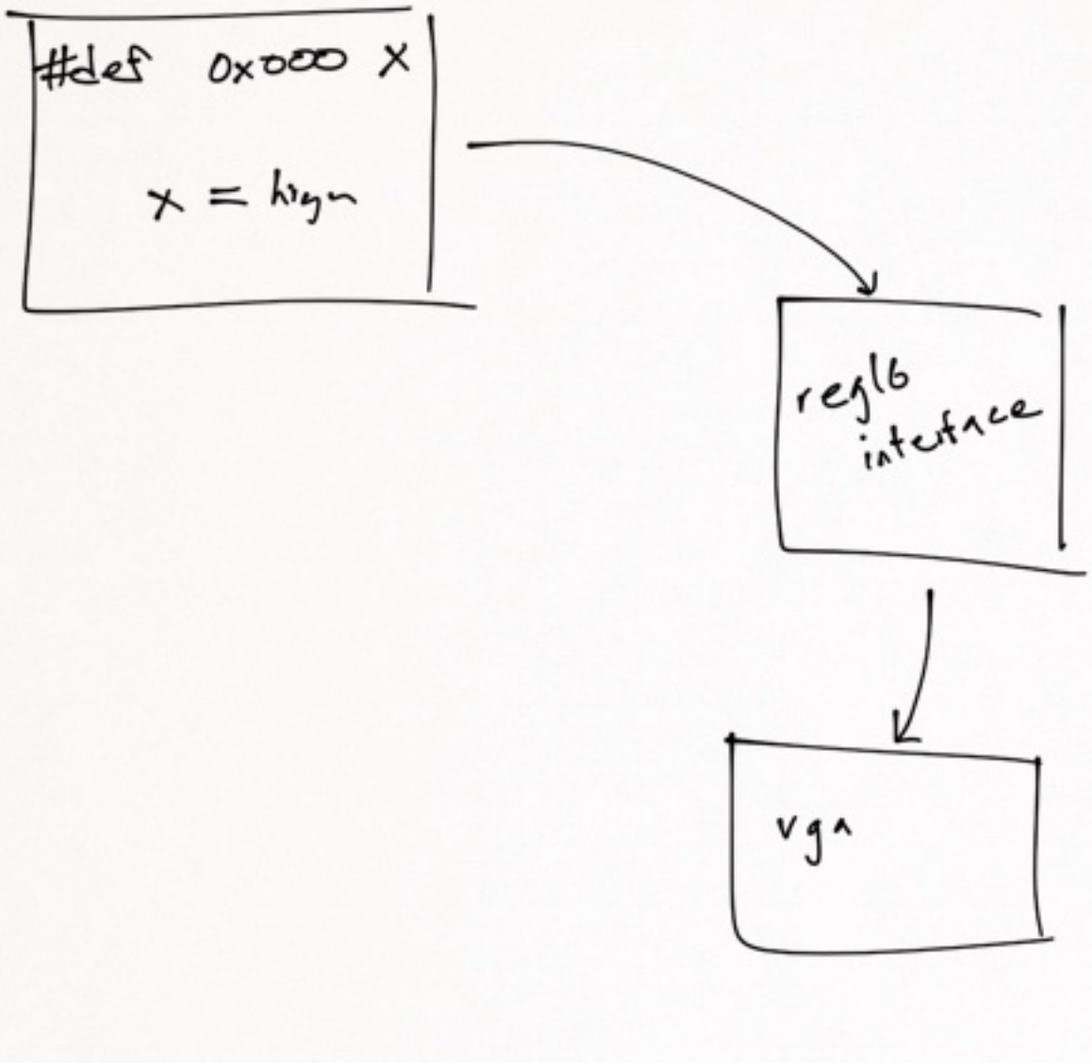
dram — 1 transistor + capacitor
sram — 16 transistors
flash - limited lifetime of writes

Slave never sends address
master never receives address

in complex systems, can have multiple masters


ADDRESS
write data
enable
x
y
color
plot
process(colour)


VHDL
C
INTERFACE
ADAPTER
VGA


#def 0x000 X
x = high
reg16 interface
vga

PLACEMENT

"Greedy algorithm"
↳ swap and only keep better results

sometimes you roll dice & keep bad results more
this probability of keeping worse results should decrease

} simulated annealing
↳ Altera

Xilinx solves using mathematical matrix to minimize $d_x + d_y$ for everything
↳ but this produces decimal for position when really positions are integer discrete
↳ it also puts things in the same position
↳ to make better, use legalization
↳ what's a legal position what's not

FINAL REVIEW

A1Q1 write VHDL description of an 8-to-1 two-bit mux

```
library ieee;
use ieee.std_logic_1164.all;

entity MUX is
   port(a,b,c,d,e,f,g,h: in std_logic_vector(1 downto 0);
     sel: in std_logic_vector(2 downto 0);
     output: out std_logic_vector(1 downto 0)
   );
end MUX;

architecture BEHAVIOURAL of MUX is
begin
   process(sel,a,b,c,d,e,f,g,h) --combinational
   begin
      case sel is
         when "000" => output <= a;
            .....
         when others => output <= h;
      end case;
   end process;
end BEHAVIOURAL;
```

AIQ2 write VHDL of 3-input 8-output decoder

a decoder takes an input # and sets that # bit off the output to one (asserts) and all other bits to zero

```
library ieee;
use ieee.std_logic_1164.all;

entity DECODER is
    port(
        input: in std_logic_vector(2 downto 0);
        output: out std_logic_vector(7 downto 0)
    )
end DECODER;

architecture BEHAVIOURAL of DECODER is
begin
  process(input)
  begin
    case input is
        when "000" => y <= "00000001";
        when "001" => y <= "00000010";
            . . . . .
        when others => y <= "10000000";
    end case;
  end process;
end BEHAVIOURAL;
```

AIQ3 write VHDL for the following table

En	w_1	w_2	y_0	y_1	y_2	y_3
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1
0	x	x	0	0	0	0

```
library ieee;
use ieee.std_logic_1164.all;

entity DECODER is
    port(
        w : in std_logic_vector(1 downto 0);
        en : in std_logic;
        y : out std_logic_vector(3 downto 0)
    );
end DECODER;

architecture BEHAVIORAL of DECODER is
begin
    process (en, w)
    begin
        if en='1' then
            case w is
                when "00" => y<= "1000";
                when "01" => y<= "0100";
                . . .
            end case;
        else
            y <= "0000";
        end if;
    end process;
end behavioral;
```

AIQ4 what is wrong with the following code for a decoder

```
entity demux is
      port( a   : in std_logic;
            sel : in std_logic_vector( 2 downto 0 );
            q   : out std_logic_vector( 7 downto 0 ) );
end demux;
architecture eece353 of demux is
begin
      process(sel,a)
      begin
            case(sel) is
                  when "000" => q(0) <= a;
                  when "001" => q(1) <= a;
                  ...
                  when others => q(7) <= a;
            end case;
      end process;
end eece353;
```

it's not driving all other bits of q with logic

to fix it put q<="00000000"; at the beginning.

to expand on this, the current code would tell the circuit to maintain their old values which would imply memory so it would create registers

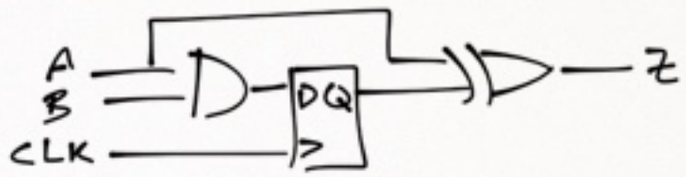
AIQ5 write VHDL for JK-flip-flop

J	K	Q
0	0	previous val of Q
0	1	0
1	0	1
1	1	Q'

```
entity JKFF is
    port(
        clk, j, k : in std_logic;
        q : out std_logic
    );
end JKFF;

architecture BEHAVIOURAL of JKFF is
begin
    process(clk);
        variable temp_q : std_logic;
    begin
        if rising_edge(clk) then
            if J='0' then
                if k='1' then
                    temp_q := '0';
                end if;
            else
                if k='0' then
                    temp_q := '1';
                else
                    temp_q := not temp_q;
                end if;
            end if;
        end if;
        q<=temp_q;
    end process;
end BEHAVIOURAL;
```

-- I forgot to use a variable to store information without reading in an out bit

A1Q6 write VHDL for following circuit

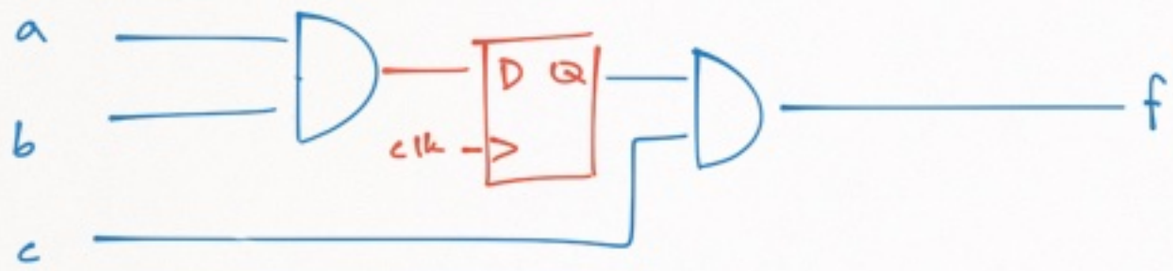



```
architecture BEHAVIOURAL of CIRCUIT is
begin
signal temp_q : std_logic;
    process(clk)
    begin
        if rising_edge(clk) then
            temp_q <= a and b;
        end if;
    end process;

    process (q, temp_q)
    begin
       z <= a xor temp_q;
    end process;
end BEHAVIOURAL;
```

