

VHDL – VHSIC Hardware Description Language

VHSIC – Very High Speed Integrated Circuit

Describes hardware, not a programming language

Originally used to model and simulate system behavior

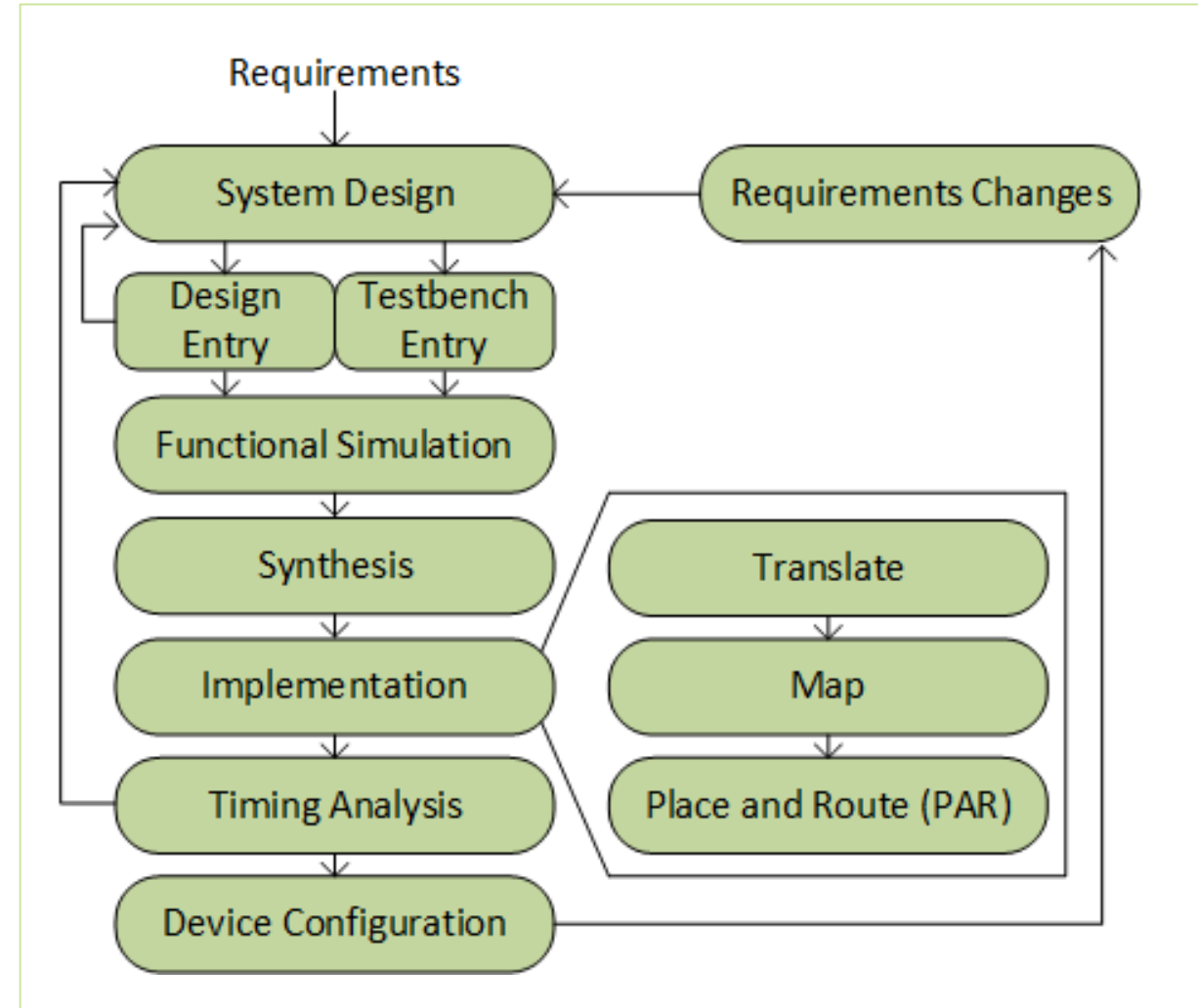
- Designers still designed with gates

In early 90s tools developed to ‘synthesize’ description into hardware

- Became standard for logic designers
- Remember - Not all VHDL models can be synthesized efficiently (if at all)

With VHDL, a simple logic block or a full ASIC can be:

- Modeled
- Simulated
- Synthesized
- Implemented



HDL Based Design Flow

Requirements Specification

- Customer develops a software requirements document full of TBDs. Often the designers must help the customer develop the specification

System Design

- Requirements breakdown, trade matrices, white boards, ..
- Could include software, hardware, firmware, mechanical engineers.
- Module interfaces are laid out.

Design Entry

- Code!

Test Bench Entry

- Typically a different person from the coder.

Functional Simulation

- Simulate the VHDL code via Modelsim or other tool.

Synthesis

- Analogous to compiling C code.

Implementation

- Translate – Merges incoming netlists and constraints
- Map – Fits the design into the available resources of the target device. Don't know which specific resources yet.
- Place and Route (PAR) – Targets specific locations on the part to ensure timing closure.

Timing Analysis

- Post place and route simulation contains detailed timing information for the design
- Tends to take very long to simulate due to very small injected propagation delays.

Device Configuration

- Program the .bit file onto chip of choice

Customer Changes Requirements – Scope Creep



Entity and Architecture Concept

From the top

- System – collection of modules
- Modules contain:
 - Entity – only exposes external interfaces for the architecture. (wrapper)
 - Architecture – detailed description of the internal structure or behavior of a module

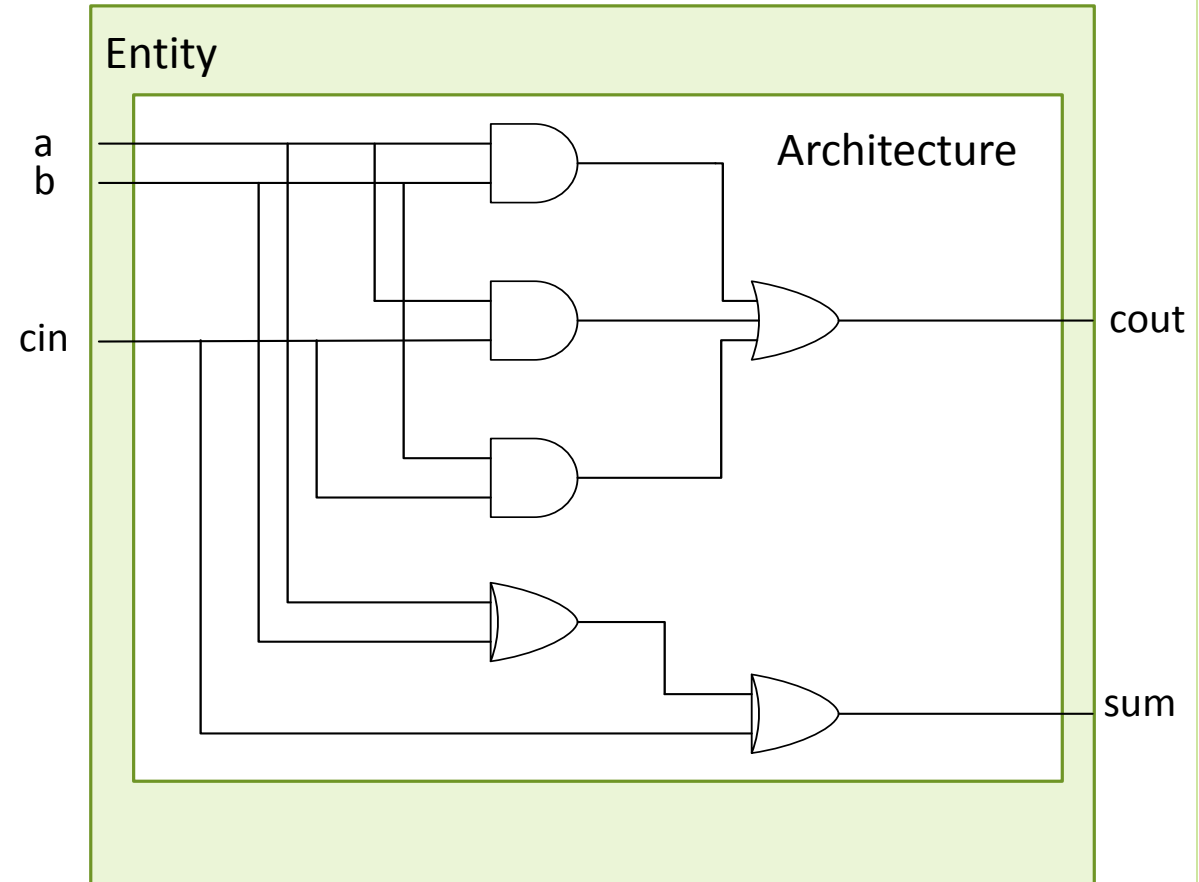
```
-- Dr. Kaputa  
-- single bit full adder
```

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

```
entity full_adder is  
  port (  
    a      : in std_logic;  
    b      : in std_logic;  
    cin    : in std_logic;  
    sum    : out std_logic;  
    cout   : out std_logic  
  );  
end full_adder;
```

```
architecture arch of full_adder is  
begin  
  sum <= (a xor b) xor cin;  
  cout <= (a and b) or (b and cin) or (cin and a);  
end arch;
```

3 inputs			2 outputs	
A_1	B_1	C_{in}	Σ_1	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



Hierarchy – Full Adder [2 bit]

VHDL supports hierarchical Design

One architecture can reference another entity

```
-- Dr. Kaputa
-- two bit full adder

library ieee;
use ieee.std_logic_1164.all;

entity full_adder_2bit is
    port (
        a      : in std_logic_vector(1 downto 0);
        b      : in std_logic_vector(1 downto 0);
        sum     : out std_logic_vector(1 downto 0);
        cout    : out std_logic
    );
end full_adder_2bit;

architecture arch of full_adder_2bit is
    component full_adder is
        port (
            a      : in std_logic;
            b      : in std_logic;
            cin     : in std_logic;
            sum     : out std_logic;
            cout    : out std_logic
        );
    end component;

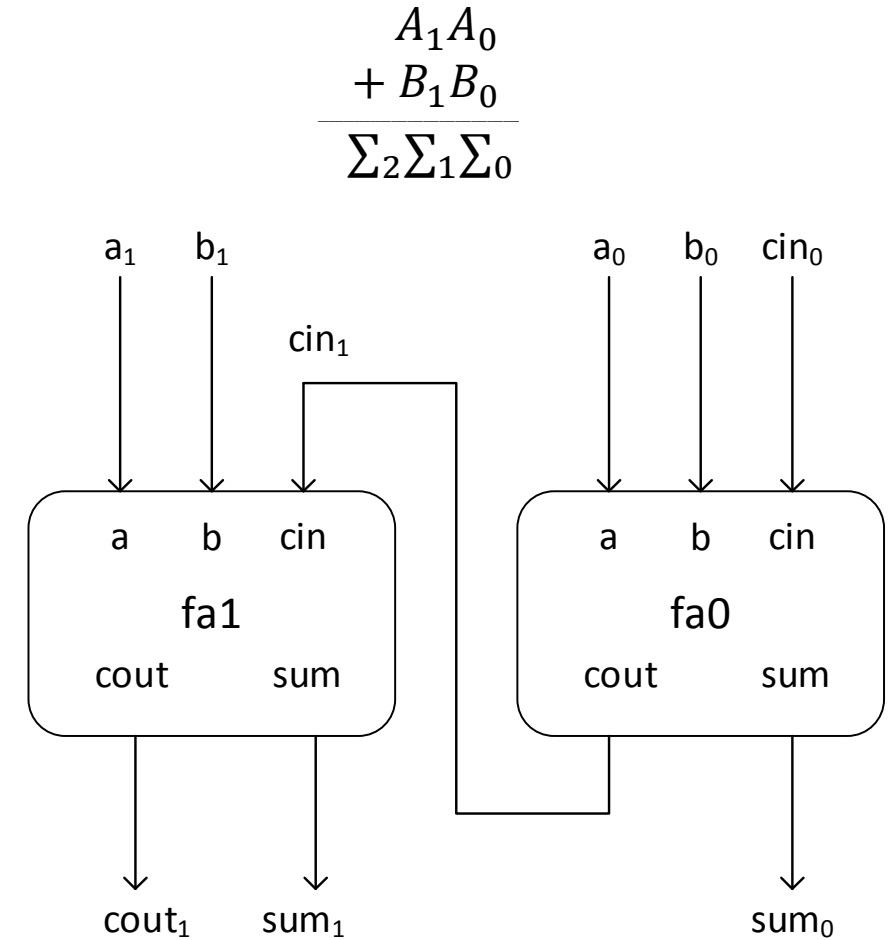
    -- Implementation of the 2-bit full adder using two 1-bit full adders
    -- fa0 handles the least significant bit (LSB) and fa1 handles the most significant bit (MSB)
    -- The carry-out of fa0 (cout_0) is connected to the carry-in of fa1 (cin_1)
    -- The final carry-out (cout) is the carry-out of fa1 (cout_1)
    -- The final sum (sum) is the concatenation of sum_1 and sum_0
    -- fa0: full_adder
    -- port map(
    --     a => a(0),
    --     b => b(0),
    --     cin => cin(0),
    --     sum => sum(0),
    --     cout => cin_1
    -- );
    -- fa1: full_adder
    -- port map(
    --     a => a(1),
    --     b => b(1),
    --     cin => cin_1,
    --     sum => sum(1),
    --     cout => cout
    -- );
end arch;
```

```
signal cin : std_logic_vector(1 downto 0);

begin
    cin(0)    <= '0';

    fa0: full_adder
        port map(
            a      => a(0),
            b      => b(0),
            cin     => cin(0),
            sum     => sum(0),
            cout    => cin_1
        );

    fa1: full_adder
        port map(
            a      => a(1),
            b      => b(1),
            cin     => cin_1,
            sum     => sum(1),
            cout    => cout
        );
end arch;
```



Structural vs. Behavioral VHDL

Structural Architecture

- Describes a system in terms of “what it is”
- contains
 - *signal declarations*, for internal interconnections
 - *component instances*
 - instances of previously declared entity/architecture pairs
 - *port maps* in component instances
 - connect signals to component ports

Behavioral Architecture

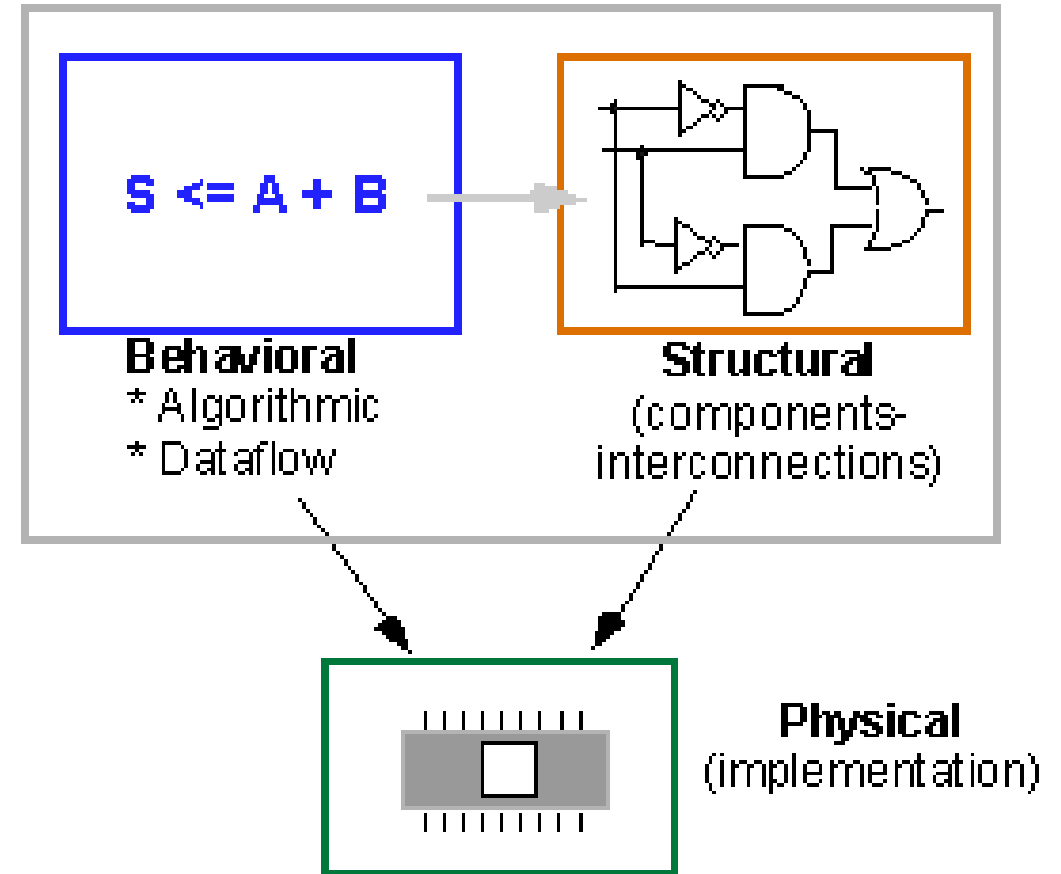
- Describes a system in terms of “what it does”
- Contains
 - Algorithms more abstracted away from hardware instantiation.
 - How is an adder or a multiplier inferred with my version of the tools?

VHDL – 87 Method

- Can't directly instantiate entity/architecture pair
- Instead
 - include *component declarations* in structural architecture body
 - instantiate components

VHDL – 93 Method

- `uut : entity work.full_adder(arch)`



Testbenches Overview

Testbench = VHDL entity that applies stimuli (drives the inputs) to the Design Under Test (DUT) and (optionally) verifies expected outputs.

Results viewed in a wave window or written to a file.

Since *Testbench* is written in VHDL, it is not restricted to a single simulation tool (portable).

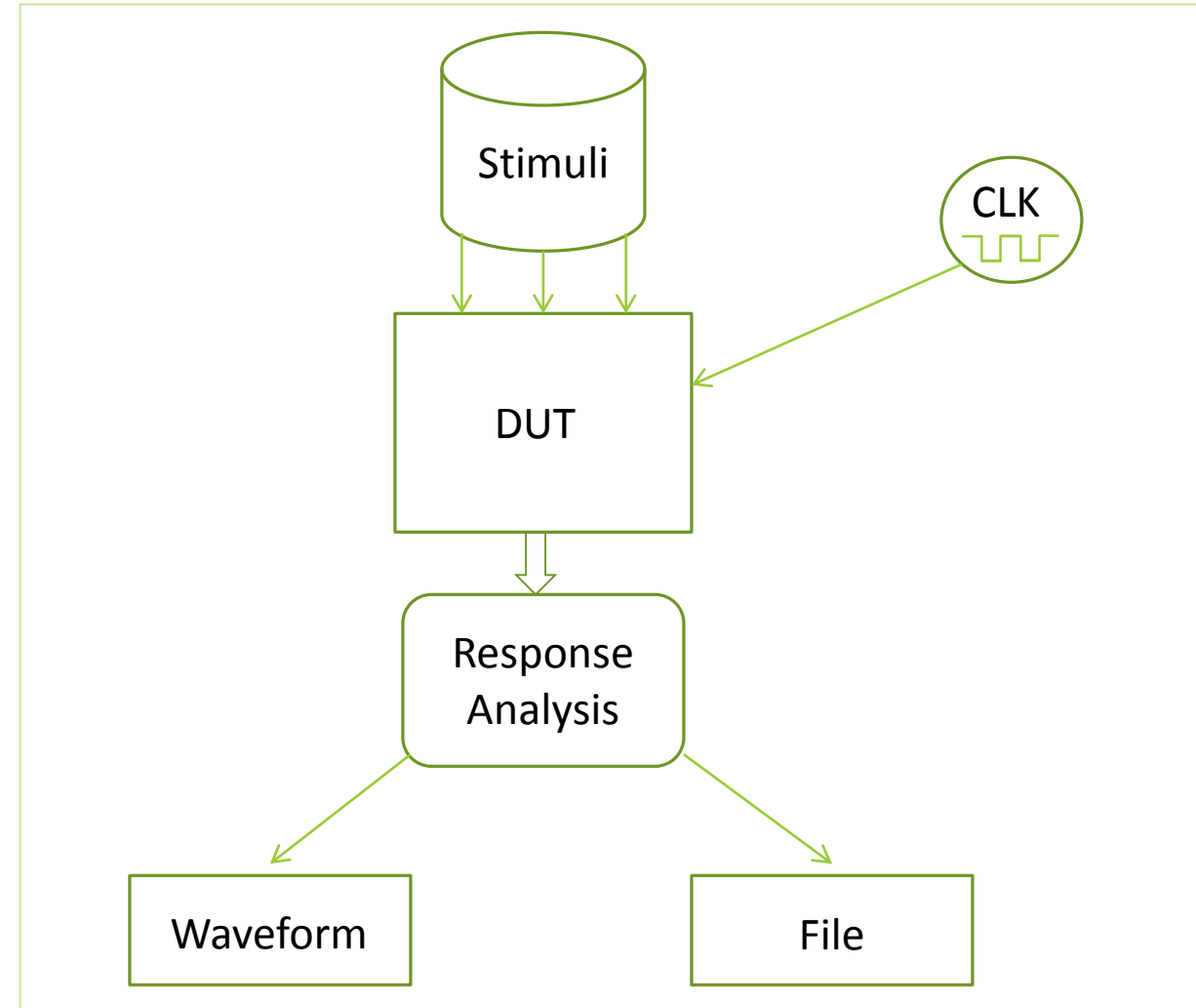
The same *Testbench* can be easily adapted to test different implementations (i.e. different *architectures*) of the same design.

Does not need to be synthesizable

- Can use behavioral models, time, variables, etc..