-- I made the whole thing synchronous when the Z is sensitive to a

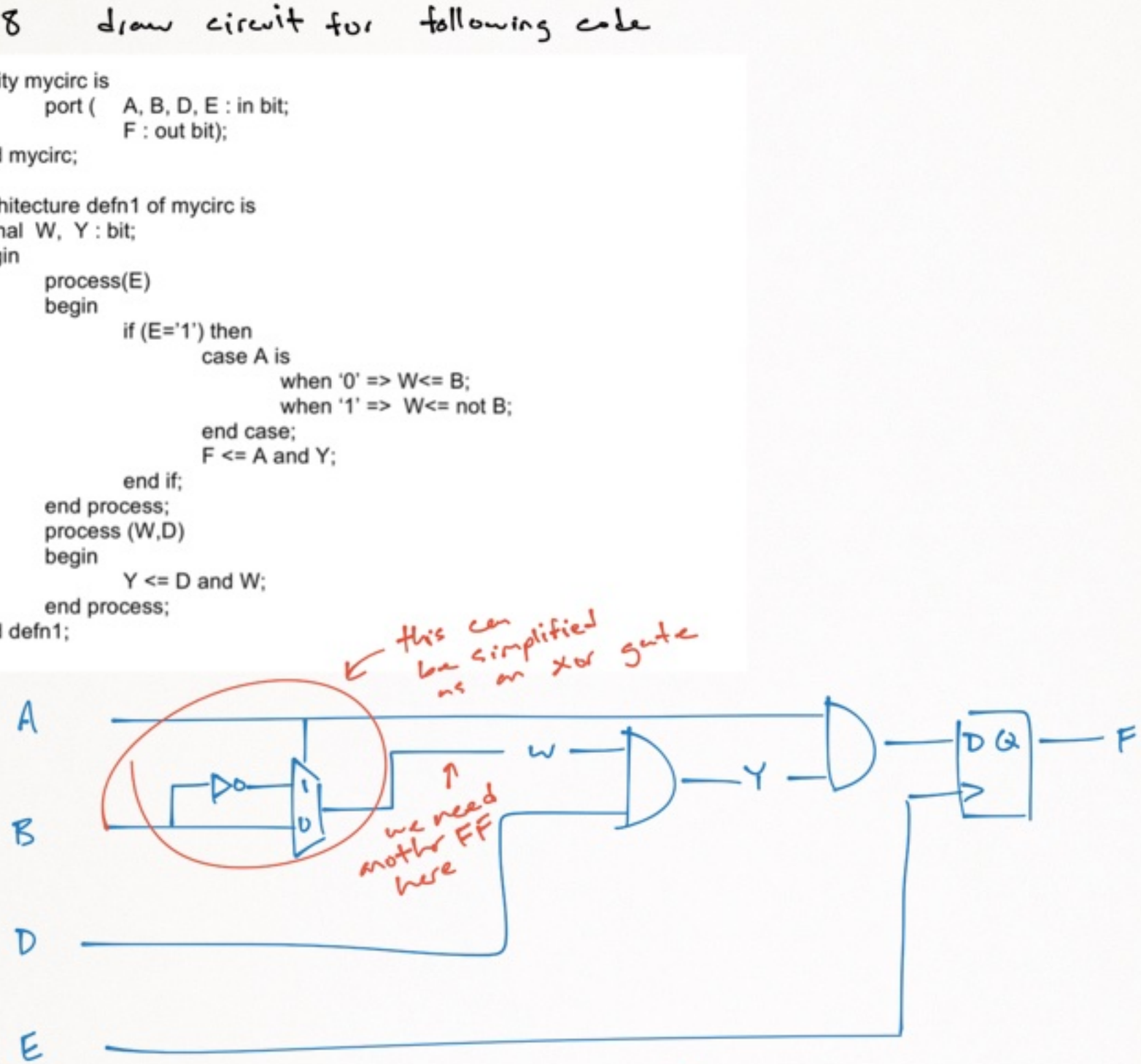
A1Q7 draw circuit for the following code

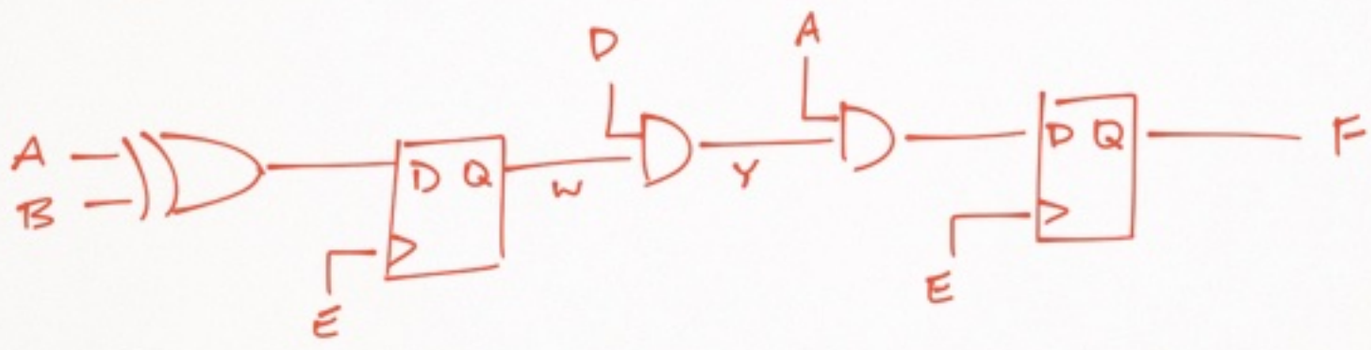
```
entity mycirc is
        port (    A, B, C, CLK : in bit;
                  F : out bit);
end mycirc;

architecture defn1 of mycirc is
signal  W : bit;
begin
        process(CLK)
        begin
                if (CLK='1') then
                    W <= A and B;
                end if;
        end process;

        F <= W and C;

end defn1;
```



AIQ8 draw circuit for following code

```
entity mycirc is
        port (    A, B, D, E : in bit;
                  F : out bit);
end mycirc;

architecture defn1 of mycirc is
signal  W,  Y : bit;
begin
        process(E)
        begin
                if (E='1') then
                        case A is
                                when '0' => W<= B;
                                when '1' =>  W<= not B;
                        end case;
                        F <= A and Y;
                end if;
        end process;
        process (W,D)
        begin
                Y <= D and W;
        end process;
end defn1;
```



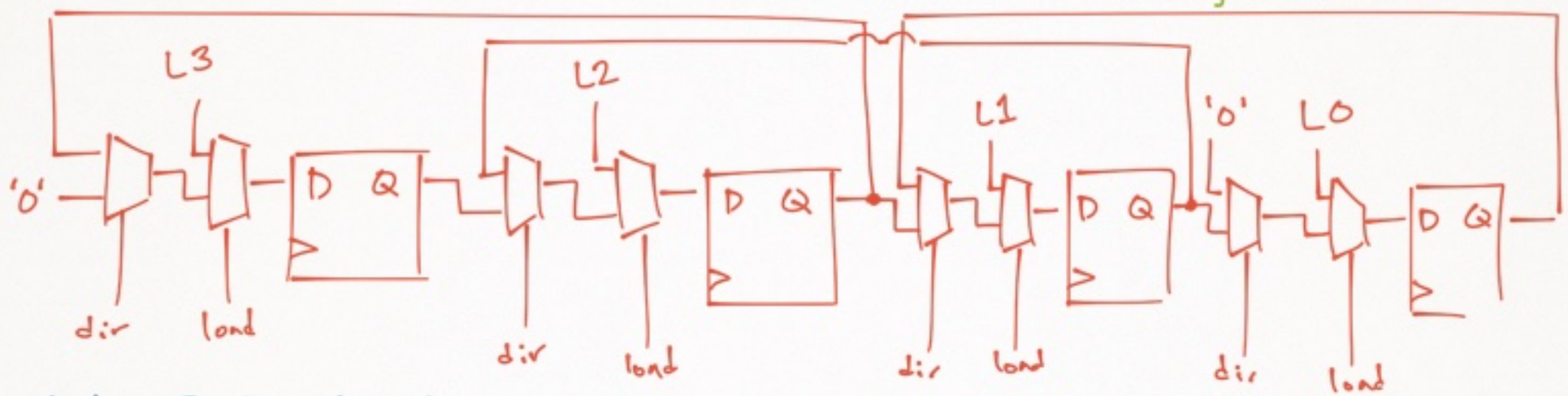


A2Q2 draw schematic of a 4-bit universal shift register. use D-flip-flops, muxs, & gates that has parallel load capability

A2Q3 write VHDL using single process

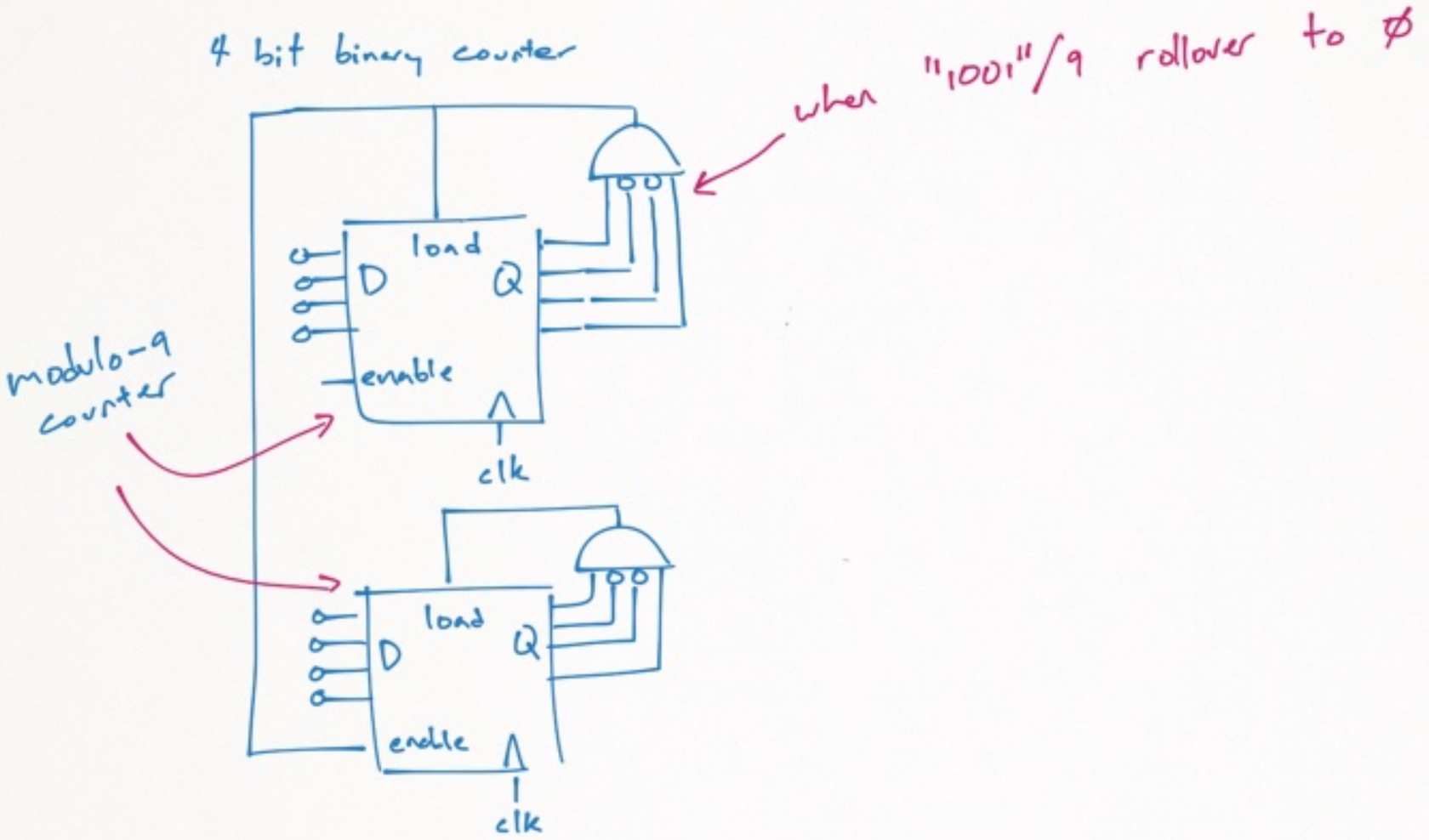
↳ left & right directionality

↳ cascaded FF with each feeding into the next




```
architecture BEHAVIOURAL of SHIFTREG
begin
    process (clk)
        variable reg : std_logic_vector(3 downto 0);
    begin
        if rising_edge(clk) then
            if load ='1' then
                reg := L;
            elsif dir = '0' then
                reg := '0' & reg(3 downto 1);
            else
                reg := reg(2 downto 0) & '0';
            end if;
            outsig <= reg(0);
        end if;
    end process;
end BEHAVIOURAL;
```

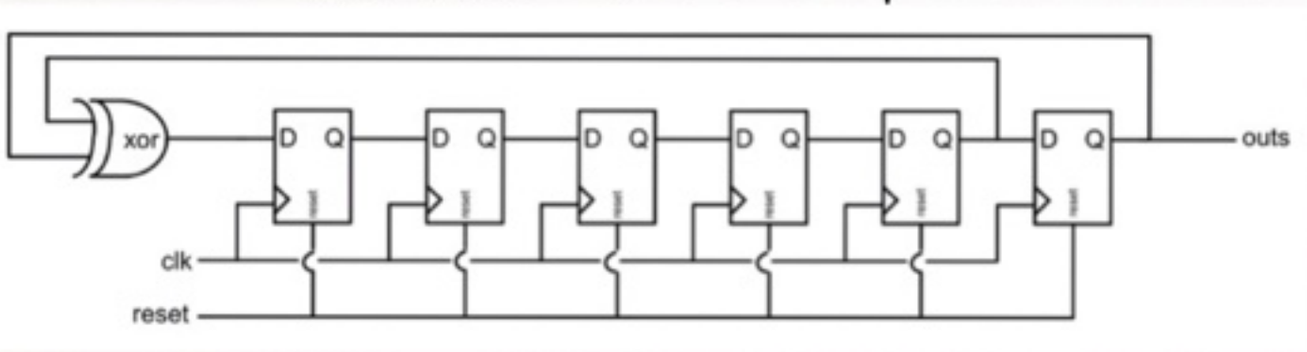
A203 design 8-bit BCD counter using 4 bit binary counters



A2Q4 write VHDL for it

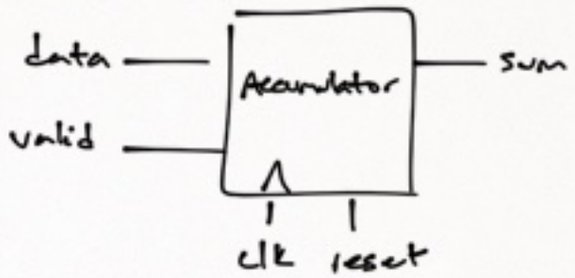
```
architecture BEHAVIOURAL of Q4 is
begin
   process(clk)
      variable BCD0 : unsigned(3 downto Ø) := "0000";
      veriable BCD1 : unsigned(3 downto Ø) := "0000";
   begin
      if rising_edge(clk) then
         if BCD0 = "1001" then
            BCD0 := "0000";
            if BCD1 = "1001" then  BCD1 := "0000";
            else BCD1 := BCD1 + 1;
            end if;
         else
            BCD0 := BCD0 + 1;
         end if;
         output <= std_logic_vector(BCD1 & BCD0);
      end if;
   end process;
end BEHAVIOURAL;
```

A2Q5 consider the following linear-feedback shift register (LFSR), it is asynchronous write VHDL that uses one process




```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numerical.all;   use ieee.numeric_std.all;

entity LFSR is
    port( clk, reset : in std_logic;
          outs : out std_logic
    );
end LFSR;

architecture BEHAVIOURAL of LFSR is
begin
    process( clk, reset )
        variable reg : std_logic_vector(5 downto 0);
    begin
        if reset ='1' then
            reg := "000000";
        else
            if rising_edge(clk) then
                reg := (reg(5) xor reg(4)) & reg(5 downto 1);
                                (reg(1) xor reg(0))
            end if;
        end if;
        outs <= reg(5);
                reg(0);
    end process;
end BEHAVIOURAL;
```

A2Q6 write VHDL for a 16-bit accumulator

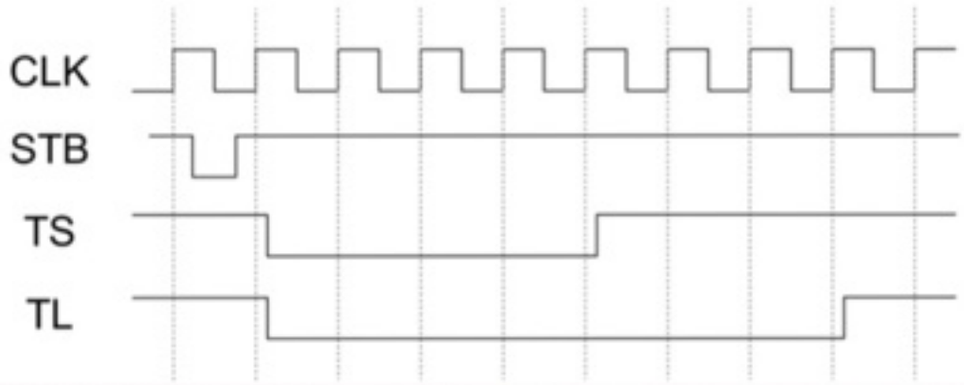


if valid is '1' whatever is stored in accumulator + data is output in sum each clk cycle

```
entity ACCUMULATOR is
    port( clk, reset, valid : in std_logic;
          data : in std_logic_vector(15 downto 0);
          sum : out std_logic_vector(15 downto 0)
    )
end ACCUMULATOR;

architecture BEHAVIOURAL of ACCUMULATOR is
begin
  process (clk)
     variable stored : unsigned(15 downto 0);
  begin
     if rising_edge(clk) then
        if reset = '1' then
            stored := 0;
        else
            if valid ='1' then
                stored := unsigned(data) + stored;
            end if;
        end if;
     end if;
     sum <= std_logic_vector(stored);
  end process;
end BEHAVIOURAL;
```

A2Q7 write VHDL for this:

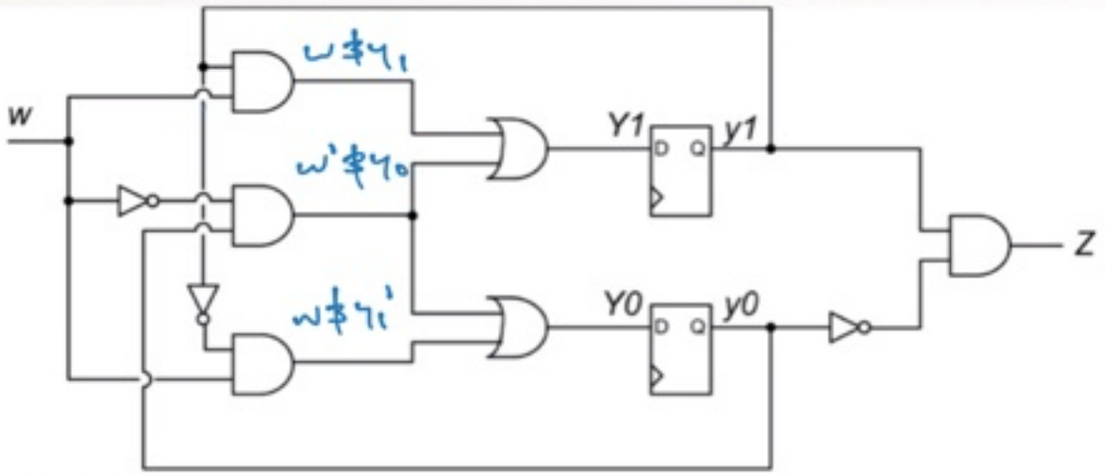
7. Write synthesizable VHDL code to specify the following behaviour. When input STB goes low, outputs TL and TS go low at the next rising clock edge. When input STB goes high, TS goes high after 4 clock ticks, at TL goes high after 7 clock ticks. If STB goes low again before TL goes high, the count should reset (start counting again at 0). (If this was being marked, marks would be deducted for solutions which are excessively complex or long.)




```
architecture BEHAVIOURAL of RANDOM_WAVEFORM is
begin
    process(clk)
      variable counter : integer := 0;
    begin
       if rising_edge(clk) then
            if STB ='0' then
                 TS ='0';
                 TL ='0';
                  counter =0;
            else
                if counter = 4 then
                     TS = '1';
                elsif counter = 7 then
                     TL = '1';
                end if;
                counter = counter + 1;
            end if;
      end if;
```

```
architecture BEHAVIOURAL of TIMER is
    signal count: unsigned(3 downto 0) := "0000";
begin
    process(clk,STB)
    begin
        if STB = '0' then
           count <= "1000";
        elsif rising_edge(clk) then
           if count > 0 then
              count <= count - 1;
           end if;
        end if;
    end process;

    process(count)
    begin
        if count >3 and count <8 then TS<='0';
        else TS <='1';
        end if;
        if count >0 and count <8 then TL<='0';
        else TL <='1';
        end if;
    end process;
```

A3Q1 consider the following circuit



Assume the following:

- Delay of each logic gate: 1 ns
- Set up time of each flip-flop: 0.2 ns
- Hold time of each flip-flop: 0 ns
- Clk-to-Q delay of each flip-flop: 0.5 ns

$t_{cp} = t_{cycle} - t_{su} - t_{clktoq}$

setup requirement:

$T \geq t_{clktoq} + t_{p\text{-}max} + t_{su}$

$t_{p\text{-}min} \geq t_{hold}$

↑ for our purposes, $t_{hold} = 0ns$

a) what's the maximum frequency of the clock in this circuit?

critical path = 0.5ns + 3(1ns) + 0.2ns = 3.7ns

↳ max delay from any flip-flop output to any flip-flop input

$$\text{max frequency} = \frac{1}{\text{critical path}} = \frac{1}{3.7} 10^9 \text{ Hz} = 270 \text{ MHz}$$

b) what's the max possible hold time?

$t_{hold} \leq t_{p\text{-}min}$, $t_{p\text{-}min} = 2(1ns)$

c) write VHDL for the above circuit using single process

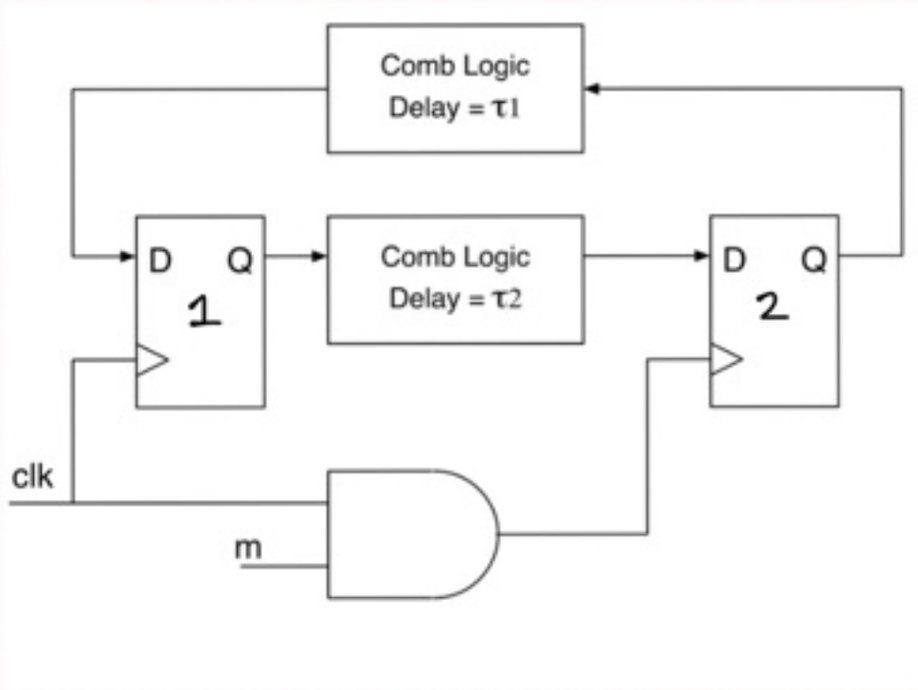
```
if rising_edge(clk) then
    y0 := (not w and y0) or (w and not y1);
    y1 := (not w and y0) or (w and y1);
    z <= not y0 and y1;
```

```
signal y : std_logic_vector(1 downto 0)
process(clk)
variable y_temp : std_logic_vector(1 downto 0)
if rising_edge(clk) then
    y_temp(0) := (not w and y_temp(0)) or ...
    y_temp(1) := (not w and ...
    y <= y_temp;
    z <= not y_temp(0) and y_temp(1);
```