Does not contain any ports to outside

- Entity is empty
- All necessary stimuli is created in the testbench



Testbench Anatomy

```
-----  
-- Dr. Kaputa  
-- delta time test bench  
-----  
  
library ieee;  
use ieee.std_logic_1164.all;  
  
entity delta_time_tb is  
end delta_time_tb;  
  
architecture beh of delta_time_tb is  
  
    signal sig1 : std_logic;  
    signal sig2 : std_logic;  
    signal sig3 : std_logic;  
    signal a     : std_logic;  
    signal b     : std_logic;  
    signal c     : std_logic;  
    signal d     : std_logic;  
    signal e     : std_logic;
```

Comments

Libraries

Entity with no ports

Signal declarations
'can have any name'

```
component delta_time is  
    port (  
        a      : in std_logic;  
        b      : in std_logic;  
        c      : in std_logic;  
        d      : in std_logic;  
        e      : out std_logic;  
    );  
end component;  
  
begin  
    a <= '0', '1' after 6 ns;  
    b <= '0', '1' after 5 ns, '0' after 8 ns;  
    c <= '0', '1' after 7 ns;  
    d <= '0';  
  
    uut: delta_time  
        port map(  
            e => e,  
            a => a,  
            b => b,  
            c => c,  
            d => d  
        );  
end beh;
```

Declare component of the entity being tested. Ports and name must match exactly.

Signal assignments can be concurrent or in a process

Map the internal signals to the entity or unit under test (UUT)

Clock and Reset Generation

Concurrent assignment

```
Clk_tb <= not (clk_tb) after PERIOD/2;
```

- Remember to initialize clk_tb

Process

```
Clk_gen: PROCESS
```

```
  BEGIN
```

```
    clk_tb <= '0';
```

```
    WAIT FOR 3* PERIOD/4;
```

```
    clk_tb <= '1';
```

```
    WAIT FOR PERIOD/4;
```

```
  END PROCESS;
```

25% duty cycle

To generate a one time signal such as reset, use process

```
Reset_generator: PROCESS
```

```
  BEGIN
```

```
    reset_n_tb <= '0';
```

```
    WAIT FOR 2*PERIOD;
```

```
    reset_n_tb <= '1';
```

```
    WAIT;
```

```
  END PROCESS;
```

Do not assign any other signals in clk or reset processes



Report Analysis

Self-checking Testbench

- Results are compared against 'known' results

Assert statements

- An 'assert' checks a condition and will report a message if the condition is false
- Four severity levels
 - Note
 - Warning
 - Error (default)
 - Failure
- Note and warning – simulation resumes
- Error and Failure – simulation stops

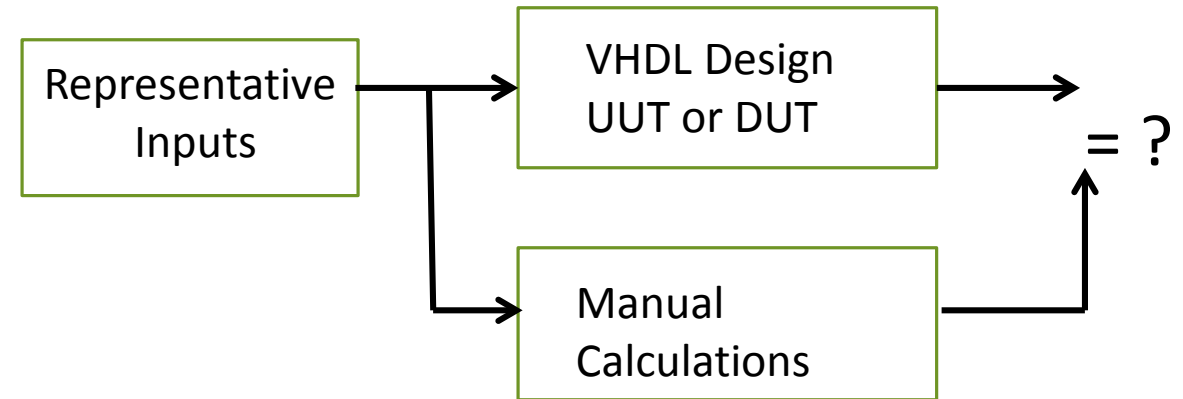
Syntax

- (Label) : ASSERT condition
REPORT expression
SEVERITY expression ;

This is a condition
that should be true

Example:

```
(Label) : ASSERT condition  
      (REPORT expression)  
      (SEVERITY expression);
```



Why?

- Assert statements checks the outputs from the UUT against known values to verify correct operation
- As designs become more complex it is not reasonable to check outputs visually

Assert Placement

Assert can be concurrent

- Will be executed when one of the signals in its condition clause has an event
- This can cause problems
 - Assert (A_tb = Y1_tb) AND (A_tb = not(Y0_tb))
 - If the event on A is supposed to change Y1 and/or Y0 there needs to be time for those signals to change (remember delta delays?)
 - If Y0 and Y1 haven't changed yet, the assert will fail
- Best to avoid concurrent assert statements

Assert can be in a process

- This works best for combinational logic
- Notice the sensitivity list.
 - Do not evaluate the condition unless the OUTPUT has changed.

ex: process(f_tb)

begin

 assert f_tb = ((a_tb and b_tb) or c_tb)

 report "incorrect results"

end process;

This is different than a sensitivity list for synthesizable logic

Assert can be in a the stimulus process loop

- Delta delays will not effect assertions.
- This is the preferred way to do assertion based testing.

```
tb : process
begin
  for i in 0 to 3 loop
    a <= std_logic_vector(unsigned(a) + 1 );
    for j in 0 to 3 loop
      b <= std_logic_vector(unsigned(b) + 1 );
      wait for 10 ns;

      ASSERT (unsigned(a) + unsigned(b) = unsigned(sum))
      report "a: " & integer'image(to_integer(unsigned(a))) & CR &
        "b: " & integer'image(to_integer(unsigned(b))) & CR &
        "sum: " & integer'image(to_integer(unsigned(sum))) & CR;
    end loop;
  end loop;
end process;
```

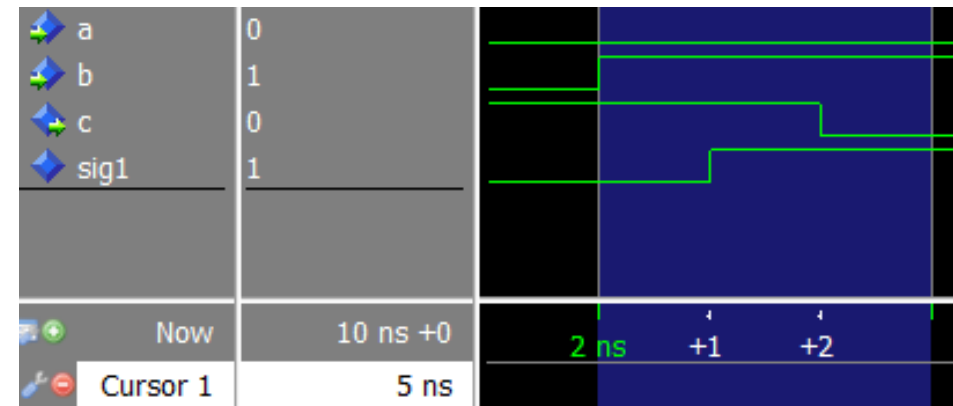
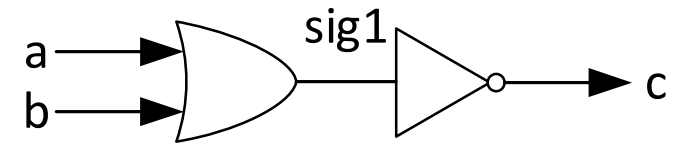
Event Driven Simulation

Simulation

- The process of conducting experiments on a model of a system for the purpose of understanding or verifying the operation of the actual system.
 - Simulation time
 - Not real time, its the time value maintained by the simulator
 - Determines when changes in input and output signals occur
- ```
graph TD; A([Get next signal assignment from the time queue]) --> B([Make assignment and determine if an event occurs]); B --> C([Execute processes triggered by this event]); C --> A;
```
- Transaction - simulation loop updates the value of a signal
  - Event – transaction where the value of the signal changes
  - Time queue:
    - A<= '1' at 2 ns, B <= '0' at 4 ns, C <= '1' at 10 ns
    - A: ('1', 2 ns), B: ('0', 4 ns), C: ('1', 10 ns) [representation on time queue]

Delta Time – An infinitesimal interval used by the simulator to maintain assignment ordering for assignments that take place at the same simulation time.

```
sig1 <= a or b;
c <= not sig1;
```



# Processes

A 'process' is a concurrent statement. Sequential statements can only be used within a process.

Statements are executed 'sequentially' within a process until the process is suspended either via a 'wait' statement or until there are no more statements to be executed.

A process can be triggered by resumption after a wait statement or by an event on a signal in its sensitivity list:

```
process
 begin
 a <= '1';
 wait on a;
 end process;
```

```
process(a)
 begin
 a <= '1';
 end process;
```

Above code snippets are equivalent

What is the value of  $a$  at time 0?

## Process Rules

- If a process has a sensitivity list, then it cannot contain a 'wait' statement.
- A process with a sensitivity list is always triggered at time 0 because all signals always have an initial event placed on them at time 0.
- If a process without a sensitivity list 'falls out the bottom' then it immediately loops back to the top until it hits a wait statement.

```
process
 begin
 a <= '1';
 end process;
```

```
process
 begin
 a <= not (a);
 end process;
```

- These processes generate infinite loops because they will 'fall out the bottom', loop back, and never encounter a wait statement.
- You will get a compiler warning about this – if you execute it, the Modelsim simulator may hang.

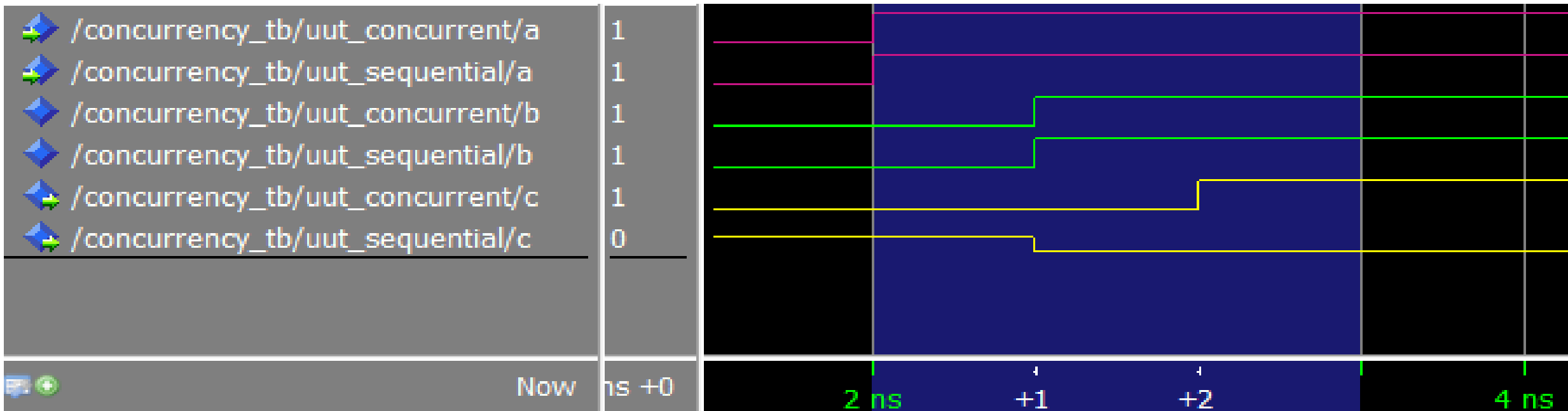
# VHDL Concurrency

## Concurrent Statements

```
begin
b <= a;
c <= b;
end beh;
```

## Sequential Statements

```
begin
process(a)
 begin
 b <= a;
 c <= b;
 end process;
end beh;
```



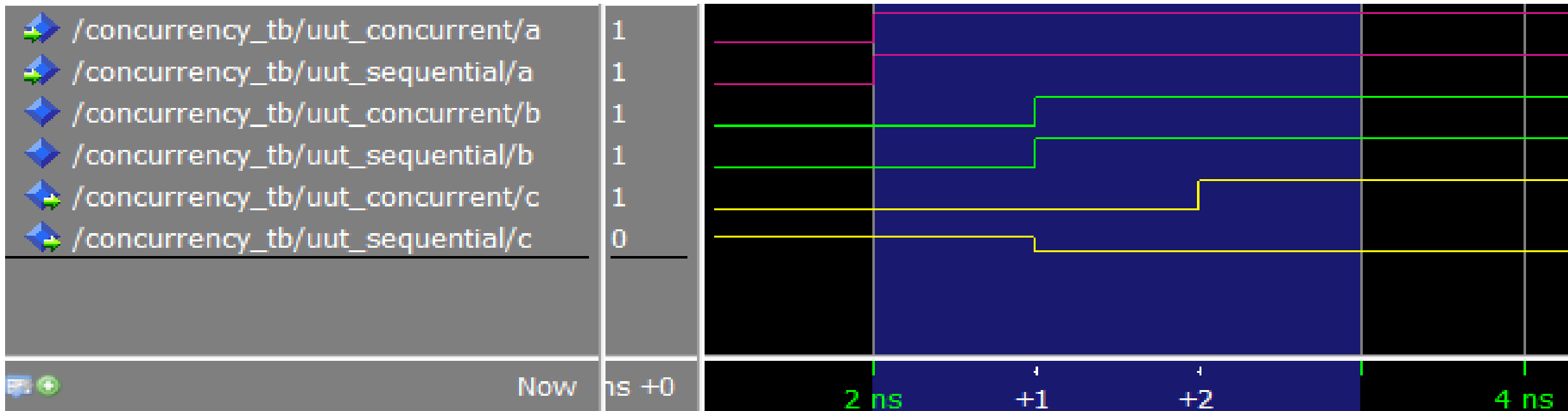
# VHDL Concurrency

## Concurrent Statements

- Concurrent statements execute whenever events on signals used by the concurrent statements trigger them.

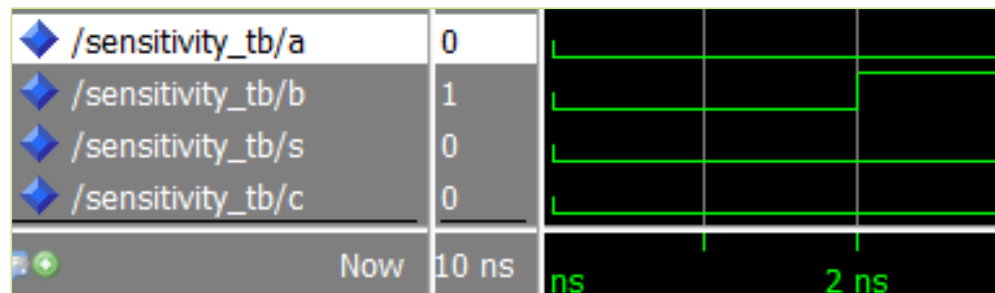
## Sequential Statements

- Any signal assignment becomes effective only when the process suspends. Until that moment, all signals keep their old values.
- Only the last assignment to a signal will be effectively executed. Therefore, it would make no sense to assign more than one value to a signal in the same process.

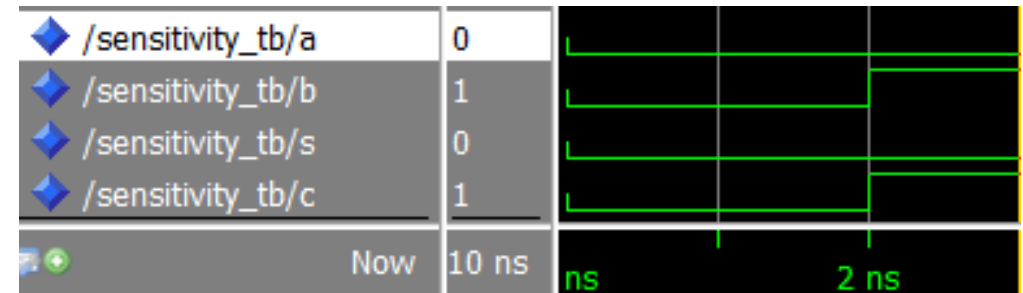


# Sensitivity Lists

```
begin
process(s,a)
 begin
 if (s = '1') then
 c <= a;
 else
 c <= b;
 end if;
 end process;
end beh;
```



```
begin
process(s,a,b)
 begin
 if (s = '1') then
 c <= a;
 else
 c <= b;
 end if;
 end process;
end beh;
```



Sensitivity lists do not effect synthesis..... only simulation.

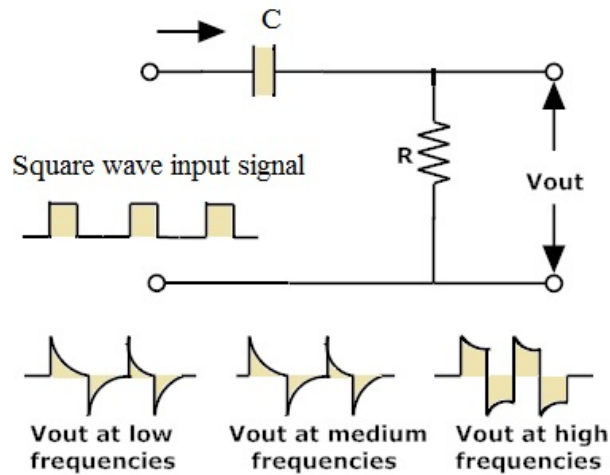
# VHDL Delay Models

## Inertial Delay

- A delay model used for switching circuits. Thus a pulse whose duration is shorter than the switching time of the circuit will not be transmitted.
- Default mode for signal assignment statements.

## Transport Delay

- Delay characteristic of hardware devices such as transmission lines that exhibit infinite frequency responses.
- Any pulse is transmitted no matter how short its duration.



VHDL 1993 enables specification of pulse rejection width via the ***reject*** command.

The following three assignments are equivalent to each other:

```
Out_2 <= Input after 2 ns;
Out_2 <= inertial Input after 2 ns;
Out_2 <= reject 2 ns inertial Input after 2 ns;
```



# Signal Assignment Rules

A signal assignment within a process will add an ordered pair to the time queue. There may already be assignments to this signal on the time queue.

- If the new assignment time is AFTER the other assignment times, then the new assignment pair is added to the end of the list.
- Any ordered pairs for this signal on the assignment list that have times LATER than the new assignment are removed from the time queue.
- A signal assignment pair can only be executed by the simulator after the process suspends. The last assignment to a signal in a process takes precedence.

```
begin
 a <= 1 after 1 ns,
 a <= 2 after 2 ns;
 a <= 3 after 3 ns;
 wait;
end process;
```

First  
statement is  
ignored.

```
pa: process
begin
 ta <= '1' after 3 ns,
 ta <= 'Z' after 4 ns;
 wait;
end process pa

pb: process
begin
 tb <= 'Z' after 4 ns,
 tb <= '1' after 3 ns;
 wait;
end process pb;
```

- Schedule of 'ta' assignment is put immediately on the time queue. Because 'z' assignment happens in time AFTER the '1' assignment then both assignments are valid
- In pb, because '1' assignment happens in time BEFORE the 'z' assignment, the 'z' assignment is ignored.
- Multiple delayed sequential statements are complicated.

```
pc: process
begin
 tc <= '1';
 tc <= '0';
 wait;
end process pc
```

What value does 'tc' take?

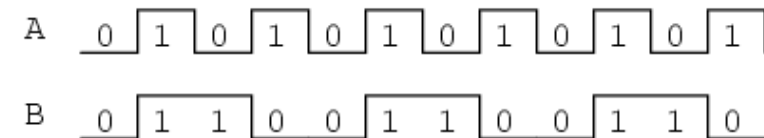
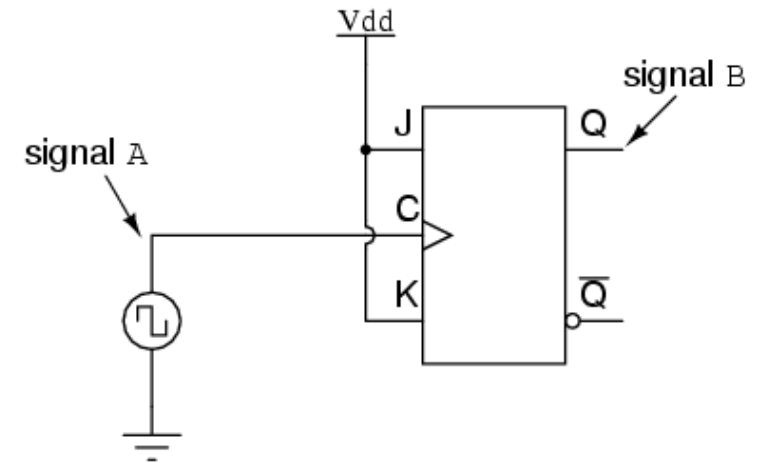
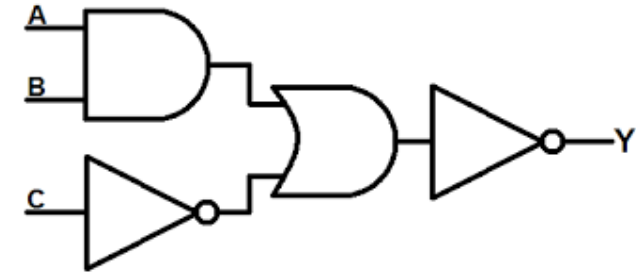
# Introduction to State Machines

## Combinational Circuits

- No internal memory
- Outputs are functions of
  - Inputs
- Ex: adder, multiplexer

## Synchronous Sequential Circuits

- Contains internal memory (Flip-flops)
- Outputs are functions of
  - Inputs, Internal state
- Sequential statements must occur within a process
- Ex: counter



# Introduction to State Machines

## State Machine

- Sequential circuit with “random” next state logic
- Digital system whose outputs are a function of its present input values and its past history of input values
- Used to Realize operations performed in a sequence of steps
- Information characterizing the effect of a FSM’s past history of input values is provided by its state.