A3Q2 consider the following circuit

Assume the following:

Set up time of each flip-flop: 0.3 ns
Hold time of each flip-flop: 0.2 ns
Clk-to-Q delay of each flip-flop: 0.5
Delay of the AND gate: 1 ns



lets say we want to run this at 100MHz

a) what's the max value of $\tau 1$?

$$T \geq t_{clk\text{-}to\text{-}q} + t_{p\text{-}max} + t_{su}$$

$$\frac{1}{100}10^{-6} = 10ns \geq 0.5ns + t_{p\text{-}max} + 0.3ns$$

$$t_{p\text{-}max} = 9.2ns$$

HOWEVER, NOTE THE CLK REACHES FF1 1ns EARLY RELATIVE TO THE CLK AT FF2

SO $t_{p\text{-}max} = 9.2ns - 1ns = \underline{8.2ns}$

b) what's the maximum $\tau 2$?

same as above but the clock reaches FF2 1ns late as oppored to early relative to FF1

$t_{p\text{-}max} = \underline{10.2ns}$

c) min value of $\tau 1$?

for $\tau 1$, the signal at FF1 will always be delayed 1ns from the signal at FF2, so

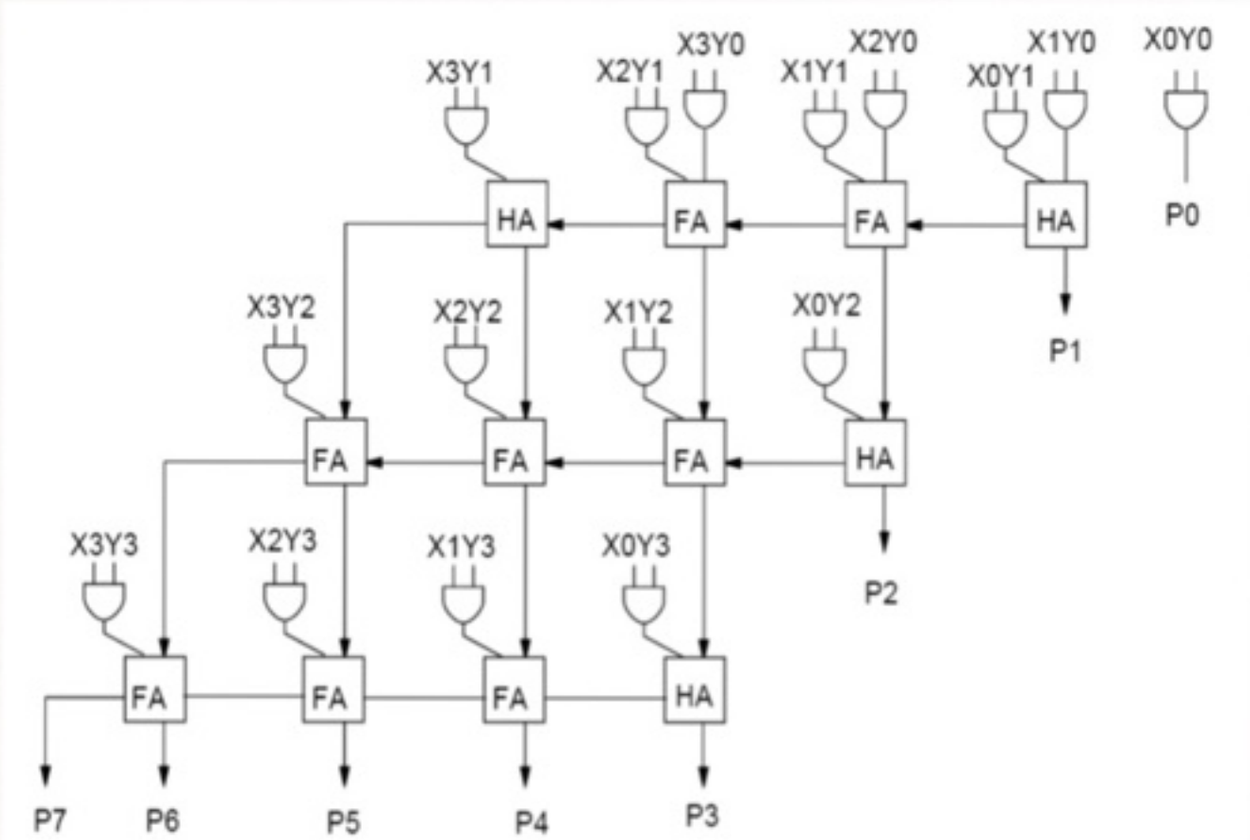
$t_{p\text{-}min} \geq t_{hold}$ $t_{p\text{-}min} = 1ns + \tau_1 \geq 0.2ns$ so $\underline{\tau 1 = \emptyset}$ and the condition still be satisfied

d) min $\tau 2$? $t_{p\text{-}min} \geq 0.2ns + 1ns = \underline{1.2ns}$

A3Q3 processor run @ 2GHz, delay of logic gates is 0.08ns
setup, hold, and clk-to-q time = 0. what's max number of gates in critical path?

$\frac{1}{2GHz} = 0.5(10^{-9}) = 0.5ns = N(0.08ns)$, $N \approx 6$ gates

A3Q4 what's the critical path of the multiplier?

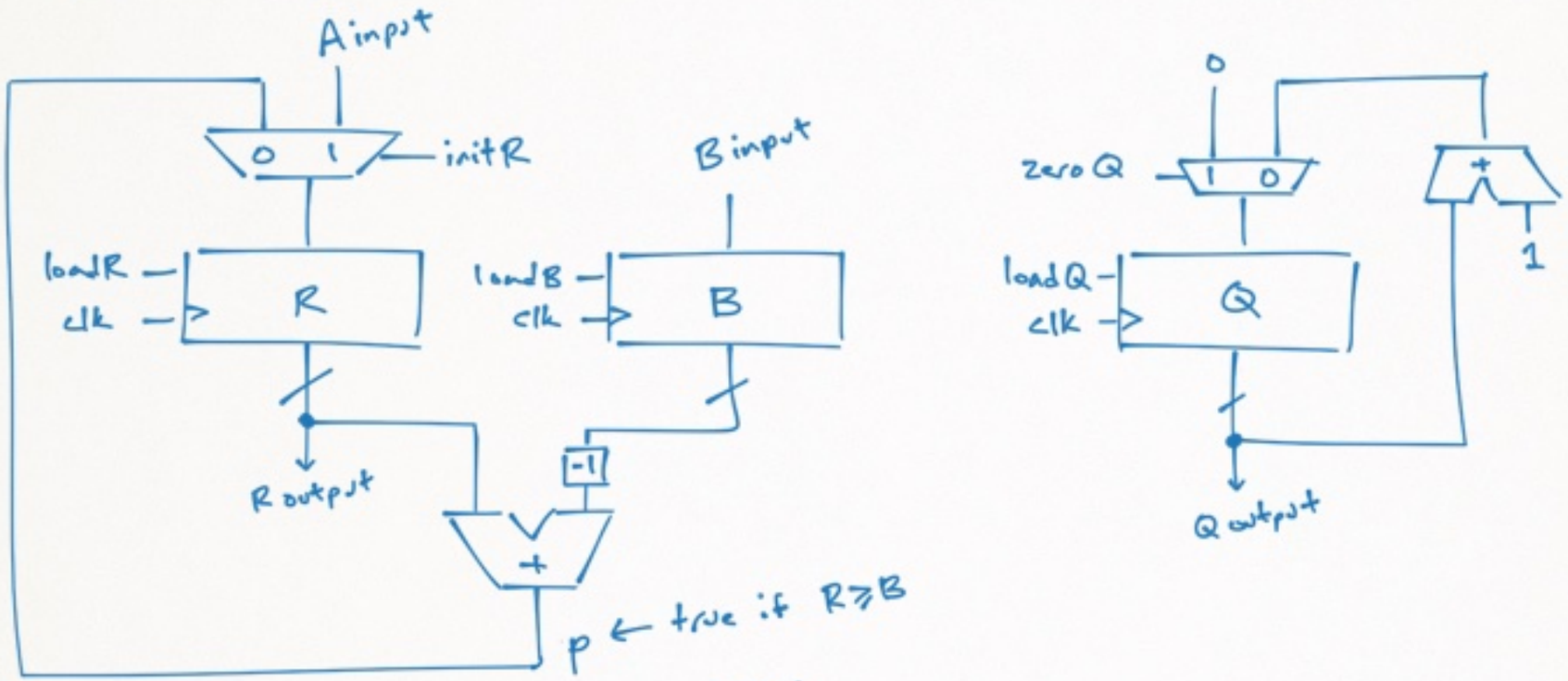


the delay of a F.A. = 3Δ

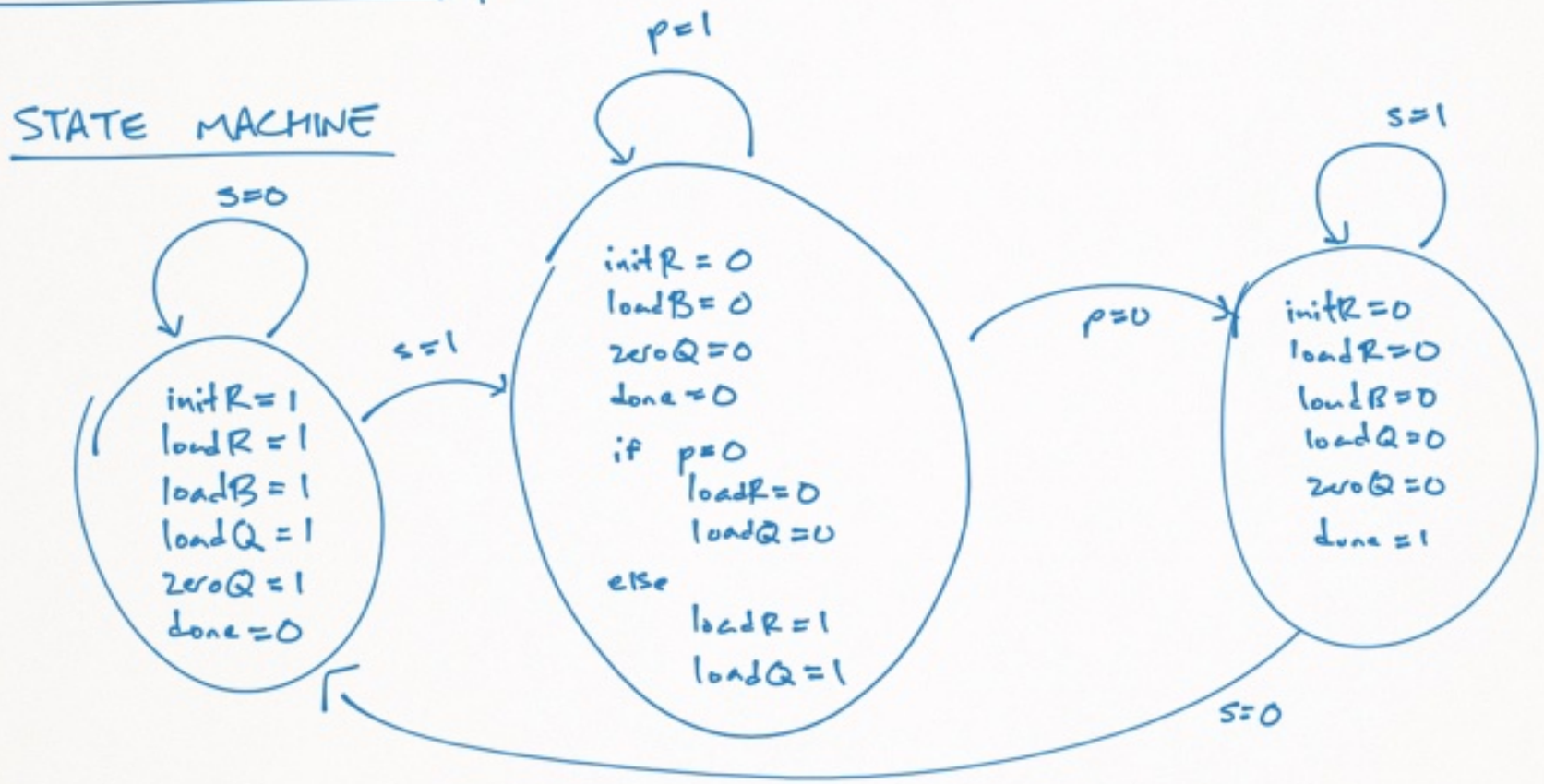
A3Q5 design a circuit to implement a divider using the algorithm

```
Q = 0
R = A
while(R ≥ B){
    R = R - B
    Q = Q + 1
}
```

DATAPATH



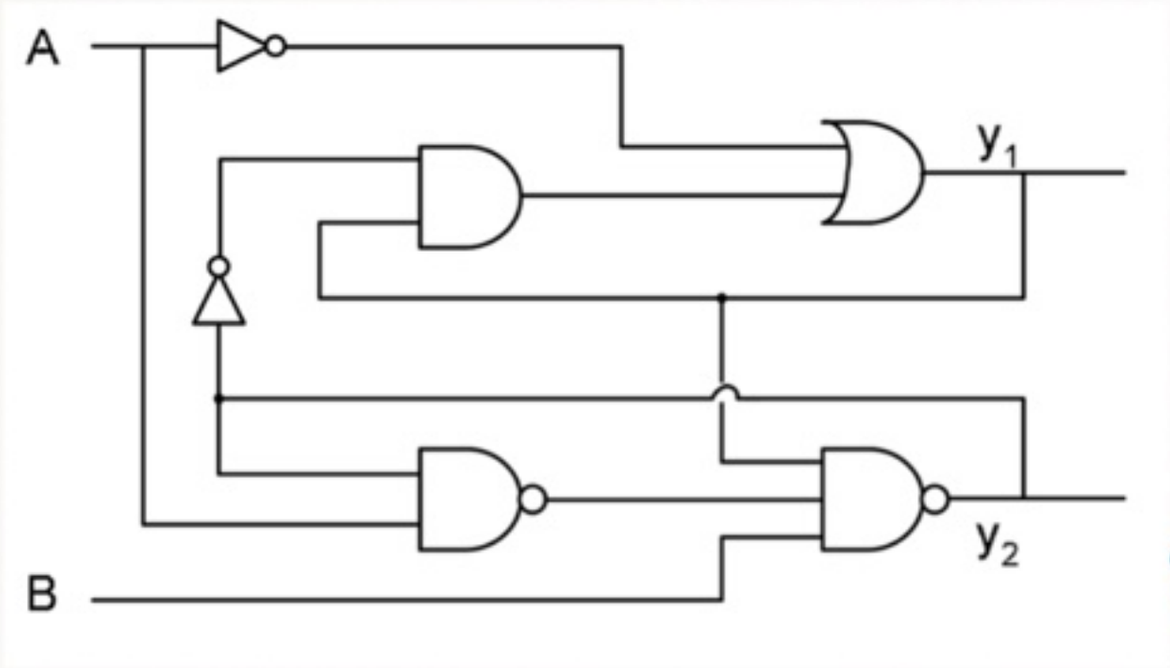
STATE MACHINE

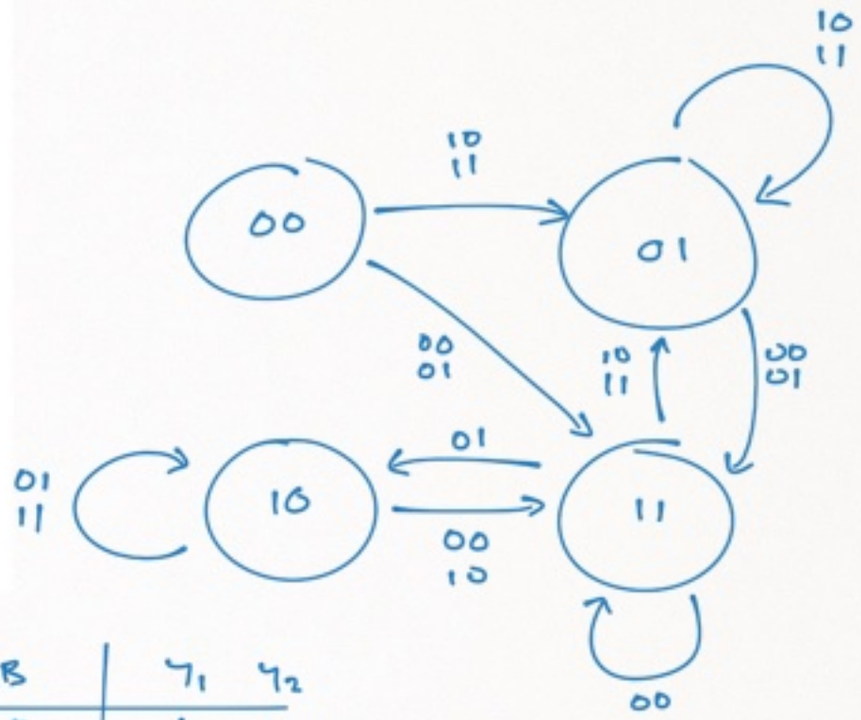


A3Q6

REVISIT

A4Q1 draw state machine for following asynchronous state machine





$$Y_1 = \bar{A} + (\bar{Y}_2 \cdot Y_1)$$

$$Y_2 = \overline{(B \cdot \overline{(A \cdot Y_2)} \cdot Y_1)}$$

$$= \overline{B \cdot Y_1 \cdot (\bar{A} + \bar{Y}_2)}$$

$$= \bar{B} + \bar{Y}_1 + \overline{(\bar{A} + \bar{Y}_2)}$$

$$\overline{\bar{A} + \bar{Y}_2}$$

A=0 Y=0 $\quad \overline{1+1} = \bar{1} = 0$

A=1 Y=0 $\quad \overline{1+0} = \bar{1} = 0$

A=1 Y=1 $\quad \overline{0+0} = \bar{0} = 1$

Y_1	Y_2	A	B	Y_1	Y_2
0	0	0	0	1	1
0	0	0	1	1	1
0	0	1	0	0	1
0	0	1	1	0	1
0	1	0	0	1	1
0	1	0	1	1	1
0	1	1	0	0	1
0	1	1	1	0	1
1	0	0	0	1	1
1	0	0	1	1	0
1	0	1	0	1	1
1	0	1	1	1	0
1	1	0	0	1	1
1	1	0	1	1	0
1	1	1	0	0	1
1	1	1	1	0	1

A4Q2 consider following asynchronous state machine equations

$Y_1 = AB' + Y_1 Y_2 B$

$Y_2 = AB + Y_1$

Are there any static hazards (potential glitches) if these are implemented directly? if so, how can you eliminate them?

Y_1

Y_1Y_2 / AB	00	01	11	10
00	0	0	0	0
01	0	0	1	0
11	0	0	1	0
10	1	1	1	1

Y_1Y_2B

AB'

this fixes glitch, add term Y_1Y_2A

these are not covered which poses a hazard

Y_2

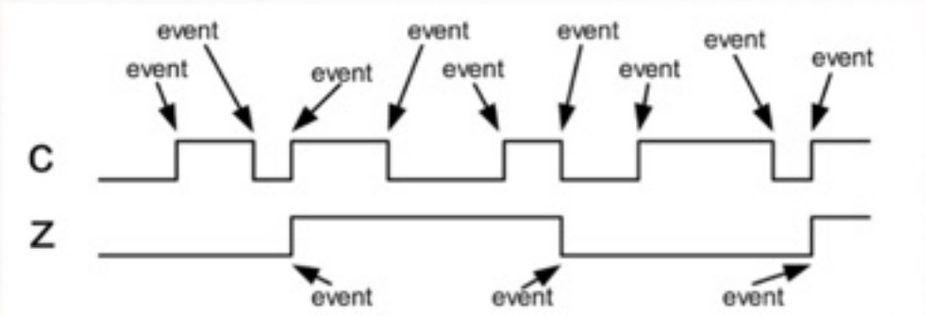
Y_1Y_2 / AB	00	01	11	10
00	0	0	1	1
01	0	0	1	1
11	1	1	1	1
10	0	0	1	1