Ex. Airplane Engine Control – active, standby, error



## Elements

- Input signals
- Next state function
- Symbolic states
- Output function
- Output signals



# State Machine Inputs

## Clock

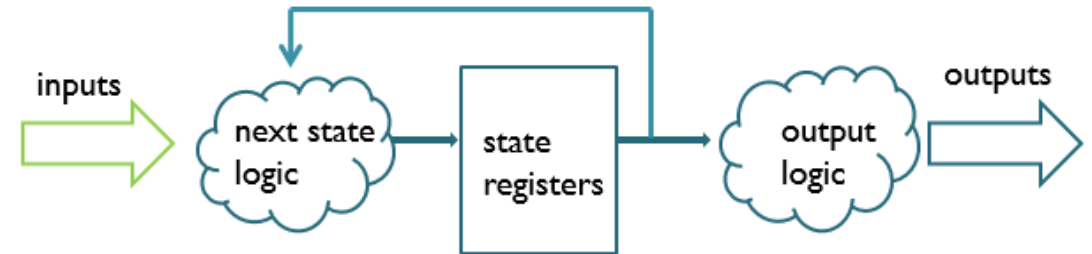
- state machines are synchronous and need a regular periodic clock input. All state changes happen on active clock edge

## Reset

- Puts the state machine in a known state
- Typically active low for DE2

## Other Inputs

- From outside the system
- From inside the system ... counters, etc,
- Everything sampled on clock edge



## Sample Inputs

- Air speed
- Altitude
- Fuel level
- Engine cmd
- Faults



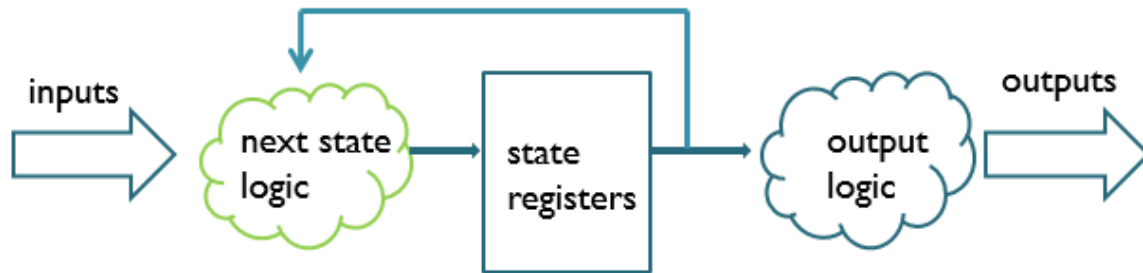
# Next State Logic and State Registers

The next state is computed by combinational logic

- Considers:
  - present state
  - Inputs

Next state is clocked into state registers on active clock edge

```
If (present_state = active) then
 if (faults = true) then
 next_state = standby
 end if
end if
```



Present State

- Provides information about its past input history
- Determines current outputs and next state
- Several different state encoding techniques are available

|       | Binary assignment | Gray code assignment | One-hot assignment | Almost one-hot assignment |
|-------|-------------------|----------------------|--------------------|---------------------------|
| idle  | 000               | 000                  | 000001             | 00000                     |
| read1 | 001               | 001                  | 000010             | 00001                     |
| read2 | 010               | 011                  | 000100             | 00010                     |
| read3 | 011               | 010                  | 001000             | 00100                     |
| read4 | 100               | 110                  | 010000             | 01000                     |
| write | 101               | 111                  | 100000             | 10000                     |



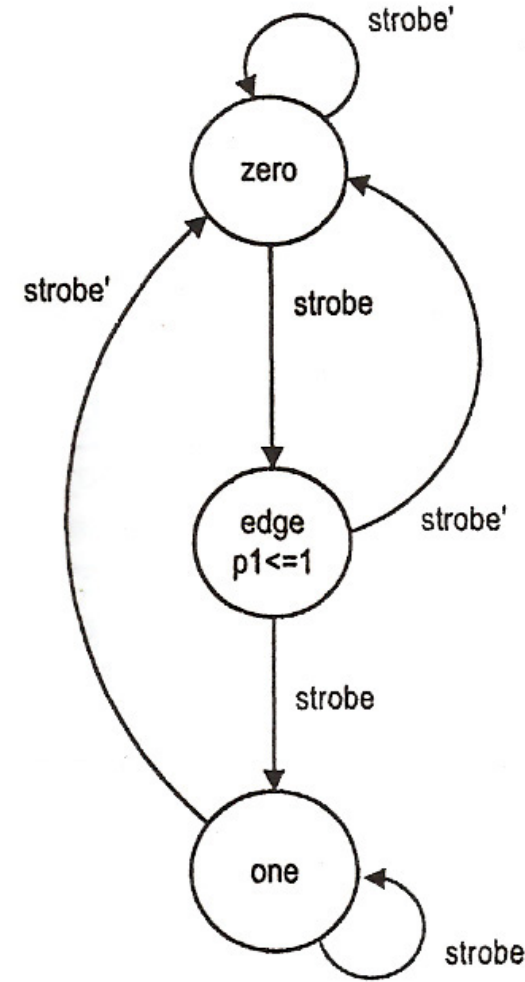
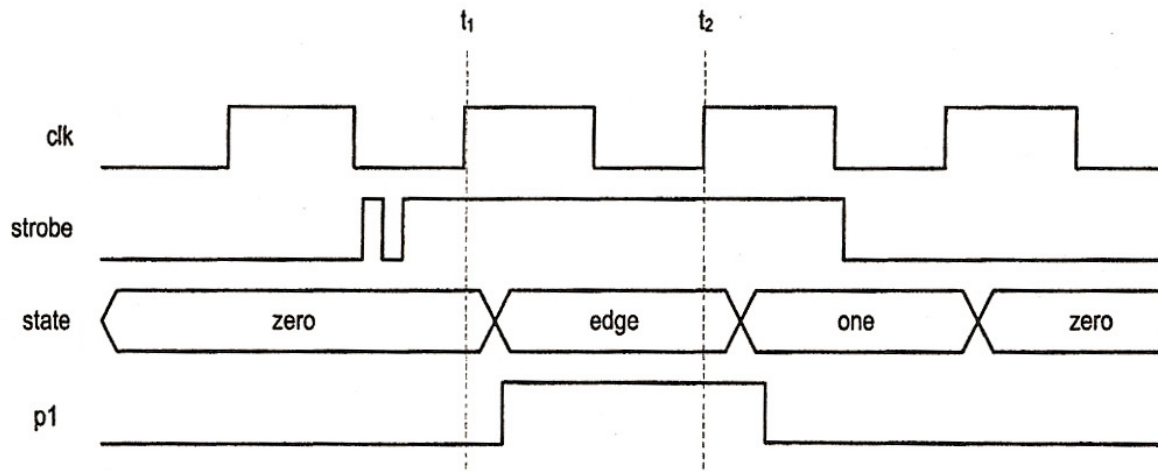
# State Machine Representations

## State Diagram

- States
- Transitions

## VHDL Code

- State registers
- Next state logic



```
-- state register
procStateRegister: process(reset,clk)
begin
 if (reset = '1') then
 stateReg <= zero;
 elsif(clk'event and clk = '1') then
 stateReg <= stateNext;
 end if;
end process;
-- next state logic
procStateNext: process(stateReg,strobe)
begin
 p1 <= '0';
 stateNext <= stateReg;
 case stateReg is
 when zero =>
 p1 <= '0';
 if (strobe = '1') then
 stateNext <= edge;
 end if;
 when edge =>
 p1 <= '1';
 if (strobe = '1') then
 stateNext <= one;
 else
 stateNext <= zero;
 end if;
 when one =>
 p1 <= '0';
 if (strobe = '0') then
 stateNext <= zero;
 end if;
 end case;
end process;
```

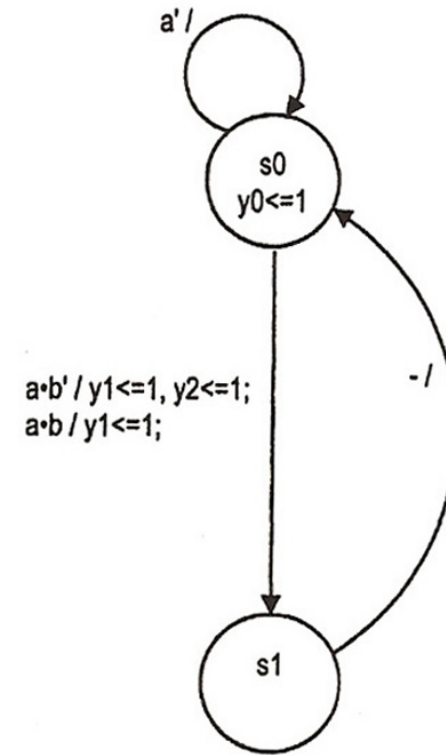
# State Machine Representations

## State Diagram

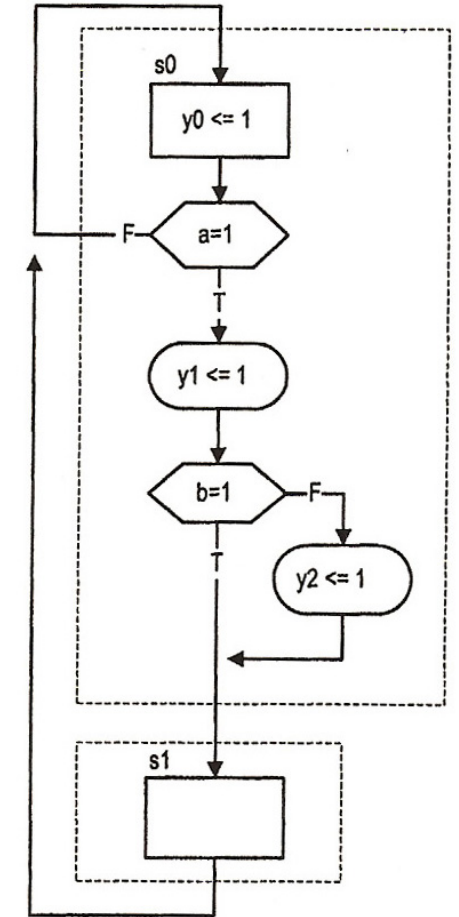
- Use for simple state transitions
- Use for few states

## Algorithmic State Machine (ASM) Chart

- Closely Resembles a Flow Chart
- Simplifies complex state transitions and Mealy outputs
- Use for more complex designs



State Diagram



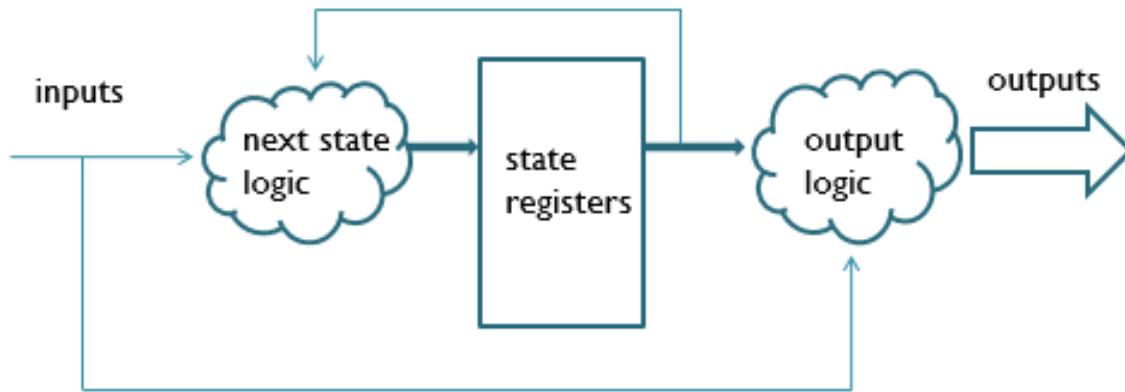
ASM Chart



# FSM Outputs - Mealy and Moore State Machines

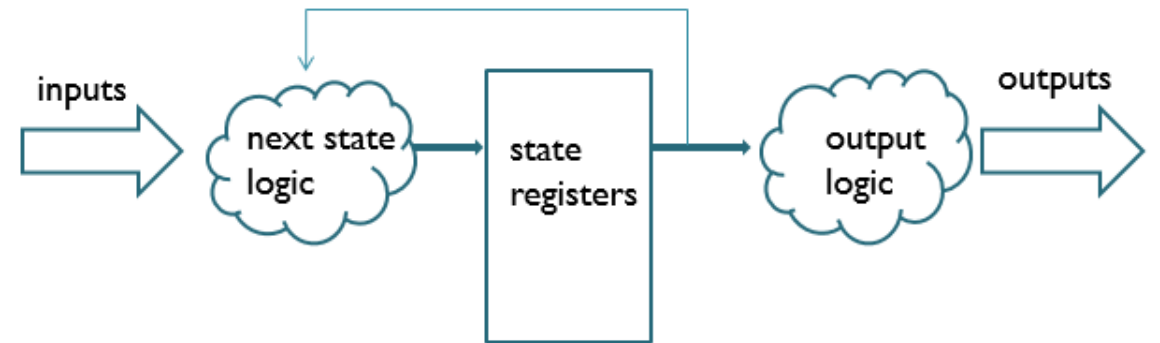
## Mealy

- Outputs dependent on current state and FSM inputs
- Can typically accomplish the same task as a Moore machine with fewer states
- Outputs react faster although they may contain glitches
- Use when every clock cycle in the design is needed



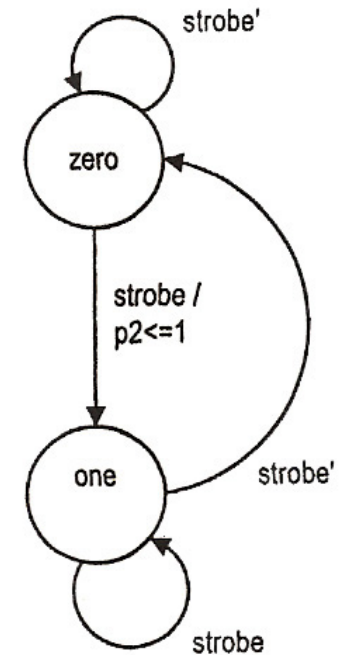
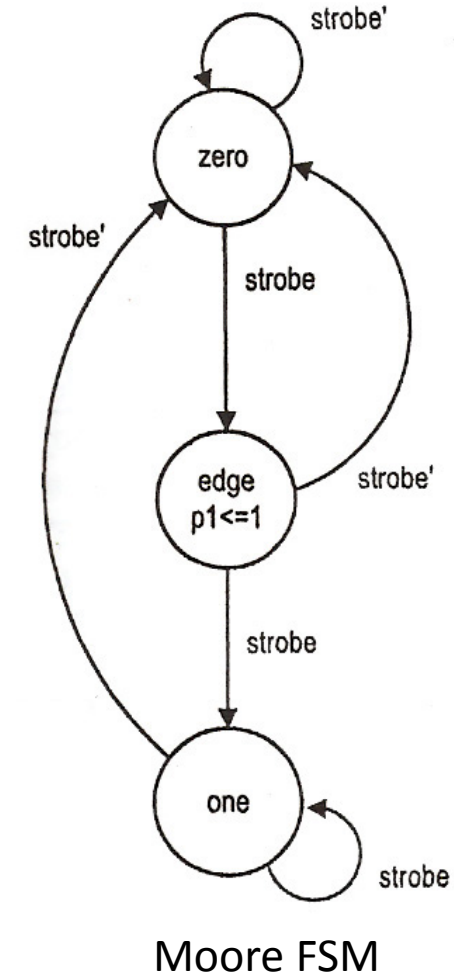
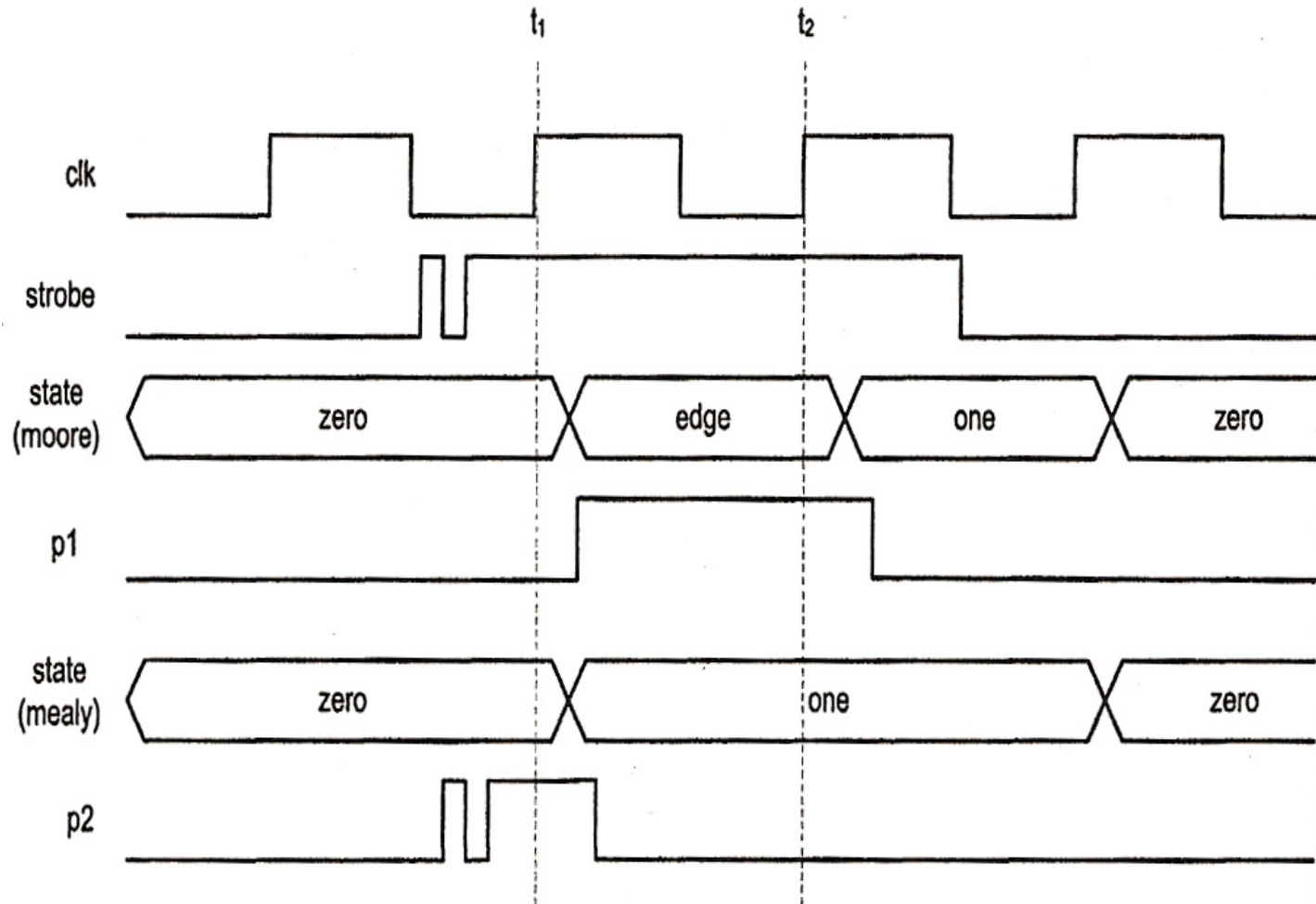
## Moore – Outputs dependent on current state only

- “Moore is Less”
- Outputs change at clock edge [one cycle later]
- Use when single clock delays and area are not key factors in the design





# FSM Outputs - Mealy and Moore State Machines



Mealy FSM

Moore FSM

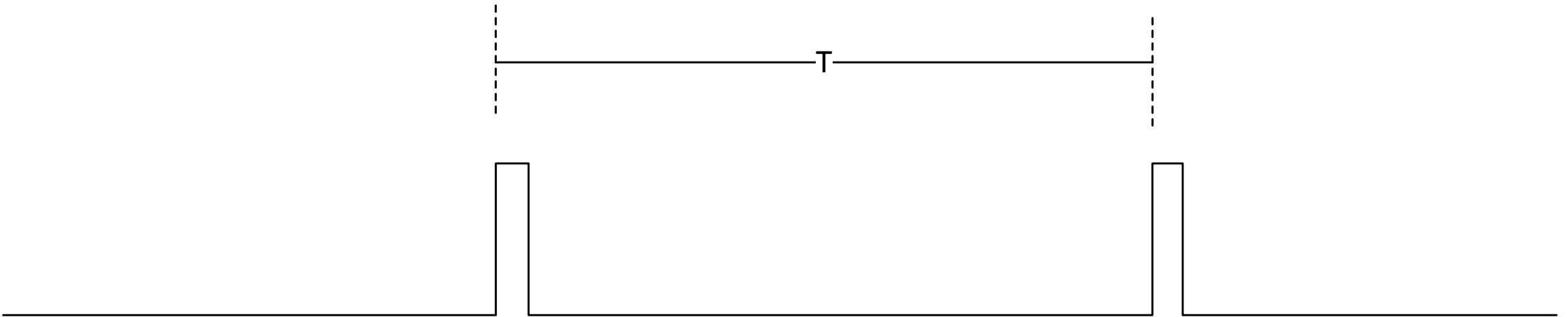
# Modular Design

## Problem Statement

- The design shall create a 20 ns wide pulse with a period of TBD  $\mu$ s



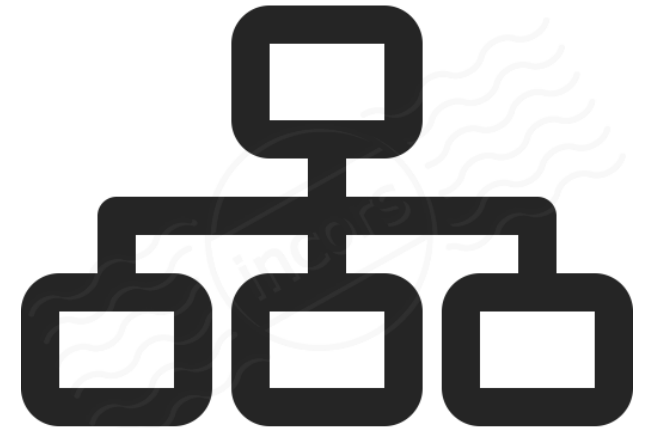
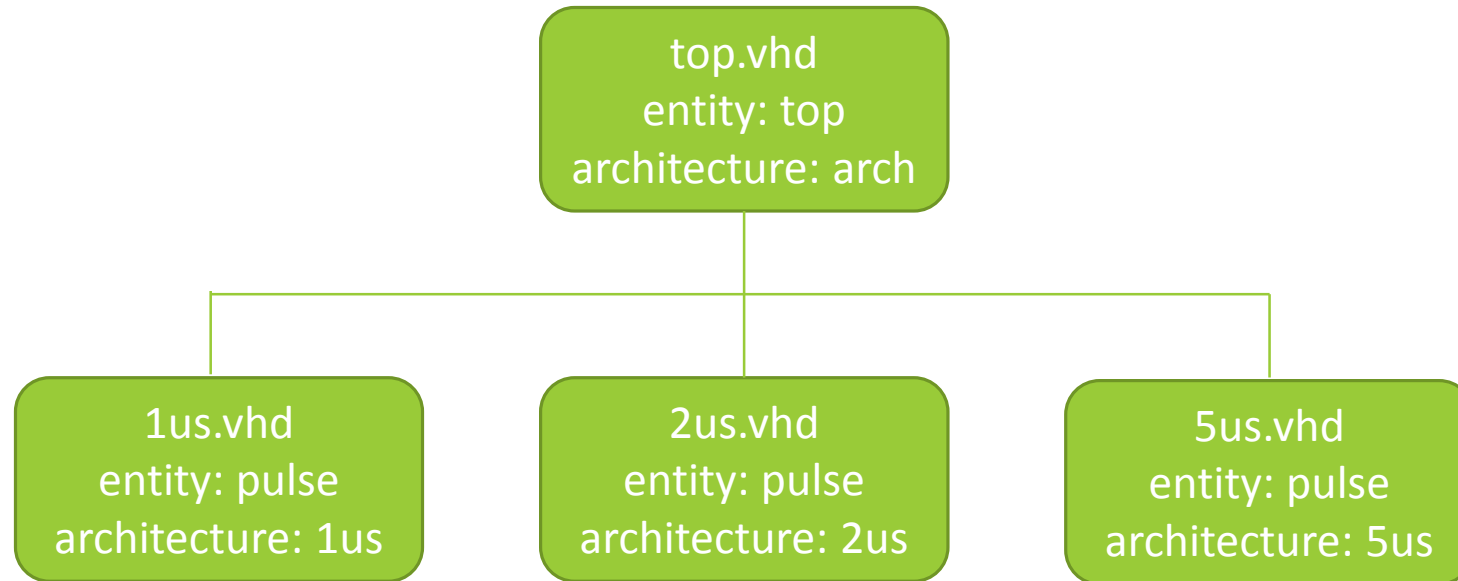
How does one design such a solution?