Y_1

AB

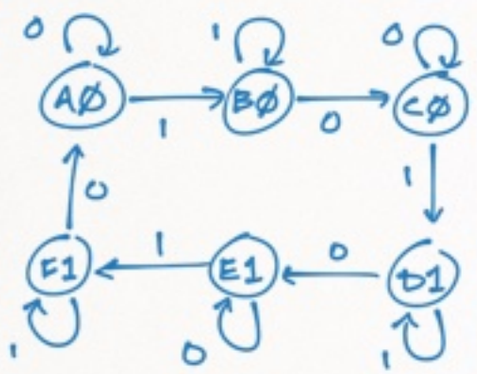
$Y_1 = AB' + AY_1Y_2 + BY_1Y_2$

A4Q3 design an asynchronous state machine



the circuit produces an event on Z for every third event on C

state machine



toggling C moves you through states, every 3 states has different output of Z

states	
A	000
B	001
C	011
D	111
E	101
F	100

consecutive states differ by 1-bit

state	next state C=0	next state C=1	output
A, 000	000	001	0
B, 001	011	001	0
C, 011	011	111	0
D, 111	101	111	1
E, 101	101	100	1
F, 100	000	100	1

y_2 y_1 y_0

y_1 (rows y_2y_1, columns y_0C)

y_2y_1 \ y_0C	00	01	11	10
00	0	0	0	1
01	–	–	1	1
11	–	–	1	0
10	0	0	0	0

$y_1 = y_0\bar{y}_2\bar{C} + y_1C + y_0y_1\bar{y}_2$

y_2 (rows y_2y_1, columns y_0C)

y_2y_1 \ y_0C	00	01	11	10
00	0	0	0	0
01	–	–	1	0
11	–	–	1	1
10	0	1	1	1

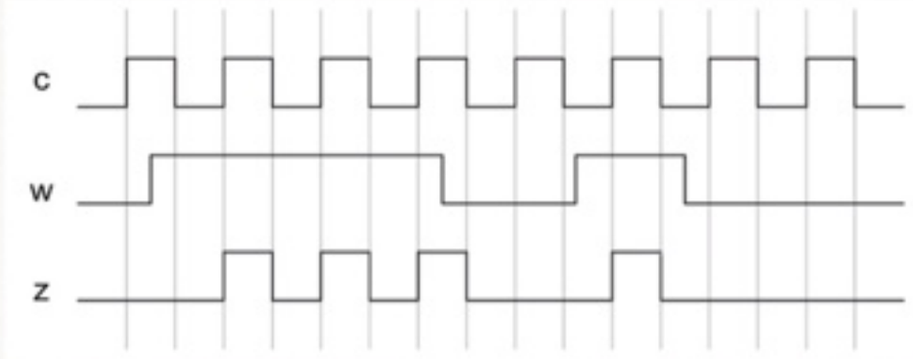
$y_2 = Cy_1 + Cy_2 + y_0y_2$

y_0 (rows y_2y_1, columns y_0C)

y_2y_1 \ y_0C	00	01	11	10
00	0	1	1	1
01	–	–	1	1
11	–	–	1	1
10	0	0	0	1

$y_0 = y_0\bar{C} + y_1y_0 + \bar{y}_2y_0 + \bar{y}_2C$

A4Q4



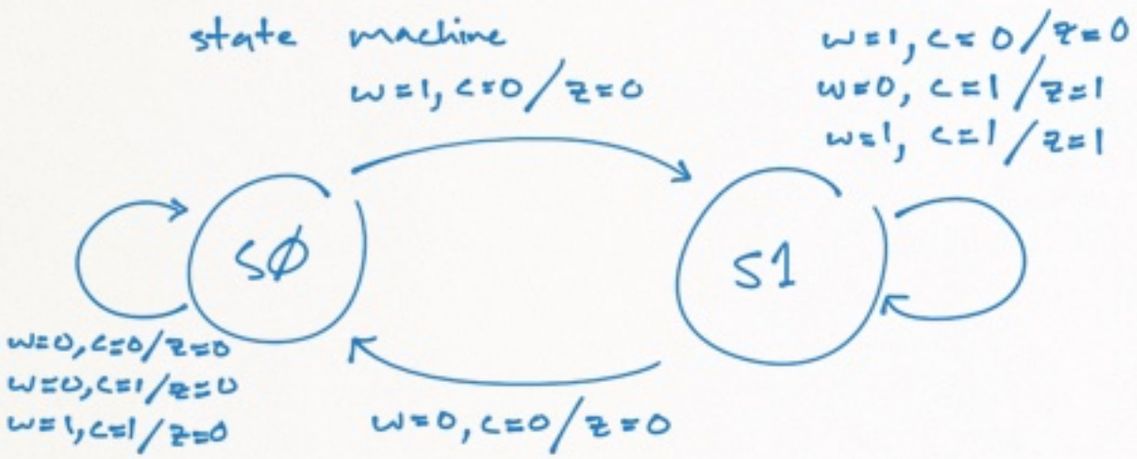
design asynchronous circuit that meets the specifications: clk c, input w, output z

z = clock when w=1, otherwise z=0

if w goes from 0→1, z turns on next rising edge

if w goes from 1→0, z turns on next falling edge

state machine



current state	next state wc=00, 01, 10, 11				output wc=00, 01, 10, 11			
S0	S0	S0	S1	S0	0	0	0	0
S1	S0	S1	S1	S1	0	1	0	1

next state

S \ wc	00	01	11	10
0	0	0	0	1
1	0	1	1	1

output

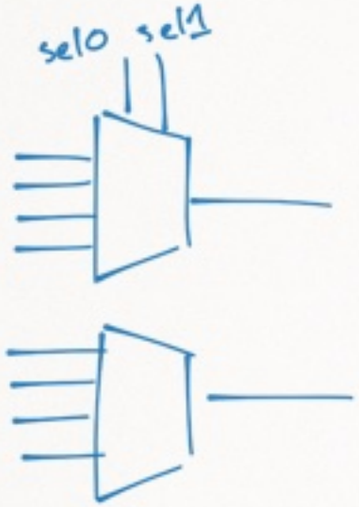
S \ wc	00	01	11	10
0	0	0	0	0
1	0	1	1	0

next state = $SC + SW + W\bar{C}$ output, $z = SC$

A4Q5 write in VHDL

```
entity WAVE is
    port (w,c : in  std_logic;
          z   : out std_logic
    );
entity WAVE;

architecture BEHAVIOURAL of WAVE is
   signal y: std_logic;
begin
   y <= (y and c) or (y and w) or (w and not c);
   y <= y and c;
end BEHAVIOURAL;
```

A4Q6 show how to implement 8-to-1 bit mux using 4-input L.U.T.s



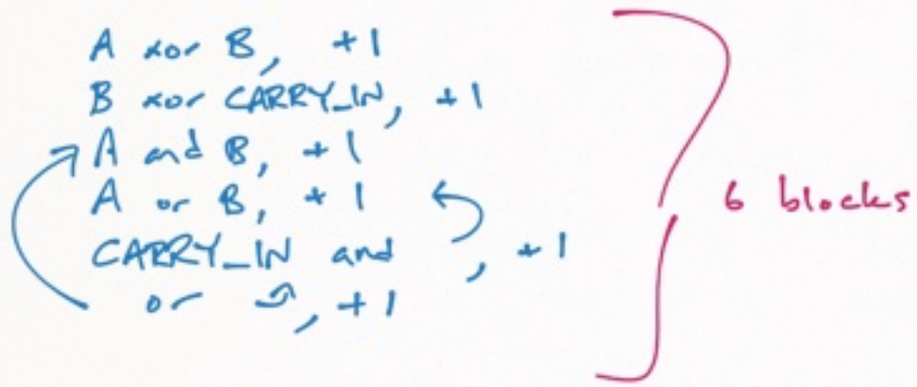
A4Q7 An n-input lookup table can implement ANY function of n inputs
how many different n-input functions are there?

an n-input L.U.T. has 2^n rows

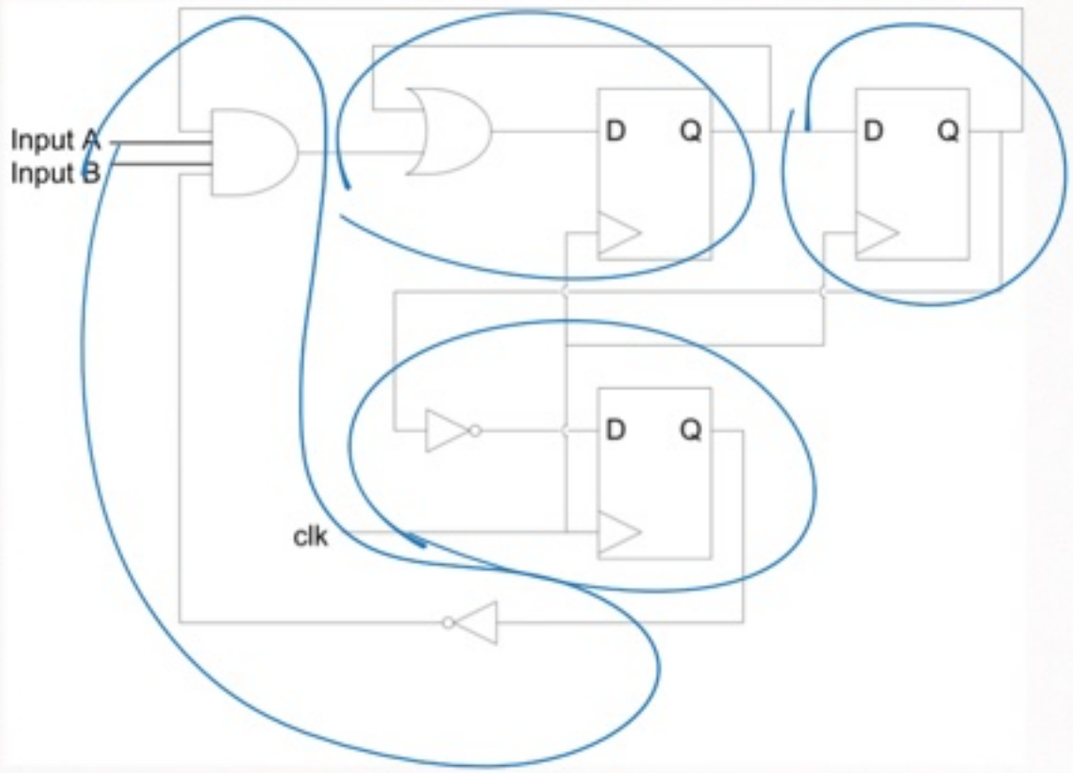
the output of any n-input LUT could be 1 or Ø, so there are 2^{2^n}

A4Q8 consider the code is compiled and mapped to FPGA consisting of 2-input L.U.T.s
How many 2-input L.U.T.s would be required?

```
architecture BEHAVIOURAL of FULL_ADDER is
begin
    SUM <= A xor B xor CARRY_IN;
    CARRY_OUT <= (A and B) or (CARRY_IN and (A or B));
end BEHAVIOURAL;
```

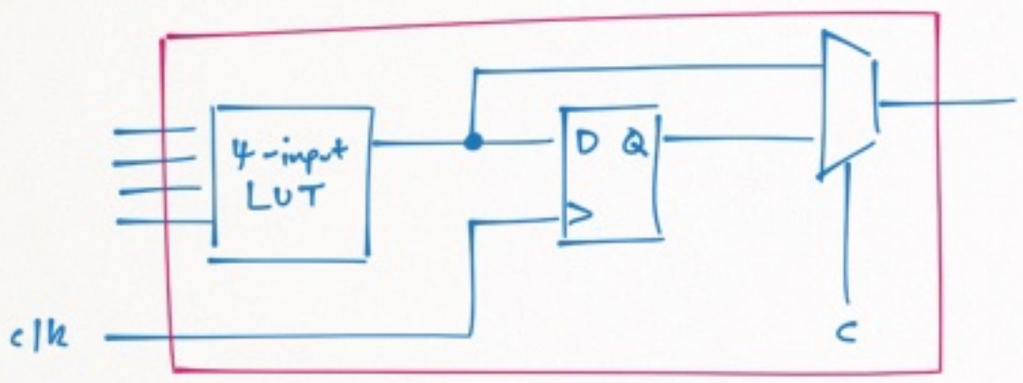


A4Q9



how many FPGA logic blocks are required to implement the circuit?

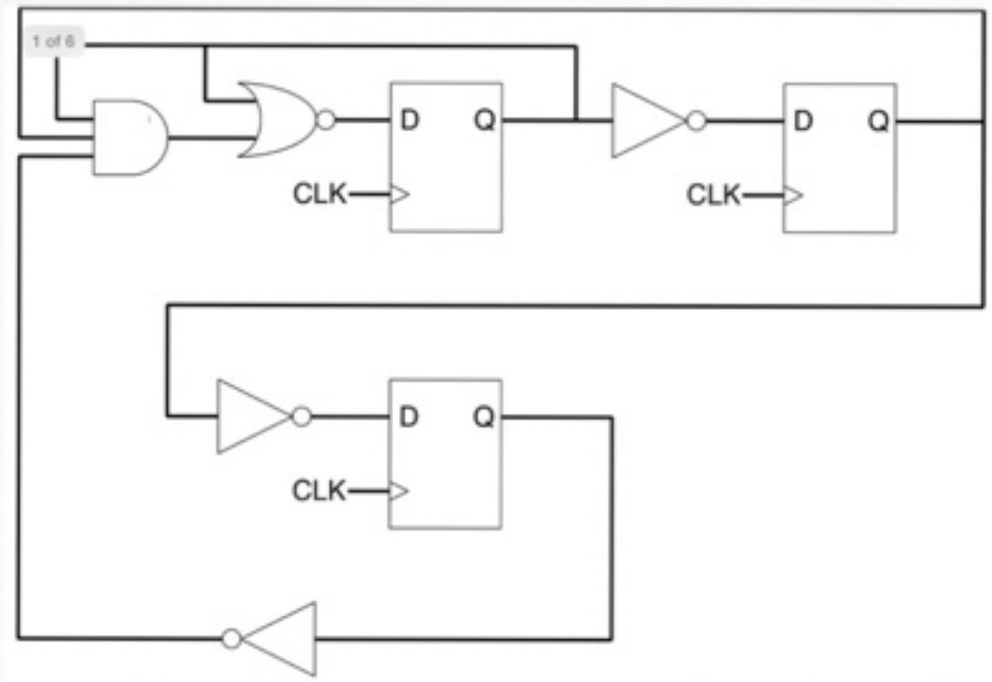
one logic block looks like this:



- this can incorporate a single gate and a flip-flop into one logic block
- the not gate and the and gate can be combined into one logic block
- then the flip-flop can get its own as well

4 logic blocks total

P1Q1



FF hold-time is 0.1ns
FF setup time is 0.2ns
FF clk-to-q is 0.4ns
inverter delay is 0.5ns
AND delay is 1ns
NOR delay is 0.8ns

a) what is the min. clk period

TGSUPMAXCQ, THLM

$T \geq t_{setup} + t_{p\_max} + t_{clktoq} = 0.2ns + 0.4ns + 0.5ns + 1ns + 0.8ns = \underline{2.9ns}$

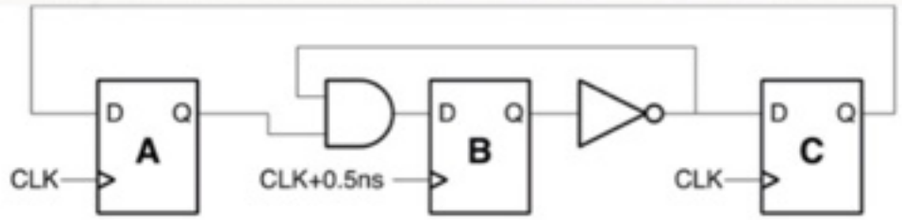
b) what is max clock period?

there is none, $\underline{\infty}$

c) what would be the max hold time?

$t_{hold} \leq t_{p\_min} = \underline{0.5ns}$

P1Q2



Assume the following:

- Hold time of each flip-flop is 0.5ns
- Setup time of each flip-flop is 0.2ns
- Clock-to-Q delay of each flip-flop is 0.1ns
- Delay of the NOT gate is 0.4ns
- Delay of the AND gate is 0.8ns
- As shown on the diagram, the clock connected to flip-flop B always arrives 0.5ns late

a) what's the minimum clock-period of this circuit

$T \leq t_{setup} + t_{p\text{-}max} + t_{clk\text{-}to\text{-}q}$

A→B : 0.8ns − 0.5ns
B→C : 0.4ns + 0.5ns
C→A : 0ns
B→B : 0.4ns + 0.8ns

$T \leq 0.2 + 0.3 + 0.1ns = 0.6ns$
$T \leq 0.2 + 0.9 + 0.1ns = 1.2ns$
$T \leq 0.3ns$
$T \leq 1.2ns + 0.2ns + 0.1ns = 1.5ns$

b) which flip-flops have a hold time violation?

$t_{hold} \leq t_{p\text{-}min}$
$0.5 \leq C \rightarrow A = 0ns$
$0.5 \leq A \rightarrow B = 0.3ns$

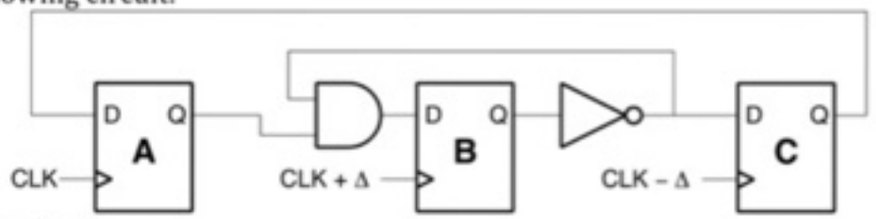
A FF has violation
B FF has violation

c) to fix hold-time violations, what flip-flops should you get?

ones with no hold time

P1Q3

3. Consider the following circuit:



Assume the following:

- Hold time of each flip-flop is 0.25ns
- Setup time of each flip-flop is 0.2ns
- Clock-to-Q delay of each flip-flop is 0.1ns
- Delay of the NOT gate is 0.4ns
- Delay of the AND gate is 0.8ns
- As shown on the diagram, the clock connected to flip-flops B and C suffer from *jitter* causing their clock edges to arrive early or late by an amount Δ; if an edge arrives early for B, it always arrives late for C by the same amount, and vice-versa. For any given clock edge, the value of Δ ranges from –0.2ns to +0.3ns.

a. What is the minimum clock period of this circuit? **You must show your work.**

$T \geq t_{setup} + t_{p\_max} + t_{clktoq}$

A→B: $0.2 + 0.1 + 0.8 - \Delta = 1.1 - \Delta$

B→C: $0.2 + 0.1 + 0.4 + 2\Delta = 0.7 + 2\Delta$

C→A: $0.2 + 0.1 + 0 = 0.3$

B→B: $0.2 + 0.1 + 0.4 + 0.8 = 1.5$

b) which flip-flops have violations?

$t_{hold} \leq t_{p\_min}$

$0.25ns \leq \emptyset$ A ✗

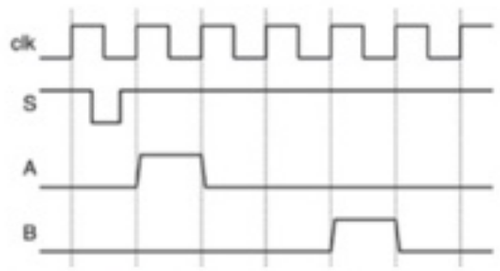
$0.25ns \leq 0.8 - 0.3$ B ✓

$0.25ns \leq 0.4 + 2\Delta = 0.4 - 0.6 = -0.2$ C ✗

c) how to fix?

P1Q5

Write synthesizable VHDL code that corresponds to the behaviour below. When the input signal S goes high (at any time, **irrespective of what the clock is doing**), a timer is started. The timer outputs two 1-cycle pulses: pulse A starts with the first rising clock edge that occurs immediately after S goes high, while pulse B starts on the fourth rising edge of clock after S goes high. A block diagram and timing diagram below illustrate the desired behaviour. (Ignore any metastability issues that this problem might face. Also, assume that S will not go low again until after the timer is finished, ie after the falling edge of B.)




```
library IEEE;
use ieee.std_logic_1164.all;

entity q2 is
  port( S, clk : in std_logic;
        A, B   : out std_logic );
end q2;

architecture behavioural of q2 is
```

```
begin
  signal blah : std_logic;
  process(S)
      if S='1' then
          blah='1';
      end if;
end process;
 process(clk)
  variable counter : std_logic_vector := "00";
    if rising_edge(clk) then
        if blah='1' and counter="00" then
            A='1';
        elsif blah='1' and counter="10" then
            B='1'; counter="00";
        else
            A='0';
            B='0';
        end if;

        counter=counter+1;
end process;
```

this was his solution which seems like bad form because both clk & s are in the same sensitivity list

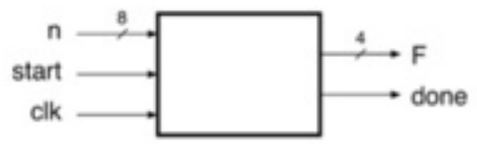
```
process(clk, s)
begin
    if s='0' then
        count <="1000";
    else
        if rising_edge(clk) then
            count<='0' & count(3 downto 0);
            A<=count(3);
            B<=count(0);
        end if;
    end if;
end process;
```

P1Q6

Given an 8-bit unsigned integer n, design a circuit that calculates a 4-bit unsigned integer F, the *square root* of n. i.e. $F = \sqrt{n}$. The algorithm is given below.

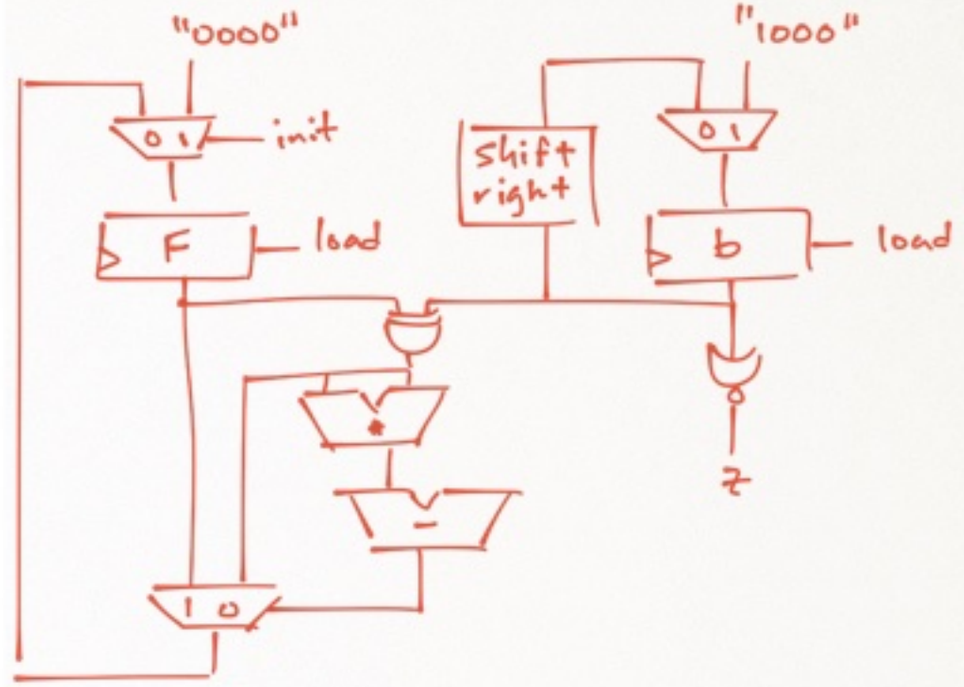
```
b := "1000"          -- b indicates which bit to examine
F := "0000"          -- builds up solution in F
while b != 0
    F := F XOR b     -- sets bit in F
    if (F*F > n) then
       F := F XOR b  -- clears bit from F
    end if
    b := b / 2       -- shifts b right by 1 position
end while
```