# Use Variable Component Hierarchy

Create a single entity with variable architectures

- Entity: only exposes external interfaces for the architecture
- Architecture: detailed description of the internal structure or behavior of a module

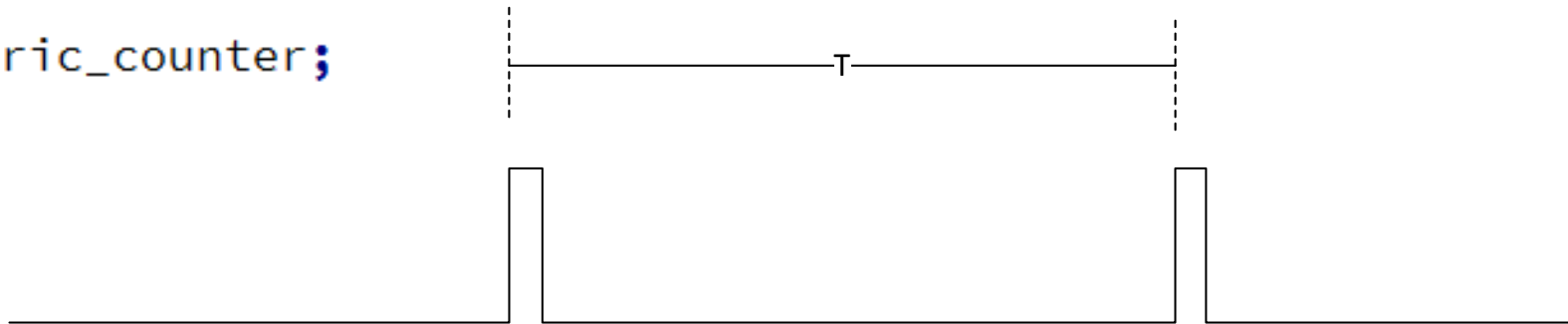


# Single Component with Variable Count Port

Assume the pulse period is limited to 100 pulses

- Need 7 bits to encode a value of 100
- What are the limitations to this design?

```
entity generic_counter is
 port (
 max_count : in std_logic_vector(6 downto 0);
 reset : in std_logic;
 clk : in std_logic;
 output : out std_logic
);
end generic_counter;
```



# Single Component with Generic Constant

---

Used to develop a module/function that can take on different parameters in different applications

- Example parameters:
  - Prop delay
  - Bit widths for address, registers, counters, etc.
  - Counter duration
- Default value assigned to generic when module/function is developed
- True value is assigned during component instantiation
  - This value can be different than the default value

**Generic Values are Substituted in Before Mapping to Logic Gates**



# Generic Counter Example Code

```

-- Dr. Kaputa
-- generic counter demo

library ieee;
use ieee.std_logic_1164.all;

entity generic_counter is
 generic (
 max_count : integer range 0 to 100 := 3
);
 port (
 reset : in std_logic;
 clk : in std_logic;
 output : out std_logic
);
end generic_counter;

architecture beh of generic_counter is

 signal count_sig : integer range 0 to max_count := 0;
```

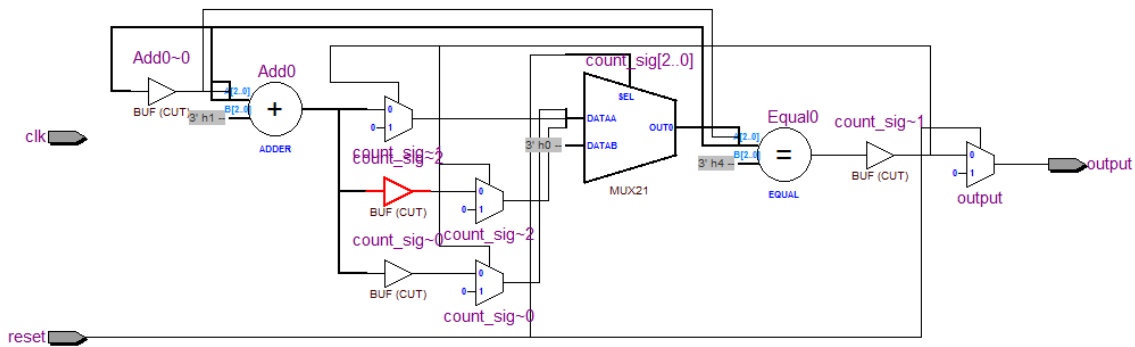
```
begin
process(clk,reset)
 begin
 if (reset = '1') then
 count_sig <= 0;
 output <= '0';
 elsif (clk'event and clk = '1') then
 if (count_sig = max_count) then
 count_sig <= 0;
 output <= '1';
 else
 count_sig <= count_sig + 1;
 output <= '0';
 end if;
 end if;
 end process;
end beh;
```



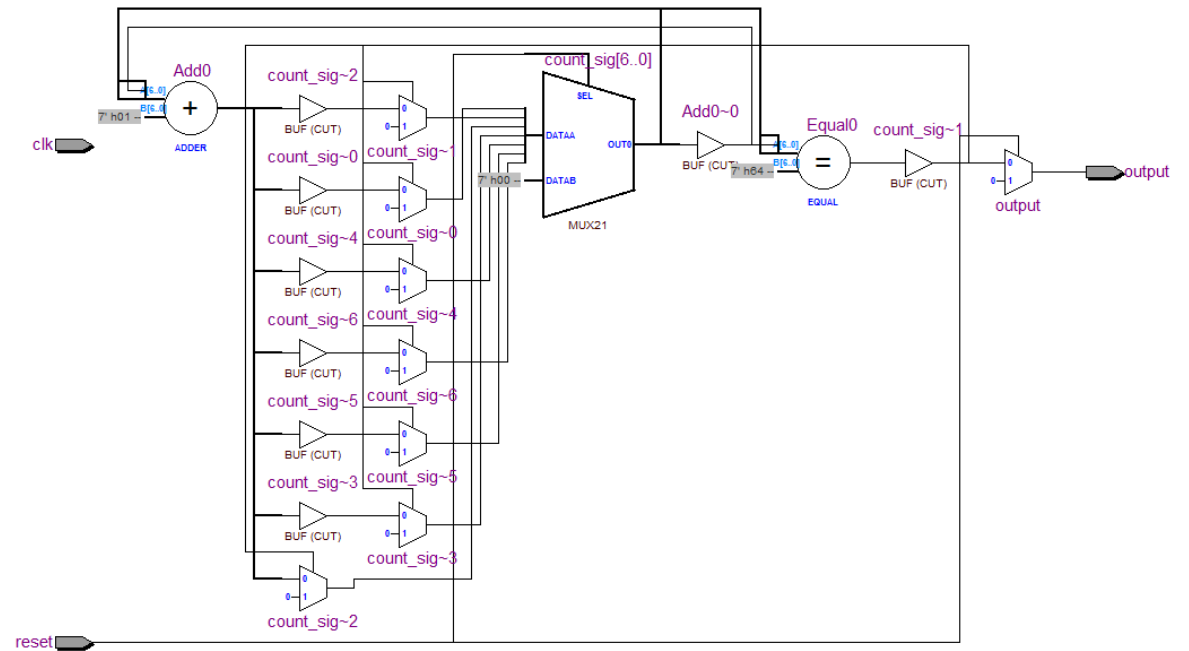
# RTL View Comparison

max\_count => 4

- Less logic is instantiated



max\_count => 100;



# Generic Counter Instantiation

Assign a value to the generic port

```
uut: generic_counter
 generic map (
 max_count => 4
)
 port map(
 reset => reset,
 clk => clk,
 output => output
);
end beh;
```

Leave generic port unassigned

```
uut: generic_counter
 port map(
 reset => reset,
 clk => clk,
 output => output
);
end beh;
```

What is the value of *max\_count*?





# Multiple Generics

A module can have more than one generic constant

- Example : a sizable n-bit register with programmable prop delay

Entity reg\_pd is

```
generic (t_pd : time;
 width : integer);
port (clk, reset_n : in std_logic;
 data_in : in std_logic_vector(width-1 downto 0);
 data_out : out std_logic_vector (width-1 downto 0)
End entity reg_pd;
```

Architecture behavioral of reg\_pd

```
begin
 store: process(clk, reset_n)
 begin
 if (reset_n = '0') then
 data_out <= (others => '0');
 elsif (rising_edge(clk)) then
 data_out <= data_in after t_pd;
 end if;
 end process;
end behavioral;
```



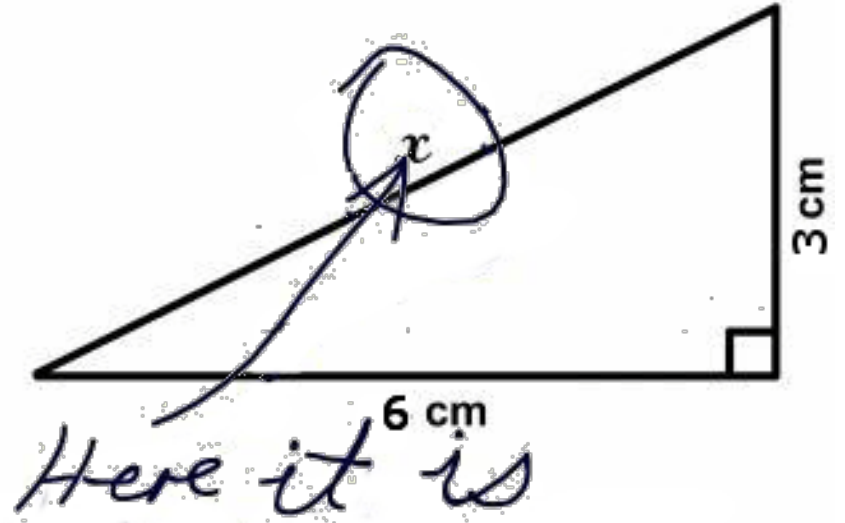
# Simple Math Questions

$$1 + 1 = ?$$

$$1 + 1 = 2 \quad - \text{ decimal}$$

$$1 + 1 = 10 \quad - \text{ binary}$$

3. Find x.



“There are only 10 types of people in this world, those who understand binary and those who don't.”

# How do Computers Perceive Numbers?

## Three Main Items to Consider

- Stored Integer – Actual bits stored in a computer register
- Number System – System used to ‘interpret’ the stored integers
- Perceived Value – Value determined by applying the number system to the stored integer

| Stored Integer | Number System | Perceived Value |
|----------------|---------------|-----------------|
| 0111           | Unsigned      | 7               |
| 0111           | Signed        | 7               |
| 1111           | Unsigned      | 15              |
| 1111           | Signed        | -1              |

Actual low level circuitry is not aware of number systems or perceived values



# How do Computers Perceive Fractional Numbers?

| Stored Integer | Number System | Perceived Value |
|----------------|---------------|-----------------|
| 011            | Unsigned      | 3               |
| 011            | Signed        | 3               |
| ?              | ?             | 1.5             |
| ?              | ?             | -1.5            |

Q Formatting:

[q.m.n]

Sign Bit

Integer Bit

Fraction Bit

# Q Format

$$0111 = 7 \quad [s.3.0]$$

$$(1 * 2^0) + (1 * 2^1) + (1 * 2^2) = 7$$

$$011.1 = 3.5 \quad [s.2.1]$$

$$(1 * 2^{-1}) + (1 * 2^0) + (1 * 2^1) = 3.5$$



# Q Format Conversion Techniques

## Stored Integer to Perceived Value

- Determine decimal value
- Divide by  $2^n$ 
  - q.m.n
- $1011 [s.1.2] \Rightarrow -5 \Rightarrow -1.25$

## Perceived Value to Stored Integer

- Multiply by  $2^n$ 
  - q.m.n
- Convert to binary
- $-1.25 \Rightarrow -5 \Rightarrow 1011$



# Q Format Examples

| Stored Integer | Number System | Perceived Value |        |
|----------------|---------------|-----------------|--------|
| 0111           | u.4.0         | 7               | (7/1)  |
| 0111           | s.3.0         | 7               | (7/1)  |
| 1111           | s.0.3         | -.125           | (-1/8) |
| 1111           | u.1.3         | 1.875           | (15/8) |
| 0111           | s.0.3         | .875            | (7/8)  |
| 1000           | s.0.3         | -1              | (-8/8) |



# Q Format Examples

$$0101 [s.0.3] = .625$$

$$5/(2^3) = .625$$

$$1101 [s.2.1] = -1.5$$

$$-3/(2^1) = -1.5$$

$$.5 [s.1.2] = 0010$$

$$.5 * (2^2) = 2$$

$$.5 [s.0.3] = 0100$$

$$.5 * (2^3) = 4$$





# Q Format Ranges and Precision

| Format | Largest positive value | Least negative value | Precision        |
|--------|------------------------|----------------------|------------------|
| Q0.15  | 0.999969482421875      | -1                   | 0.00003051757813 |
| Q1.14  | 1.99993896484375       | -2                   | 0.00006103515625 |
| Q2.13  | 3.9998779296875        | -4                   | 0.00012207031250 |
| Q3.12  | 7.999755859375         | -8                   | 0.00024414062500 |
| Q4.11  | 15.99951171875         | -16                  | 0.00048828125000 |
| Q5.10  | 31.9990234375          | -32                  | 0.00097656250000 |
| Q6.9   | 63.998046875           | -64                  | 0.00195312500000 |
| Q7.8   | 127.99609375           | -128                 | 0.00390625000000 |
| Q8.7   | 255.9921875            | -256                 | 0.00781250000000 |
| Q9.6   | 511.984375             | -512                 | 0.01562500000000 |
| Q10.5  | 1023.96875             | -1,024               | 0.03125000000000 |
| Q11.4  | 2047.9375              | -2,048               | 0.06250000000000 |
| Q12.3  | 4095.875               | -4,096               | 0.12500000000000 |
| Q13.2  | 8191.75                | -8,192               | 0.25000000000000 |
| Q14.1  | 16383.5                | -16,384              | 0.50000000000000 |
| Q15.0  | 32,767                 | -32,768              | 1.00000000000000 |

# VHDL Objects [signal, variable, constant, ~~file~~]

## Signal

- Most common object
  - `signal a : std_logic;`
  - `a <= '1';`
- Used to interconnect internal components

## Variable

- Can be thought of as a symbolic memory location
- No timing information is associated with a variable
  - `variable a : integer;`
  - `a := 1;`

## Constant

- Holds a value that cannot be changed
- Capitalization is used
  - `constant BUS_WIDTH : integer := 32;`
  - `constant BUS_BYTES : integer := BUS_WIDTH/8`

## File

- Holds file data, not synthesizable

## Alias

- Not a data object but rather an alternate name for an existing object
  - `signal word : std_logic_vector (15 downto 0);`
  - `alias enable_bit: std_logic is word(7);`



# VHDL Data Types

## *Type*

- is a named set of values with common characteristics.

## *Scalar*

- is one-dimensional while *composite* is multidimensional

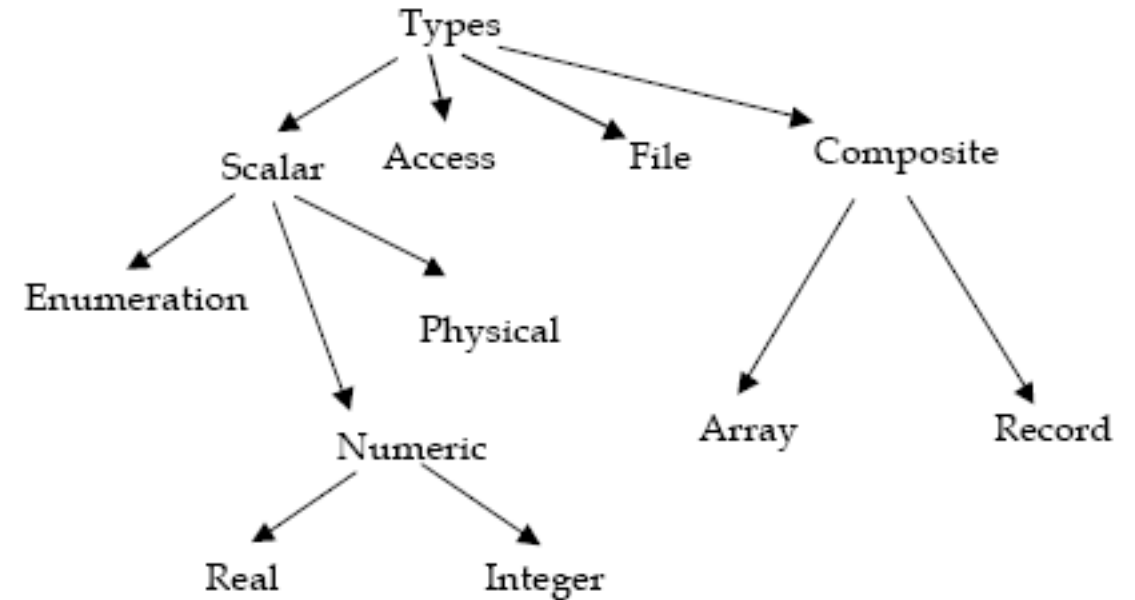
## *Physical*

- type is used for physical variables, i.e. variables that have units

## *Access and File*

- types facilitate access to external information through disk files

Not all data types are synthesizable!



VHDL is strongly typed

- Every signal, function parameter and function result has a type
- A few types are built in – user can define more
- In assignment statements, comparisons and function call types must match



# VHDL Libraries

## Standard Types

- boolean
- character
- string
- integer
- real
- time

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

## std\_logic\_1164

- std\_logic
- std\_logic\_vector

## numeric\_std

- unsigned
- signed

## Type Conversions

- Unsigned, Signed => Integer

```
Unsigned_int <= TO_INTEGER (A_uv) ;
Signed_int <= TO_INTEGER (B_sv) ;
```

- Integer => Unsigned, Signed

```
C_uv <= TO_UNSIGNED (Unsigned_int, 8) ;
D_sv <= TO_SIGNED (Signed_int, 8) ;
```

Array  
width = 8

- Numeric\_Std: Std\_Logic\_Vector => Integer

```
Unsigned_int <= to_integer(unsigned(A_slv));
Signed_int <= to_integer(signed(B_slv));
```

- Numeric\_Std: Integer => Std\_Logic\_Vector

```
C_slv <= std_logic_vector(to_unsigned(Unsigned_int, 8));
D_slv <= std_logic_vector(to_signed(Signed_int, 8));
```

# Synthesizable Data Types

## Integer

- Default width is 32 bits unless range is specified
  - `x : integer range 0 to 100;`
- New integer types can be defined
  - Ex: Type `day_of_month` is RANGE 0 TO 31;
- Operations:
  - `+`
  - `-`
  - `*`
  - `/`
  - `Mod`
  - `Rem`
  - `Abs`
  - `**`

## Boolean

- Can take value of true or false
- When a relational operators, `=`, `/=`, `<`, `<=`, `>` or `>=` are applied to two operands of the same type the result is a boolean value
- Operations:
  - `And`
  - `Or`
  - `Nand`
  - `Nor`
  - `Xor`
  - `Xnor`
  - `not`



# Synthesizable Data Types

## Bit

- Values are 0 and 1
- TYPE bit IS ('0','1');
- Used to model hardware logic levels
  - As opposed to boolean which models abstract conditions
- Same operations as Boolean
- Problem with bit is that it doesn't take into account the electrical properties of real signals

## Std\_logic & std\_logic\_vector

- Includes real world electrical characteristics
- All logical operations

```
TYPE std_logic IS ('U', --uninitialized
 'X', --forcing unknown
 '0', --forcing zero
 '1', --forcing one
 'Z', --high impedance
 'W', --weak unknown
 'L', --weak zero
 'H', --weak one
 '-',); --don't care
```



# Synthesizable Data Types

## Enumerated Type

- a type whose values are defined by listing (enumerating) them explicitly
- `TYPE type_name IS (name1, name2, ...);`
- Each type element must be an identifier or a character literal
- A signal of type 'enumerated\_type' can only ever have a value from the list

Ex: `TYPE state_type IS (IDLE, S0, S1, S2, S3)`

Ex: `TYPE hex_digit IS ('0','1','2','3','4','5','6','7','8','9','A','B','C','D','E','F')`

- Enumerations are not as “safe” as using constants for state machine encoding.

## Signed and unsigned

- Defined in `ieee.numeric_std` library

| Function | Argument 1 | Argument 2 | Returns |
|----------|------------|------------|---------|
| +        | signed     | signed     | signed  |
|          | signed     | integer    | signed  |
| -        | signed     | signed     | signed  |
|          | signed     | integer    | signed  |

| Function | Argument 1 | Argument 2 | Returns  |
|----------|------------|------------|----------|
| +        | unsigned   | unsigned   | unsigned |
|          | unsigned   | natural    | unsigned |
| -        | unsigned   | unsigned   | unsigned |
|          | unsigned   | natural    | unsigned |

# Non-Synthesizable Data Types

## Physical Types

- Represent physical quantities such as length, mass, time and current
- Definition contains range and units
- Time is predefined

```
Ex: TYPE resistance IS RANGE 0 TO 1E9
 UNITS
 ohm
 END UNITS resistance;
```

```
TYPE time IS RANGE
 UNITS
 fs;
 ps = 1000 fs;
 ns = 1000 ps;
 us = 1000 ns;
 ms = 1000 us;
 sec = 1000 ms;
 min = 60 sec;
 hr = 60 min;
 END UNITS;
```

## Floating Point

- Represents real numbers
- Predefined type but new floating point types can be defined

floating\_type\_definition <= RANGE simple\_expression(TO or DOWNTO) simple\_expression

Ex: Type input\_level is RANGE -10.0 TO +10.0;

- Bounds must evaluate to floating point numbers
- Operations
  - +
  - -
  - \*
  - /
  - Abs
  - \*\*





# Variables

## Two types

- Shared variables – can be accessed from multiple processes. Not synthesizable.
- Normal variables – can only be accessed from a single process or subprogram. Synthesizable

Used to store intermediate values in a process or subprogram

## Declared like a signal

- Variable count : std\_logic\_vector(3 downto 0);

## Assignments made with :=

- count := "0000";

## Variables vs. Signals

- 1] Variables can only be used in processes
- 2] Variables take value assigned to them immediately

Declared in the process before the BEGIN statement

Take the value assigned to them immediately

- Like a traditional programming language
- Remember signals don't get assigned until the end of the process

Order matters because they are updated immediately

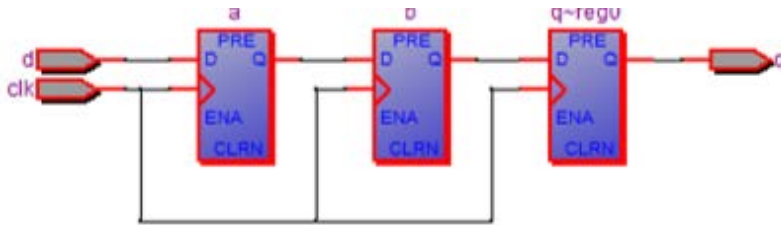
```
process
 variable x,y : integer := 0;
begin
 x := 1;
 y := x;
 wait;
end process;
```

# Variables Vs. Signals

```
entity regs is
 port(d, clk : in std_logic;
 q : out std_logic);
end regs;
```

```
architecture behave of regs is
 signal a,b : std_logic;
```

```
Begin
 process(clk)
 begin
 if (clk'event and clk = '1') then
 a <= d;
 b <= a;
 q <= b;
 end if;
 end process;
end behave;
```

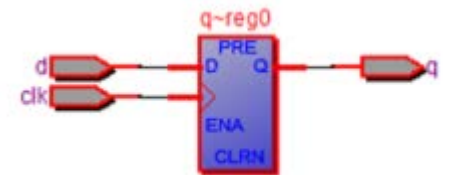


How many registers?

```
entity regv is
 Port (clk,d : in std_logic;
 q : out std_logic);
end regv;
```

```
architecture behave of regv is
 begin
```

```
 process(clk)
 variable a,b : std_logic;
 begin
 if (clk'event and clk = '1') then
 a := d;
 b := a;
 q <= b;
 end if;
 end process;
end behave;
```



How many registers?



# Arrays

---

- An array is an indexed collection of values all of the same type
- Represented as a new data type
- Can be single or multi-dimensional
- Constrained
  - bounds for index are established when the type is defined
- Unconstrained
  - bounds are established after the type is defined
- Each position in the array has a scalar index value

Why did the programmer  
quit his job?

Because he didn't get  
arrays



# Constrained and Unconstrained Array Types

## Unconstrained

### ■ Define Type

```
type std_logic_vector is array (natural range <>) of std_logic;
```

- <> is called box and is used as a place holder for index range
- Box is filled later when type is used

### ■ Declaring a Signal

```
signal bytebus : std_logic_vector(7 downto 0);
```

### ■ Predefined Unconstrained Types

- Type String is array (positive <>) of character;
  - Starts at index 1
- Type bit\_vector is array (natural <>) of bit;
  - Starts at index 0
- Type Std\_logic\_vector is array (natural <>) std\_logic
  - Starts at index 0
- Range is set when signal is declared

## Constrained

### ■ Define Type

```
type my_byte is array (7 downto 0) of std_logic;
```

### ■ Declaring Signal

```
signal type_bus : my_byte;
```



# Multidimensional Arrays and Concatenation

- **Ex: 1**

Type Large\_word is array (63 downto 0) of bit;

Type Array\_list is array (0 to 7) of large\_word;

- **Ex : 2**

Type t2D\_FFT is array (1 to 128, 1 to 128) of real;

Type list\_of\_large is array (0 to 7, 63 downto 0) of bit;

- **Ex: 3**

Type RAM\_ARRAY is array(natural <>) of std\_logic\_vector(7 downto 0);

Signal MY\_RAM : RAM\_ARRAY (1023 downto 0);

MY\_RAM(255) <= "11110000";

MY\_RAM(255)(0) <= '0';

- **Example**

Signal BYTE : bit\_vector (7 downto 0);

Signal A\_BUS, B\_BUS : bit\_vector( 3 downto 0);

BYTE <= A\_BUS & B\_BUS;

--BYTE(7) ← A\_BUS(3)

--BYTE(6) ← A\_BUS(2)

--BYTE(5) ← A\_BUS(1)

--BYTE(4) ← A\_BUS(0)

--BYTE(3) ← B\_BUS(3)

--etc...

- **Can also be done with single elements**

Signal Z\_BUS : bit\_vector(3 downto 0);

Signal a, b, c, d : bit

Z\_Bus <= a & b & c & d

--Z\_Bus(3) ← a, Z\_Bus(2) ← b, Z\_Bus(1) ← c, Z\_Bus(0) ← d



# Reference and Assignment

- Arrays can be equated rather than having to transfer element by element
- Refer to individual elements by:
  - Single index value - `A(5),A(0)`
  - Range – must be contiguous sequence in one-dimensional array – `A(15 downto 0)`

- Elements are assigned according to position, not their number
- Example

```
Signal Z_BUS : bit_vector (3 downto 0);
Signal C_BUS : bit_vector (0 to 3);
Z_BUS <= C_BUS;
```

```
--Z_BUS(3) ← C_BUS(0)
--Z_BUS(2) ← C_BUS(1)
--Z_BUS(1) ← C_BUS(2)
--Z_BUS(0) ← C_BUS(3)
```

- Be consistent to avoid issues



# Aggregates

Purpose is to bundle signals together

May be used on both sides of an assignment

A list of element values enclosed in parentheses

Used to initialize elements of an array to literal values

Keyword 'others' selects all remaining elements

A set of values can be set to a single value by forming a list of elements separated by vertical bars (pipe) |

Type 2D\_FFT is array (1 to 128, 1 to 128) of real;

Variable X\_Ray\_FFT : 2D\_FFT ((60,68) | (62,67) | (67,73) | (60,60) => 1.0, others => 0.0);

Architecture EXAMPLE of AGGREGATES is

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

```
signal Z_BUS : bit_vector(3 downto 0);
```

```
signal A_BIT, B_BIT, C_BIT, D_BIT : bit
```

Begin

--the following are positional references

```
Z_BUS <= (A_BIT, B_BIT, C_BIT, D_BIT);
```

```
(A_BIT, B_BIT, C_BIT, D_BIT) <= bit_vector("1011");
```

```
(A_BIT, B_BIT, C_BIT, D_BIT) <= BYTE(3 downto 0);
```

--the following is named association reference. Order doesn't matter

```
BYTE <= (7 => '1', 5 downto 1 => '1', 6 => B_BIT, others => '0');
```

End EXAMPLE;



# Operations and Attributes

## Operations

- One dimensional arrays of bit or boolean
  - Element by element AND, OR, NAND, NOR, XOR, XNOR

Type `large_word` is array (63 downto 0) of bit;

Variable `samp_1`, `samp_2` : `large_word` (0 to 63 => '0');

Constant `bit_mask` : `large_word` (0 to 15 => '1');

`Samp_2 := samp_1 AND bit_mask;`

- Complement of elements of a single array using NOT
  - `Samp_2 <= NOT(samp_1);`

## Attributes

- Signal `buss` : `std_logic_vector(7 downto 0);`
  - `Bus'left =`
  - `Bus'right =`
  - `Bus'high =`
  - `Bus'low =`
  - `Bus'range =`
  - `Bus'reverse_range =`
  - `Bus'length =`
  - `Bus'ascending =`





# Memory

## RAM – Random Access Memory

- Used for storing variables for programs

## ROM - Read Only Memory

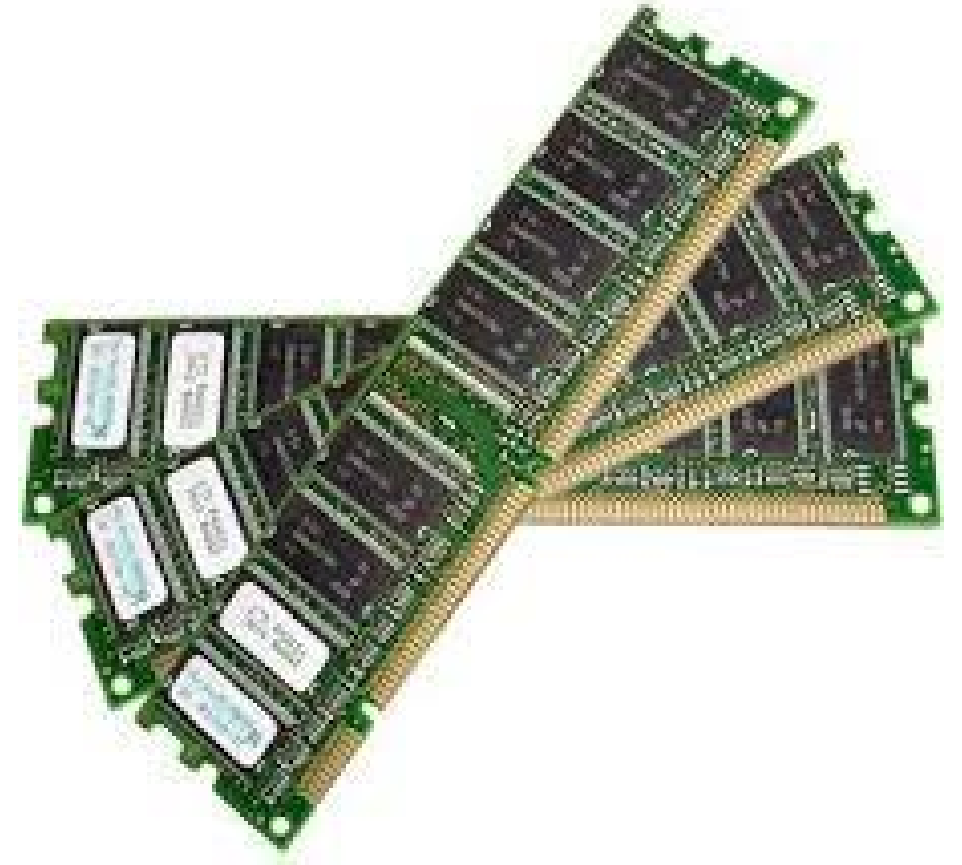
- Typically used for booting

## ROM and RAM both allow for random access

- RAM  $\Leftrightarrow$  read/write RAM
- ROM  $\Leftrightarrow$  read only RAM

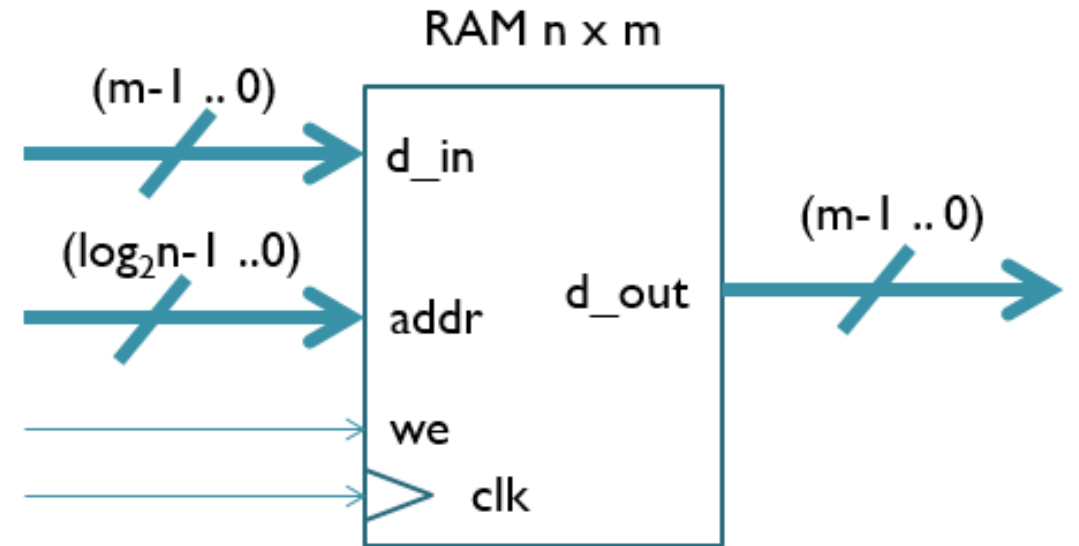
## How many gigs of RAM can a 32 bit OS use?

- 4 GB =  $2^{32}$  = 4294967296 bytes



# RAM Configuration

- An  $n \times m$  RAM has  $n$  locations each of size  $m$ 
  - Example 4K x 128 :
    - 4096 locations (1K = 1024)
    - The data in each location is 128 bits long
  - $m$  is the size of the  $d\_in$  bus and the  $d\_out$  bus
  - $n$  and  $m$  have no relationship
  - The size of the address bus is  $\log_2 n$ . Each location has a unique address



# VHDL for Simple RAM

```
entity raminfer is
 generic (addr_width : integer := 6; --default to a 64 x 256 RAM
 data_width : integer := 8);
 port(
 clk : in std_logic;
 we : in std_logic;
 addr : in std_logic_vector((addr_width - 1) downto 0);
 d_in : in std_logic_vector(data_width - 1 downto 0);
 d_out : out std_logic_vector(data_width - 1 downto 0));
end raminfer;

architecture rtl of raminfer is


 type ram_type is array ((2**addr_width - 1) downto 0) of std_logic_vector (data_width - 1 downto 0);
 signal RAM : ram_type;

begin
 process(clk)
 begin
 if (clk'event and clk = '1') then
 if (we = '1') then
 RAM(to_integer(unsigned(addr))) <= d_in;
 end if;
 d_out <= RAM(to_integer(unsigned(addr)));
 end if;
 end process;
end rtl;
```



# RAM Synthesis

| Flow Summary                       |                                                  |
|------------------------------------|--------------------------------------------------|
| Flow Status                        | Successful - Sun Oct 26 14:20:11 2014            |
| Quartus II 32-bit Version          | 13.0.1 Build 232 06/12/2013 SP 1 SJ Full Version |
| Revision Name                      | raminfer                                         |
| Top-level Entity Name              | raminfer                                         |
| Family                             | Cyclone II                                       |
| Device                             | EP2C35F672C6                                     |
| Timing Models                      | Final                                            |
| Total logic elements               | 0 / 33,216 ( 0 % )                               |
| Total combinational functions      | 0 / 33,216 ( 0 % )                               |
| Dedicated logic registers          | 0 / 33,216 ( 0 % )                               |
| Total registers                    | 0                                                |
| Total pins                         | 82 / 475 ( 17 % )                                |
| Total virtual pins                 | 0                                                |
| Total memory bits                  | 512 / 483,840 ( < 1 % )                          |
| Embedded Multiplier 9-bit elements | 0 / 70 ( 0 % )                                   |
| Total PLLs                         | 0 / 4 ( 0 % )                                    |


$$64 \times 8 = 512$$

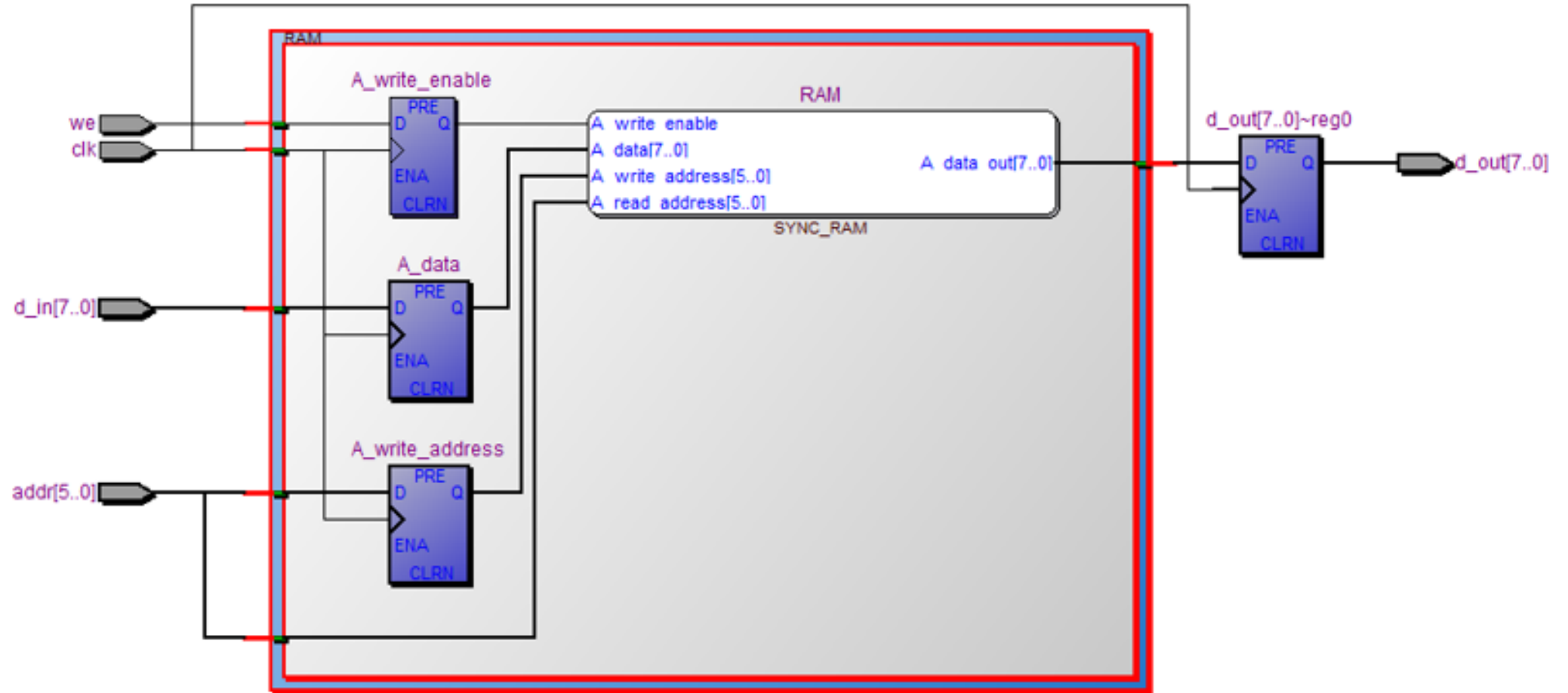
## Distributed RAM

- FPGA logic blocks have configurable lookup tables
- Each LUT can be used as a couple bits of RAM
- Many LUTs can be chained together to create a larger RAM block

## Block RAM

- Dedicated memory containing several kilobytes of RAM

# RAM RTL View



# ROM

```
-- ENTITY DECLARATION
--
ENTITY hex_to_ssd IS
PORT(inputs : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
 clk : IN STD_LOGIC;
 outputs : OUT STD_LOGIC_VECTOR(6 DOWNTO 0));
END hex_to_ssd;
--
-- STRUCTURE ARCHITECTURE BODY
--
ARCHITECTURE structure OF hex_to_ssd IS

 TYPE SSD_array IS ARRAY(0 to 9) OF STD_LOGIC_VECTOR(6 DOWNTO 0);

 SIGNAL SSD: SSD_array := ("1000000", "1111001", "0100100", "0110000",
 "0011001", "0010010", "0000010", "1111000",
 "0000000", "0010000");

begin
process (clk)
begin
 if (rising_edge(clk)) then
 outputs <= SSD (to_integer(unsigned(inputs)));
 end if;
end process;
END structure;
```

How does ROM differ from RAM?