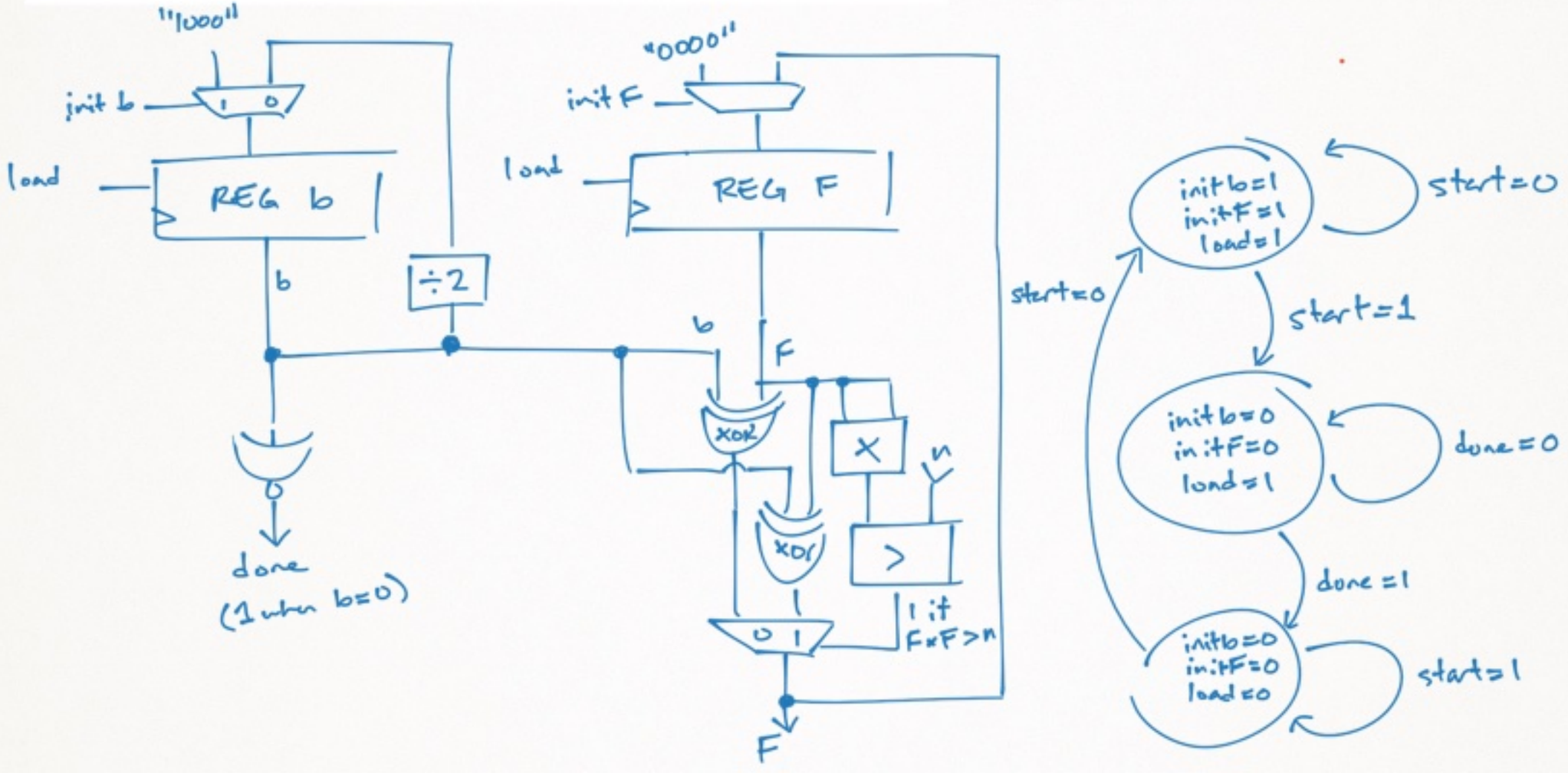
The inputs and outputs of your circuit are as shown below.



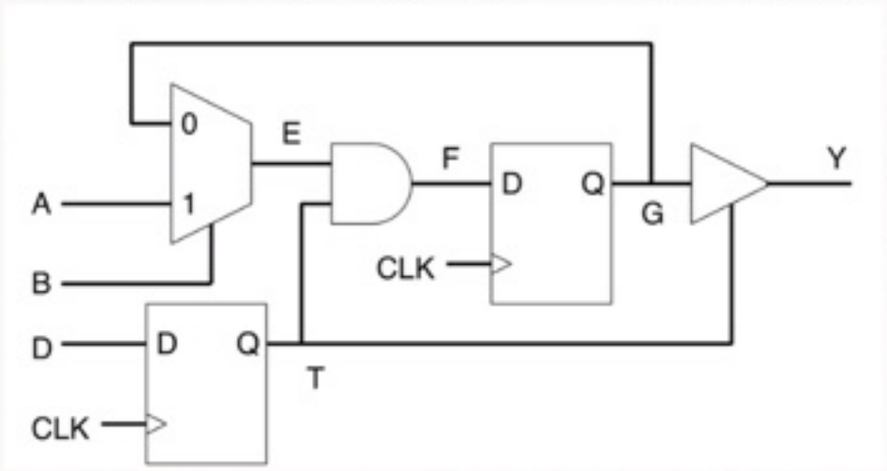
The user must supply an 8-bit value on the input bus **n**, and assert **start** (in the same clock cycle). The circuit will take several clock cycles to calculate the answer. When the circuit has computed the square root, it will supply this value on the output bus **F**, assert **done** and wait until the **start** signal is returned to 0. This is the same timing behaviour as the datapath circuits we talked about in class.

You must design your circuit using a datapath controlled by a state machine. Give your answer in terms of **one or more schematics (for the datapath) and/or one or more state diagrams (for the controlling state machine)**. Marks will be deducted for solutions that are overly large or complex.





MIQ1 write VHDL with same behaviour as circuit

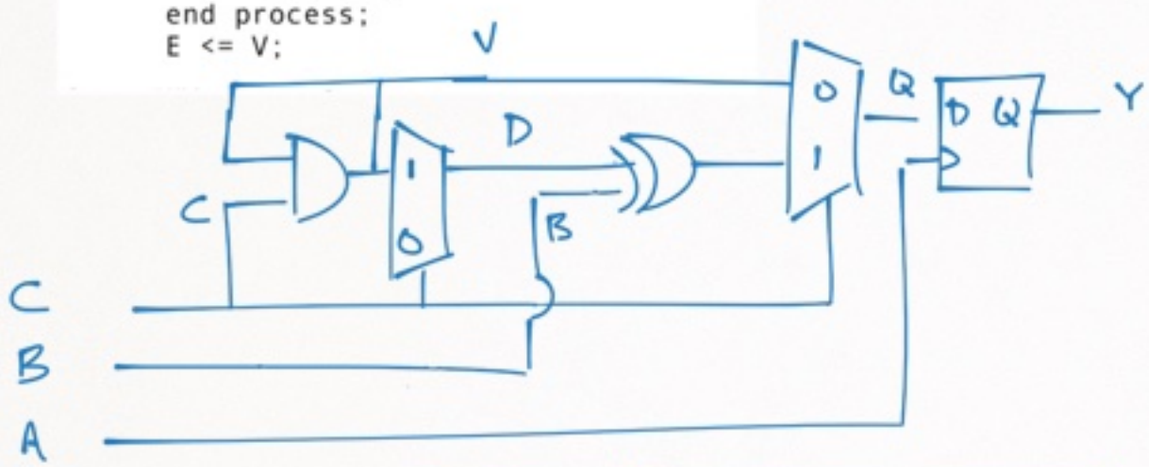


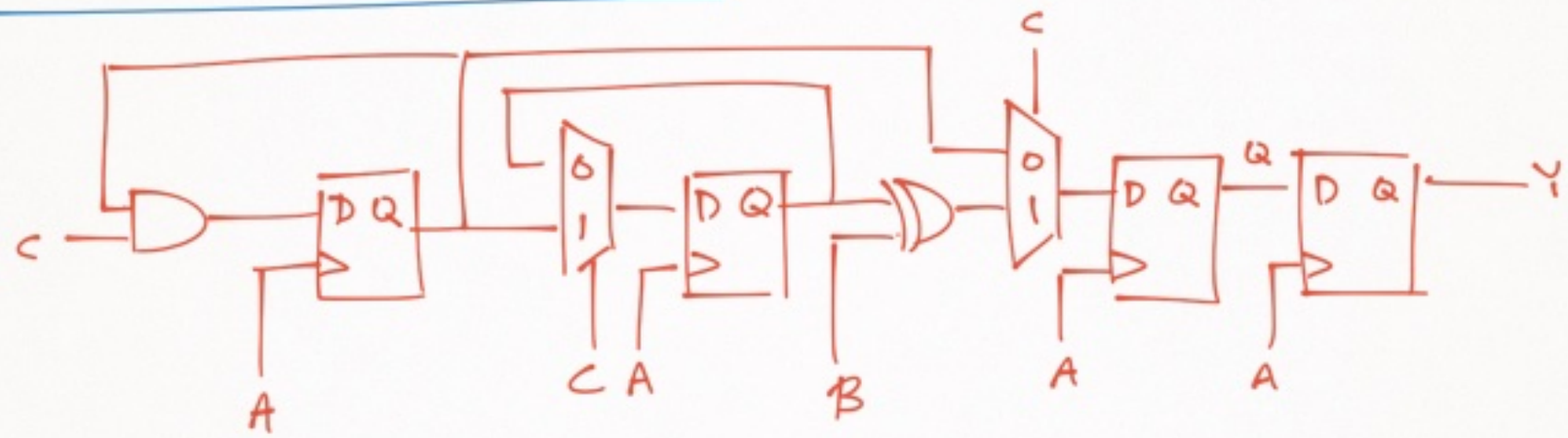

```
library ieee;
use ieee.std_logic_1164.all;

entity CIRCUIT is
   portmap(A,B,D,E : in std_logic;
           Y : out std_logic
   )
end CIRCUIT;

architecture BEHAVIOURAL of CIRCUIT is

begin
process(clk)

begin
   if rising_edge(clk) then

      if B='1' then
         E:= A;
      else
         E:= G;
      end if;
      G:= E and T;

      if T='1' then
         Y:= G;
      else
         Y:= 'X';
      end if;
      T:= D;

if B='1' then
   F:= T and A;
else
   F:= T and G;
end if;
G<= F;

if D='1' then
   Y<= F;
else
   Y<='Z';
end if;
T<= D;
```

M1Q2

```
entity CIRCUIT2 is
      port (  A,B,C : in  std_logic;
              Y     : out std_logic  );
end CIRCUIT2;
architecture BEHAVIOURAL of CIRCUIT2 is
      signal   D,E,Q : std_logic;
      signal   V     : std_logic;
begin
      process(A)
      begin
            if rising_edge(A) then
                  V <= C and E;
                  if C = '1' then
                        D <= V;
                        Q <= B xor D;
                  else
                        Q <= V;
                  end if;
                  Y <= Q;
            end if;
      end process;
      E <= V;
```

Draw schematic for this VHDL





MIQ3 transform algorithm into a circuit

```
A=0; B=1; C=1;
while(N>0){
    C=A+B;
    A=B; B=C;
    N=N-1;
}
OUT=C;
```