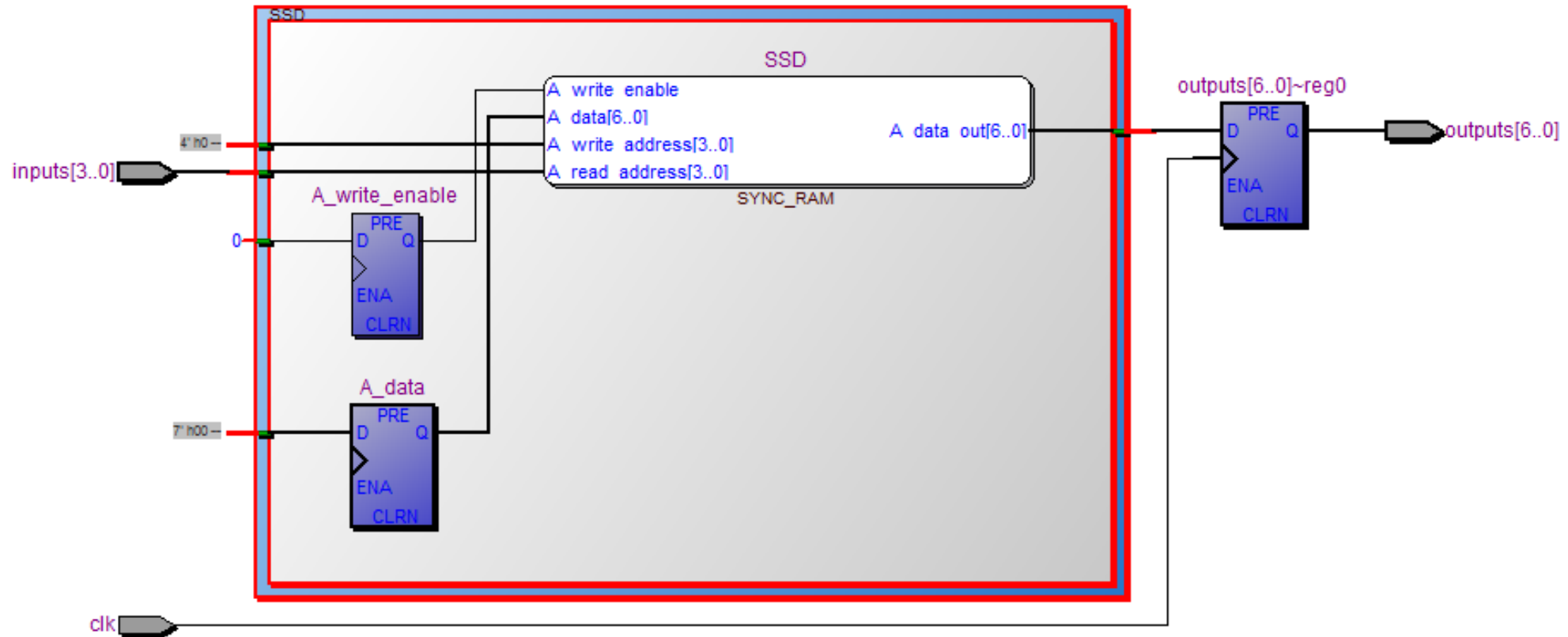
The configuration is the same (n x m)

ROM is inferred when VHDL is written as a  
look up table

case  
array



# ROM RTL View



# Loops

Contains a sequence of sequential statements that can be repeatedly executed, zero or more times

Execution continues until terminated by:

- Completion of the iteration scheme
- Execution of an exit statement
- Execution of a next statement that specifies a label outside of the loop
- Execution of a return statement (functions and procedures only)

## Iteration Scheme

- Optional
- Loop without it is infinite – cannot be synthesized
- Two iteration schemes
  - For loops - synthesizable
  - While loops
    - Not synthesizable
    - Useful in testbenches





# For Loop

```
[loop_label:] for identifier in range loop
 sequence_of_statements
end loop [loop_label];
```

```
for i in 0 to 3 loop
 mode <= std_logic_vector(to_unsigned(i,2));
 for j in 0 to 15 loop
 test_vector <= std_logic_vector(to_unsigned(j,4));
 wait for 100 ns;
 -- insert assert statement here
 end loop;
end loop;
```

Identifier or loop parameter follows **for**

- Implicitly declared
- Type is base type of discrete range
  - Integer
  - enumerated
- Only exists while loop is being executed
- Not visible outside of the loop
- Range must be in one of the following forms:
  - *integer\_expression to integer\_expression*
  - *integer\_expression **downto** integer\_expression*
- Treated as a constant in the loop
  - Can be read
  - Cannot be written
- Loop is executed once for each value in the range



# While Loop

```
[loop_label:] while condition loop
 sequence_of_statements
end loop [loop_label];
```

```
process
begin
 while error_flag /= '1' and done /= '1' loop
 Clock <= not Clock;
 wait for CLK_PERIOD/2;
 end loop;
end process;
```

Condition is evaluated before each iteration of the loop – including first iteration

- If true, loop executes
- If false, loop terminates

Must ensure that loop can terminate

- Statements inside the loop will eventually cause the loop's condition to evaluate to false
- Cause an exit statement
- Cause a return statement
- Otherwise, infinite loop



# Next and Exit Statements

## Next Statement

- Used to terminate current iteration and go to the next iteration
- *Next;*
  - Starts the next iteration of the immediately enclosing loop
- *Next loop\_label;*
  - Used in nested loops to indicate for which loop to complete the iteration
- *Next when condition;*
  - Only jump to next iteration if condition is true
- *Next loop\_label when condition;*
  - Combines previous 2 statements

## Exit Statement

- Used to terminate the execution of an enclosing loop statement
- When executed, any remaining statements in the loop are skipped
- Control is transferred to the statement after the end loop keywords
- Can contain a condition

If condition then  
Exit  
End if;

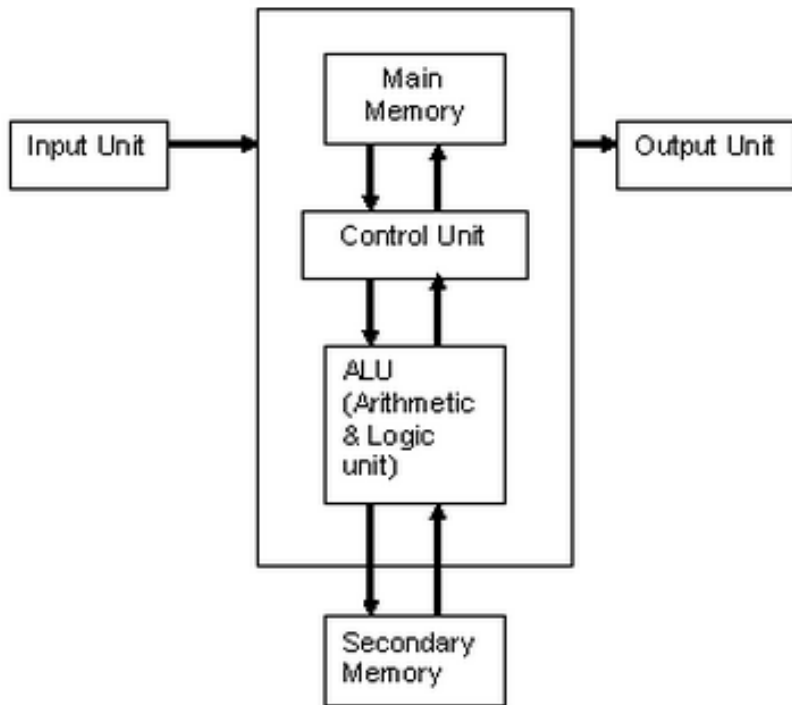
same as:

exit when condition

# Processing Architectures

## Microprocessor

- Single chip that contains just the processor
- Have a bus to external memory
- Typically used for desktop applications

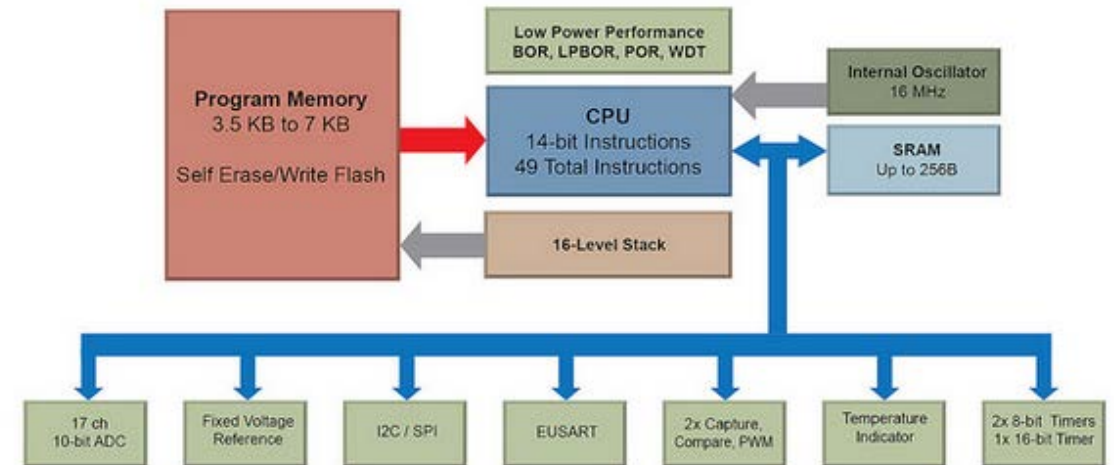


## Microcontroller

- Have a CPU, RAM, ROM, other peripherals
- Typically slower clock rate than microprocessor
- Typically used for embedded applications



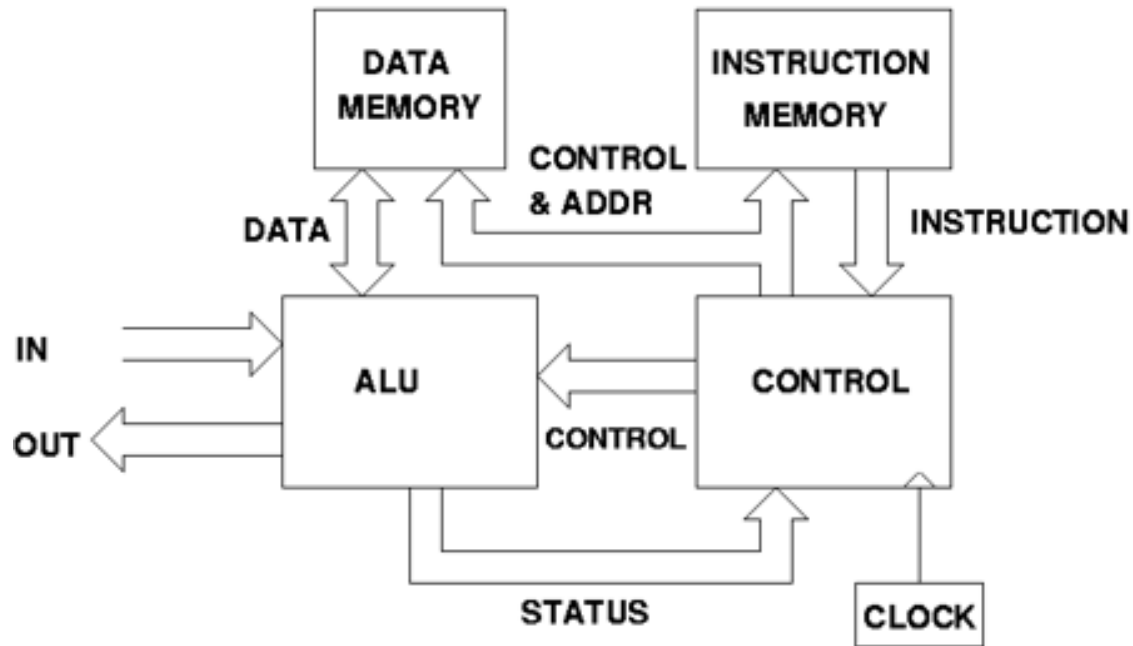
PIC16F1512, PIC16F1513



# Processing Architectures

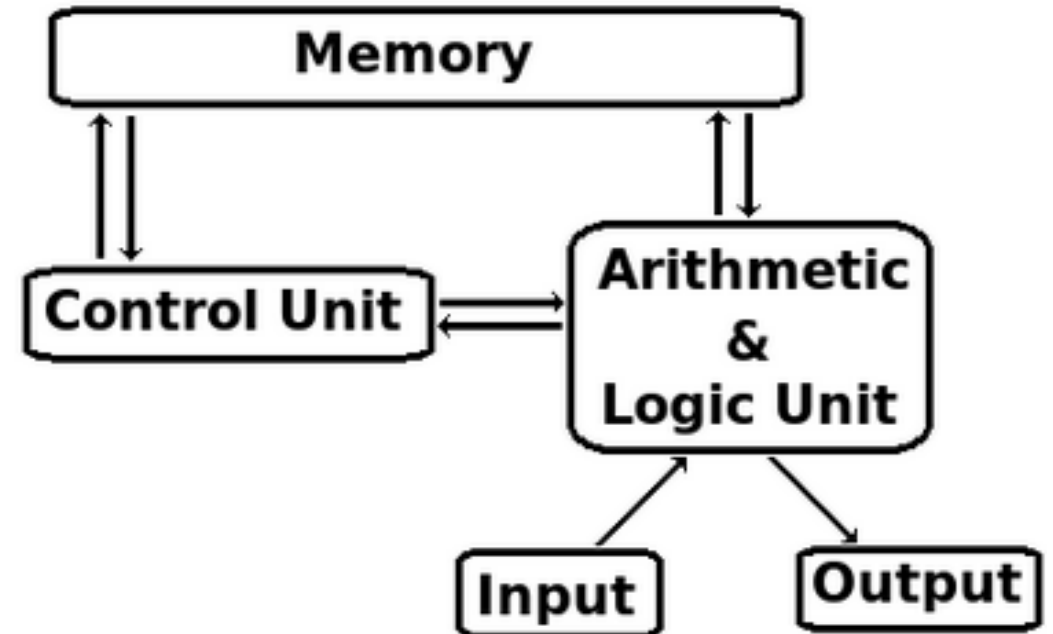
## Harvard Architecture

- Separate instruction and data memories
- Can access program and data memory simultaneously
- Program memory [ROM], data memory [RAM]



## Von-Neuman Architecture

- Single memory used for instructions and data
- Program can easily modify itself. Stored in RAM
- Can trade between larger programs or more data



# Processor Instructions

- A processor executes a program
  - A sequence of instructions, each performing a small step of a computation
- Instruction set: the repertoire of available instructions
  - Different processor types have different instruction sets
- High-level languages: more abstract
  - E.g., C, C++, Ada, Java
  - Translated to processor instructions by a compiler

- Instructions are encoded in binary
  - Stored in the instruction memory
- A processor executes a program by repeatedly
  - Fetching the next instruction
  - Decoding it to work out what to do
  - Executing the operation
- Program counter (PC)
  - Register in the processor holding the address of the next instruction



# Gamnut Core

General-Purpose Registers

|    |   |
|----|---|
| r0 | 0 |
| r1 |   |
| r2 |   |
| r3 |   |
| r4 |   |
| r5 |   |
| r6 |   |
| r7 |   |

Data Memory  
(256 × 8-bit, 8-bit addresses)

|     |  |
|-----|--|
| 0   |  |
| 1   |  |
| 2   |  |
| ... |  |
| 254 |  |
| 255 |  |

Condition Code Registers

|   |  |       |
|---|--|-------|
| C |  | Carry |
| Z |  | Zero  |

Program Counter

|    |  |
|----|--|
| PC |  |
|----|--|

Instruction Memory  
(4K × 18-bit, 12-bit addresses)

|      |  |
|------|--|
| 0    |  |
| 1    |  |
| 2    |  |
| ...  |  |
| 4094 |  |
| 4095 |  |

Arith/Logical Register

|   |   |   |   |           |  |           |  |            |  |   |  |   |           |
|---|---|---|---|-----------|--|-----------|--|------------|--|---|--|---|-----------|
| 4 |   |   |   | 3         |  | 3         |  | 3          |  | 2 |  | 3 |           |
| 1 | 1 | 1 | 0 | <i>rd</i> |  | <i>rs</i> |  | <i>rs2</i> |  |   |  |   | <i>fn</i> |

Arith/Logical Immediate

|   |           |           |           |              |
|---|-----------|-----------|-----------|--------------|
| 1 | 3         | 3         | 3         | 8            |
| 0 | <i>fn</i> | <i>rd</i> | <i>rs</i> | <i>immed</i> |

Shift

|   |   |   |   |           |           |              |  |           |
|---|---|---|---|-----------|-----------|--------------|--|-----------|
| 3 | 1 | 3 | 3 | 3         | 3         | 2            |  |           |
| 1 | 1 | 0 |   | <i>rd</i> | <i>rs</i> | <i>count</i> |  | <i>fn</i> |

Memory, I/O

|   |   |           |           |           |               |
|---|---|-----------|-----------|-----------|---------------|
| 2 | 2 | 3         | 3         | 8         |               |
| 1 | 0 | <i>fn</i> | <i>rd</i> | <i>rs</i> | <i>offset</i> |

Branch

|   |   |   |   |   |   |           |  |             |  |   |  |  |  |
|---|---|---|---|---|---|-----------|--|-------------|--|---|--|--|--|
| 6 |   |   |   |   |   | 2         |  | 2           |  | 8 |  |  |  |
| 1 | 1 | 1 | 1 | 1 | 0 | <i>fn</i> |  | <i>disp</i> |  |   |  |  |  |

Jump

|           |           |             |
|-----------|-----------|-------------|
| 5         | 1         | 12          |
| 1 1 1 1 0 | <i>fn</i> | <i>addr</i> |

Miscellaneous

|               |           |   |
|---------------|-----------|---|
| 7             | 3         | 8 |
| 1 1 1 1 1 1 0 | <i>fn</i> |   |

■ Encoding for addc r3, r5, 24

■ Arithmetic immediate, fn = 001

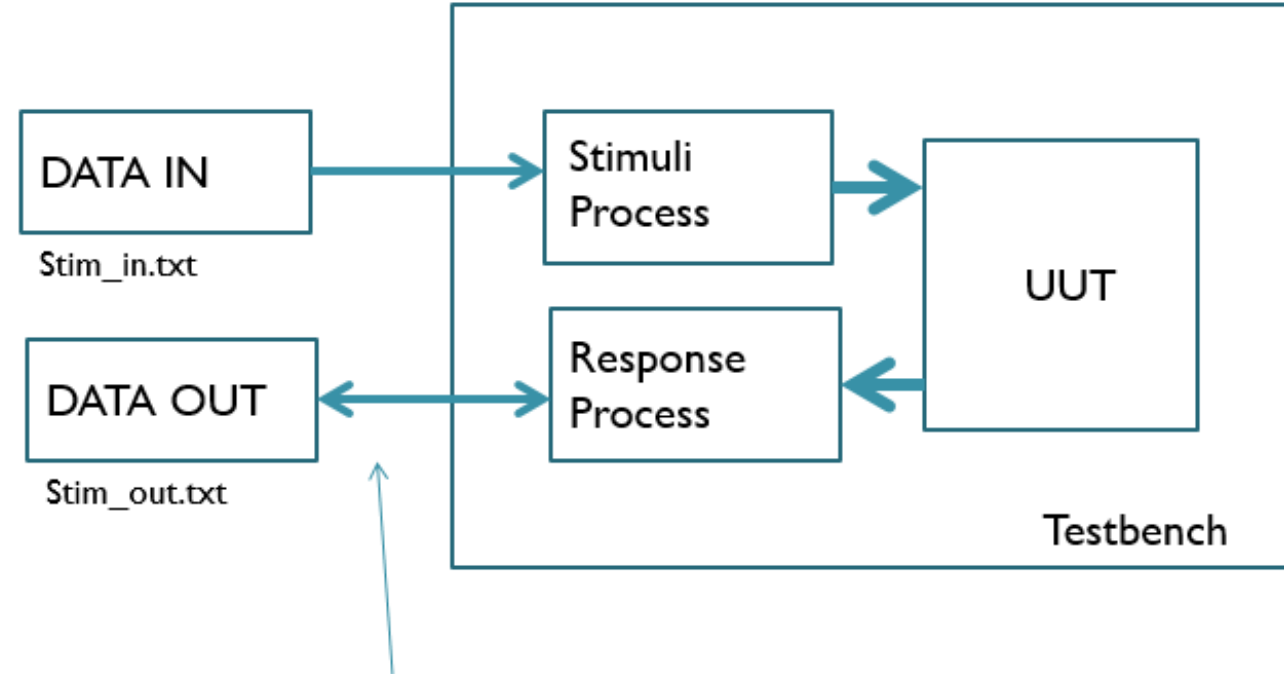
|   |           |           |           |                 |
|---|-----------|-----------|-----------|-----------------|
| 1 | 3         | 3         | 3         | 8               |
| 0 | <i>fn</i> | <i>rd</i> | <i>rs</i> | <i>immed</i>    |
| 0 | 0 0 1     | 0 1 1     | 1 0 1     | 0 0 0 1 1 0 0 0 |

■ 05D18

# File IO

## File IO simplifies testing of large circuits

- Offline input vector generation
- Input vectors can be modified without changing TB code
- Can run a large number of tests in a regression suite
- Output files give mechanism for inspecting output vectors of a design



Data\_out can be used 2 ways. 1) write the outputs To the file and compare afterwards  
2) Read the outputs and compare using assert in test Bench.



# Text IO Packages

## std.textio.all

- Functions

- readline(...)
- read(...)
- writeline(...)
- write(...)
- endfile(...)

- Data Types

- most types but not std\_logic or std\_logic\_vector

## ieee.std\_logic.textio.all

- Same functions
- Adds support for std\_logic types

## Standard VHDL Packages

<http://www.csee.umbc.edu/portal/help/VHDL/stdpkg.html>

### std.textio.all read procedure

```
procedure READ (L: inout LINE; VALUE: out BIT; GOOD: out BOOLEAN);
procedure READ (L: inout LINE; VALUE: out BIT);

procedure READ (L: inout LINE; VALUE: out BIT_VECTOR; GOOD: out BOOLEAN);
procedure READ (L: inout LINE; VALUE: out BIT_VECTOR);

procedure READ (L: inout LINE; VALUE: out BOOLEAN; GOOD: out BOOLEAN);
procedure READ (L: inout LINE; VALUE: out BOOLEAN);

procedure READ (L: inout LINE; VALUE: out CHARACTER; GOOD: out BOOLEAN);
procedure READ (L: inout LINE; VALUE: out CHARACTER);

procedure READ (L: inout LINE; VALUE: out INTEGER; GOOD: out BOOLEAN);
procedure READ (L: inout LINE; VALUE: out INTEGER);

procedure READ (L: inout LINE; VALUE: out REAL; GOOD: out BOOLEAN);
procedure READ (L: inout LINE; VALUE: out REAL);

procedure READ (L: inout LINE; VALUE: out STRING; GOOD: out BOOLEAN);
procedure READ (L: inout LINE; VALUE: out STRING);

procedure READ (L: inout LINE; VALUE: out TIME; GOOD: out BOOLEAN);
procedure READ (L: inout LINE; VALUE: out TIME);
```



# File Read and Write

## Read

- variable line\_in : line;
- variable bits\_in : bit\_vector(3 downto 0);
- file\_open(fid,"input.txt",READ\_MODE);
- readline(fid,line\_in);
- read(line\_in,bits\_in);
- file\_close(fid);

## Write

- variable line\_out : line;
- variable bits\_out: bit\_vector(3 downto 0);
- file\_open(fid,"output.txt",WRITE\_MODE);
- write(line\_out, bits\_out, right, 4);
- writeline(fid, line\_out);
- file\_close(fid);



# FPGA Memory

## Distributed Memory

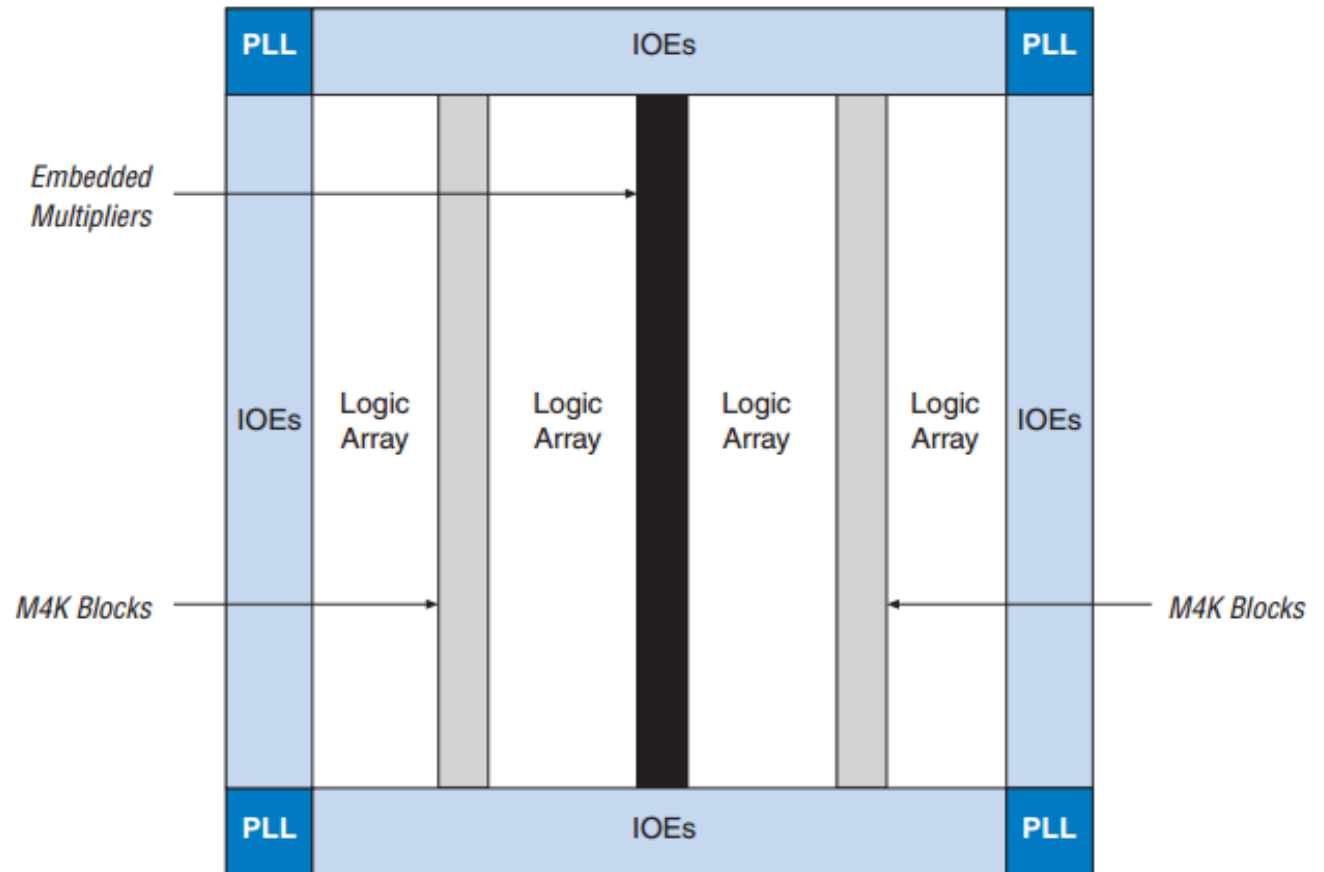
- Each LUT can be used for a couple bits of RAM
- Many LUTs can be chained together to create a larger RAM block

## Block Memory

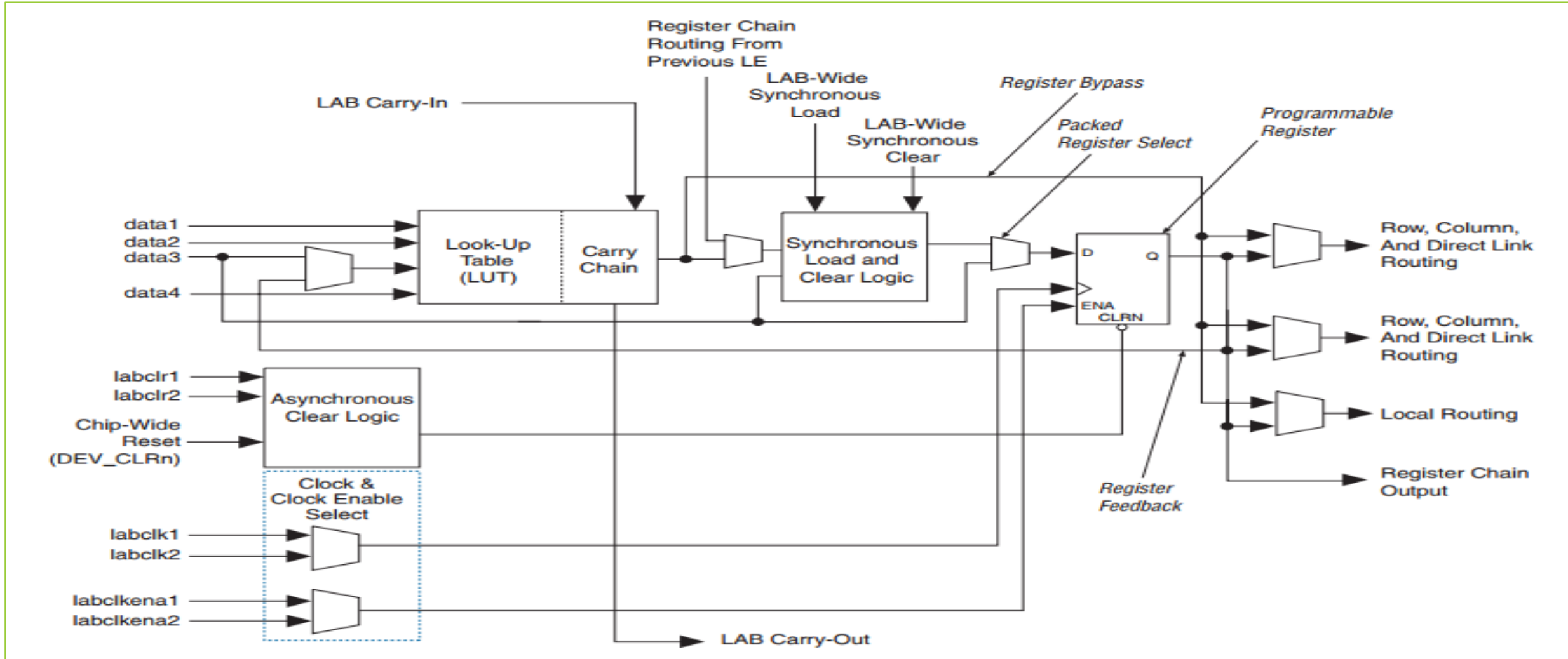
- Dedicated memory containing several kilobytes of RAM

## DE2 Board

- Cyclone II 2C35 FPGA
- 105 M4K Blocks [4 kbits] => ~52 kB



# Altera Logic Element [LE]



# Working with Altera Block Memory

## Synthesis

- Create Quartus II project
- Create new single port ROM IP block
  - Target M4K blocks
  - Single clock
  - Specify MIF file
  - Enable in-system memory editing
- Create ROM folder
  - /src/ROM
- Modify compile script
  - `set_global_assignment -name QIP_FILE ../../src/rom/rom.qip`
- Compile

## Simulation

- Compile Quartus II simulation libraries
  - altera\_mf folder is generated
- Copy Altera\_mf folder to working directory
  - /src/altera\_mf
- Modify sim.do file
  - `vmap altera_mf ../../src/altera_mf`
  - `vcom -93 -work work ../../src/rom/rom.vhd`
- Simulate



# Altera Memory Initialization File [MIF]

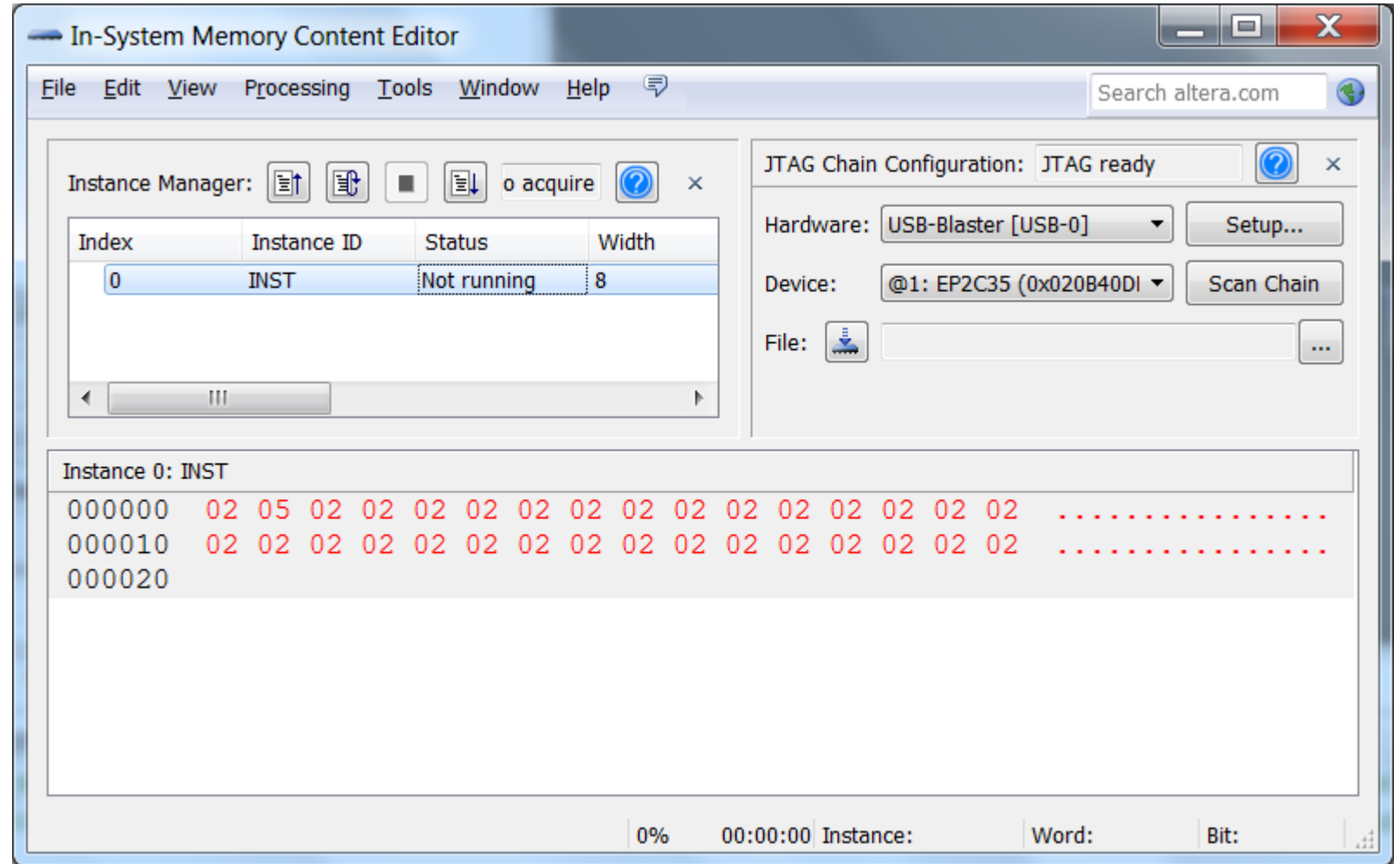
Same file is used for  
hardware memory  
initialization as well as  
for simulation runs

```
DEPTH = 16; % Memory depth and width are required. DEPTH is the number of addresses %
WIDTH = 8; % WIDTH is the number of bits of data per word %
% DEPTH and WIDTH should be entered as decimal numbers %
ADDRESS_RADIX = DEC; % Address and value radices are required %
DATA_RADIX = DEC; % Enter BIN, DEC, HEX, OCT, or UNS; unless %
% otherwise specified, radices = HEX %
-- Specify values for addresses, which can be single address or range
CONTENT
BEGIN
0 : 2;
1 : 5;
2 : 2;
3 : 2;
4 : 2;
5 : 2;
6 : 2;
7 : 2;
8 : 2;
9 : 2;
10 : 2;
11 : 2;
12 : 2;
13 : 2;
14 : 2;
15 : 2;
END;
```



# Altera In-System Memory Viewer

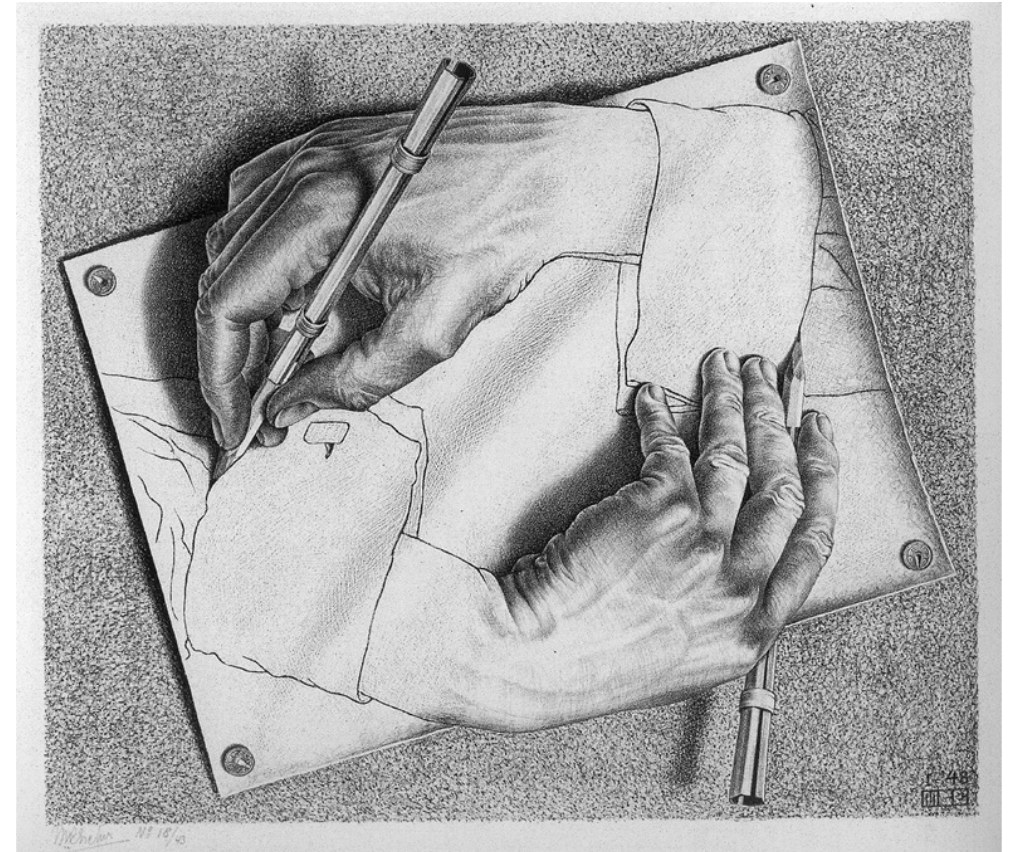
Can read and write  
memory while a program  
is running





# Generate Statements

- Can dynamically add code to a VHDL file before elaboration
- Same category as generic statements and are often used in conjunction with generic statements
- Use Cases
  - Debugging
    - Can add debug constructs such as file IO to a synthesizable UUT
  - Code Replication
    - Can use FOR GENERATE statement to create multiple copies of logic structures





# Generate for Debugging

```
debug_monitor : if debug generate
 debugger : process(clk,reset,data_address)
 use std.textio.all;
 variable debug_line : line;
 file output_file : text;
 begin
 file_open(output_file,"output.txt",WRITE_MODE);
 if clk'event and clk = '1' then
 write(debug_line,now,left,7,unit => ns);
 write(debug_line,string'("data_address"),right,20);
 write(debug_line,to_bitvector(data_address),right,20);
 writeline(output_file,debug_line);
 end if;
 end process debugger;
end generate debug_monitor;
```



# For Generate

- Used to replicate a subsystem
- Iterative instantiation of circuit parts
- A concurrent statement containing further concurrent statements that are to be replicated during elaboration of a design
- Header looks similar to a for loop
- It repeats the loop body of concurrent statements for a definite number of times
  - Range must be static (defined at compilation time)
  - Normally related to some width parameter

```
library ieee;
use ieee.std_logic_1164.all;
entity gen_xor is
 generic (
 wide : integer := 2
);
 port (
 in_vec : in std_logic_vector(wide - 1 downto 0);
 y : out std_logic
);
end gen_xor;
architecture beh of gen_xor is
 signal tmp : std_logic_vector(wide - 1 downto 0);
begin
 tmp(0) <= in_vec(0);
 xor_gen:
 for i in 1 to (wide - 1) generate
 tmp(i) <= in_vec(i) xor tmp(i - 1);
 end generate;
 y <= tmp(wide - 1);
end beh;
```

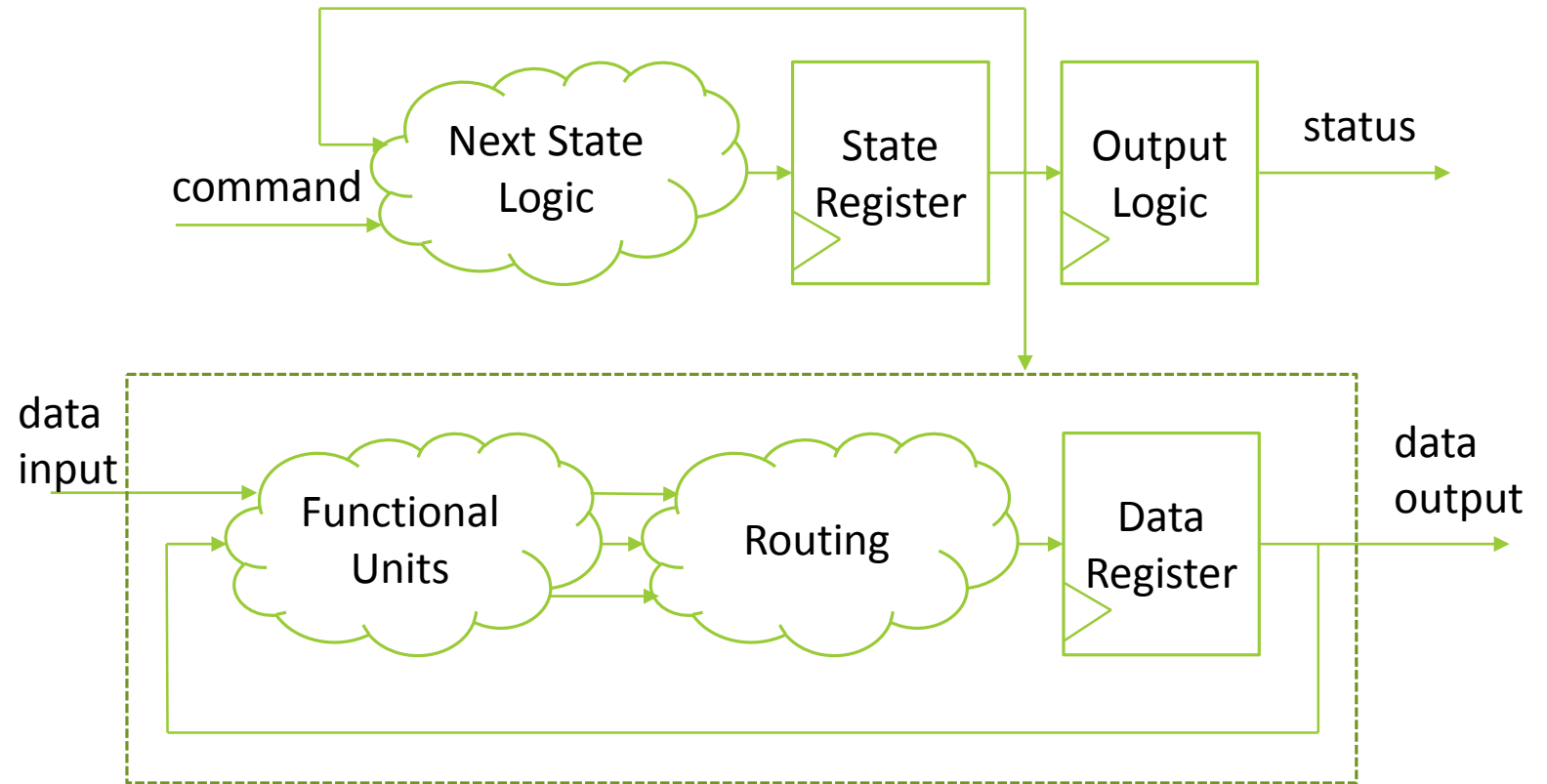
# FSMD – Finite State Machine with Datapath

## FSM

- Next State Logic
- State Register
- Output Logic

## Data Path

- Functional Units [FU]
  - Perform all mathematical computation
    - $x\_add \leq x + 1;$
    - $x\_sub \leq x - 1;$
- Routing
  - if add then
    - $x\_next \leq x\_add;$
  - else
    - $x\_next \leq x\_sub;$
  - end if
- Data Register
  - Store the intermediate computational results
  - $x \leq x\_next;$



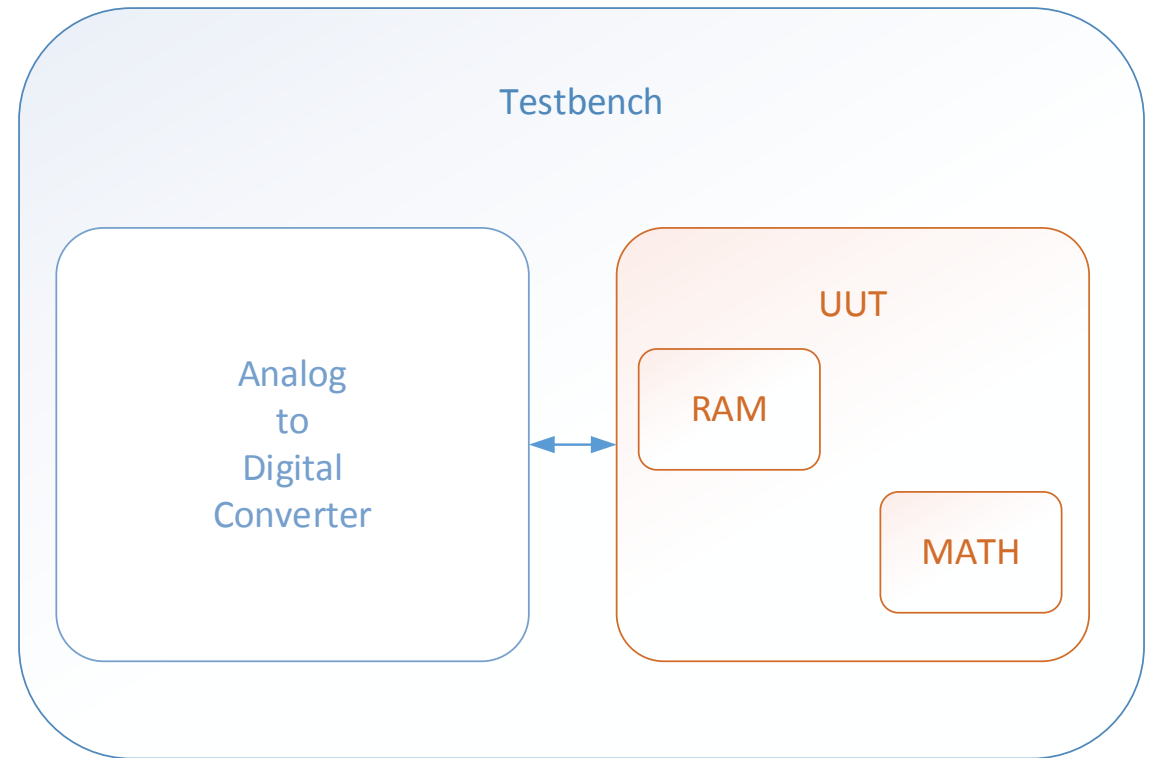
# Bus Functional Model

- A non-synthesizable model of an integrated circuit component having one or more external buses
- Used to verify firmware that talks to the external component
- Useful when actual hardware will not be in house for months yet firmware activities have already started.

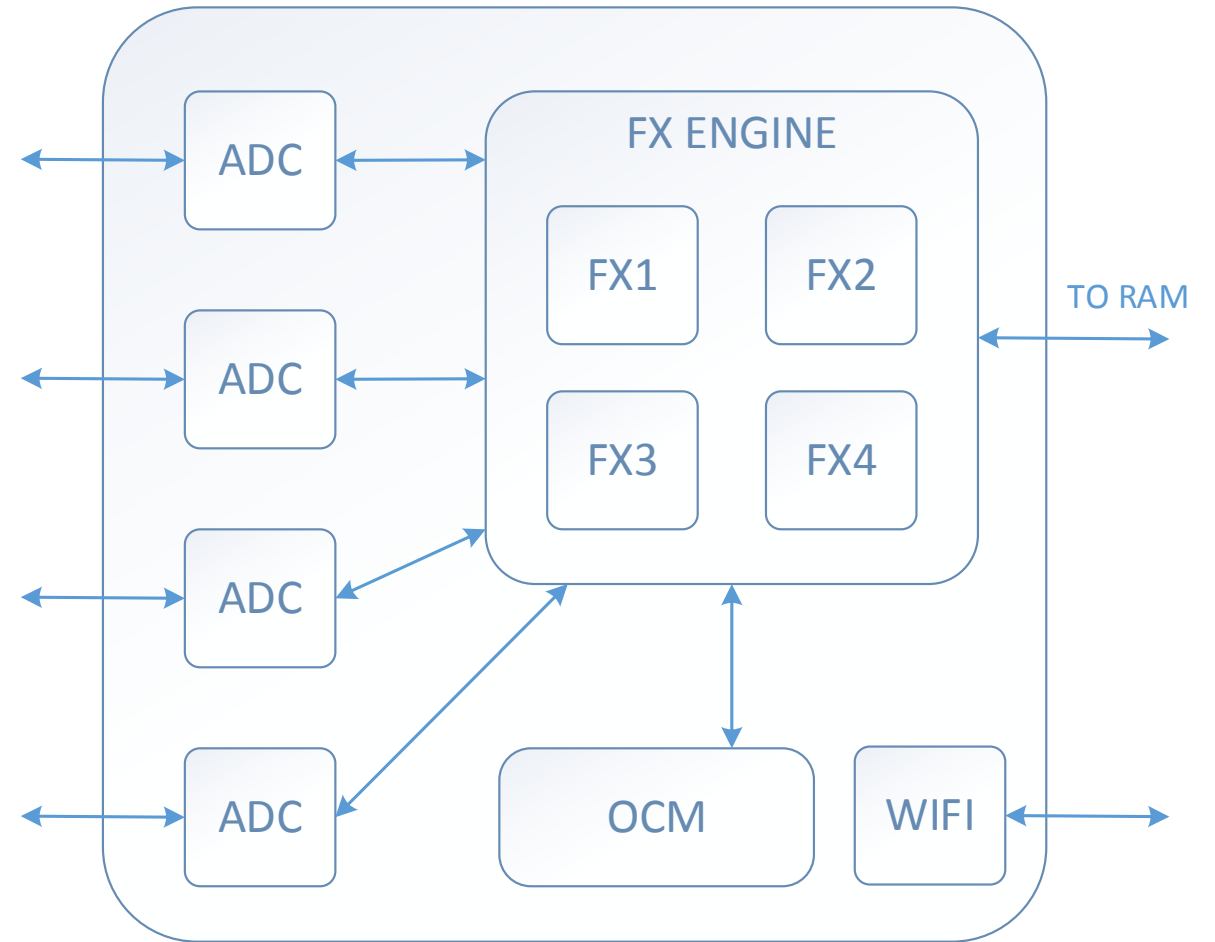
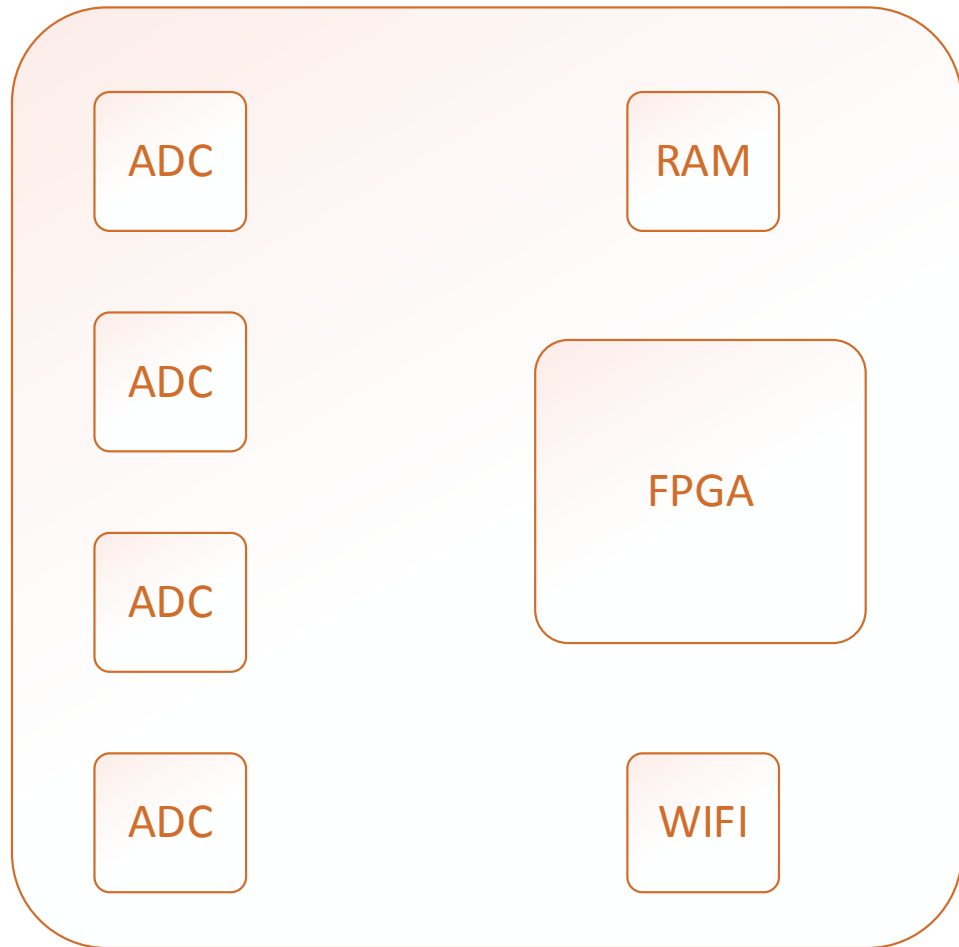


UUT is inside FPGA

ADC is outside FPGA

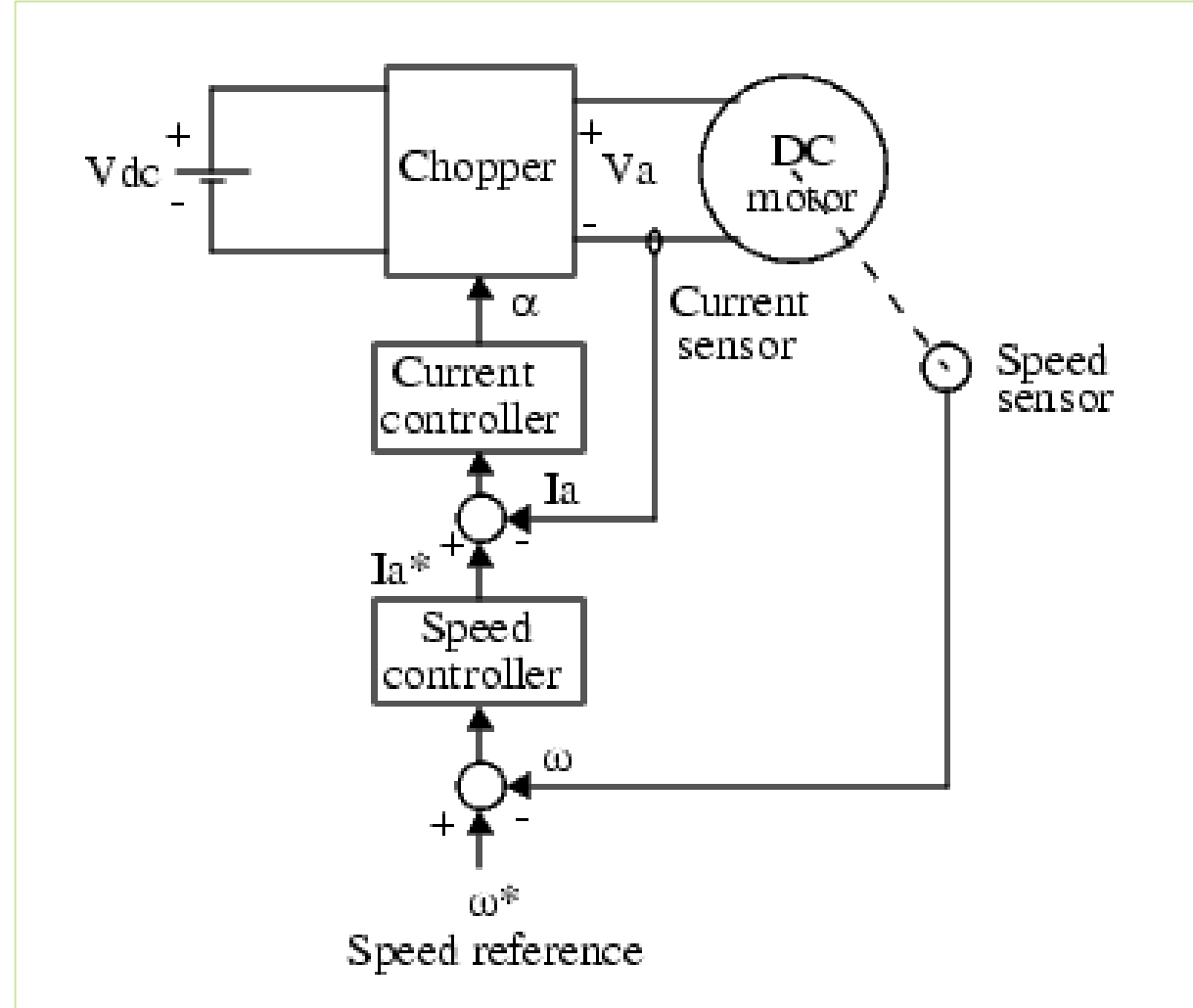


# Circuit Board Hierarchy



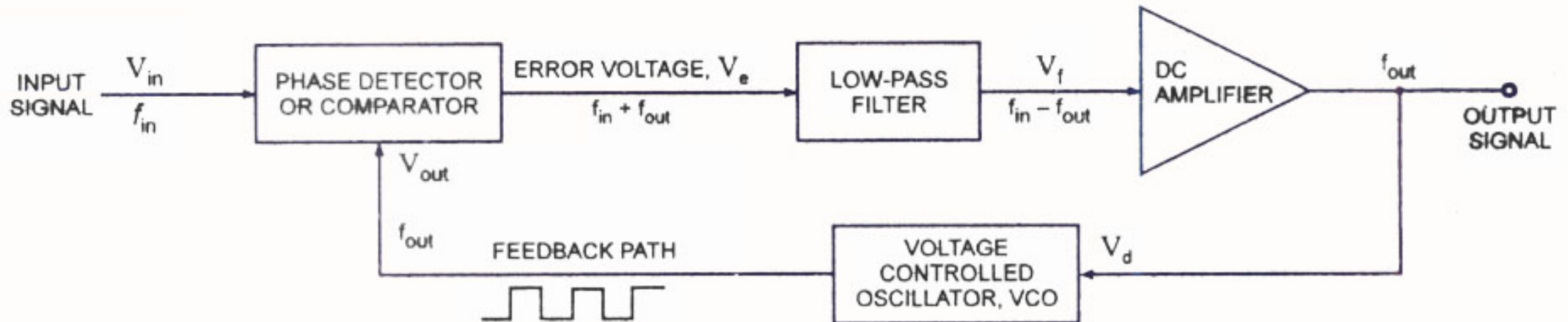
# Multirate Design

- Speed Controller
  - Control motor up to 10,000 RPM
    - $10,000/60 = 166$  cycles/sec
  - Rule of 10 means run speed loop at 2 kHz
- Current Loop
  - Rule of 10 means run current loop at 20 kHz
- How would you design this?



# Multiple Clock Domains

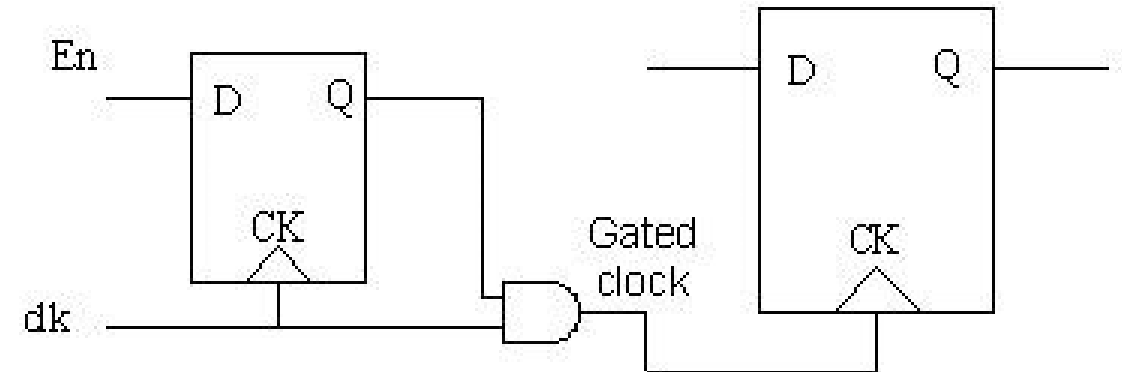
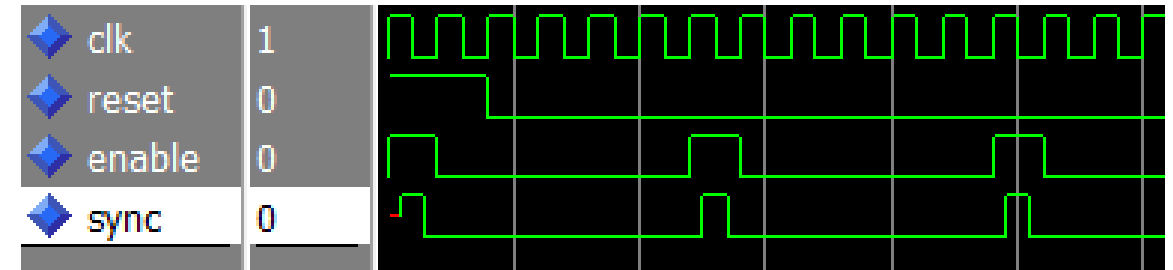
- Generated via phased locked loop [PLL] or digital clock manager[DCM]
  - Ensures phase synchronization
  - Routes new clocks on dedicated routing constructs
- Generated via clock counting
  - New clock will have phase shift relative to clock used for generation



*Block Diagram of Basic Phase-Locked Loop (PLL)*

# Single Domain with Clock Gating

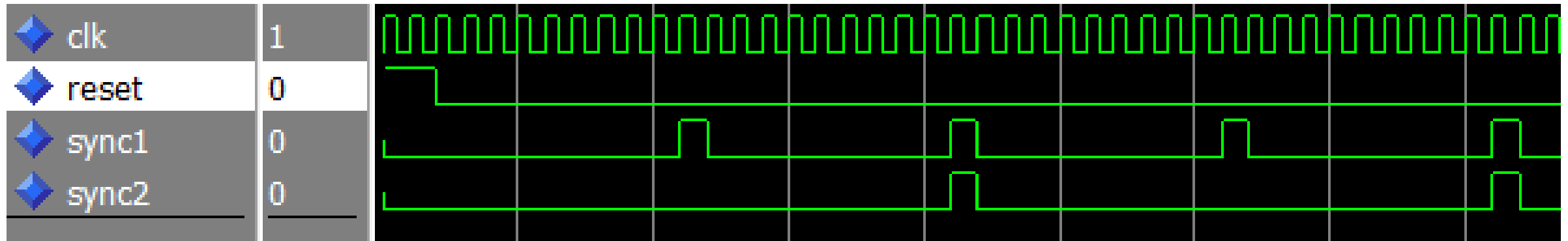
- Lower rate domains generated by “ANDING” the high rate clock with a low rate pulse
- Output clock has phase skew relative to generating clock
- Output clock might not be routed on global clock network
- Glitches on enable signal will appear on output clock signal





# Synchronous Enables

- Single clock domain with synchronous enable pulses
  - The entire design is run on the global clock domain
    - This global clock can make use of the dedicated routing structures in the FPGA
  - Signals are “updated” at different rates however they are all on the same clock domain
  - Signals at different update rates can easily communicate with each other
- Easy to implement with a generic counter module



# Multirate Design Summary

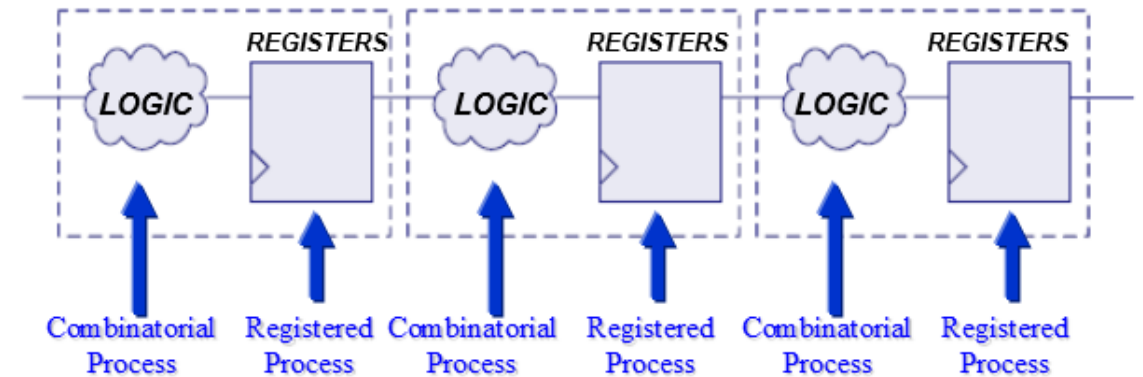
- Multiple clock domains
  - Must add special synchronizer logic for signals on different clock domains to communicate with each other
- Single clock domain with gating
  - Enable signal can change independent of clock signal resulting in very short pulses
  - A glitch enable will result in enabled glitches
  - Can interfere with the construction and analysis of the clock distribution network
- Single clock domain with synchronous enable pulses
  - The entire design is run on the global clock domain
    - This global clock can make sure of the dedicated routing structures in the FPGA
  - Signals are “updated” at different rates however they are all on the same clock domain
  - Signals at different update rates can easily communicate with each other



# Synchronous Sequential RTL Logic Design

- Rules of good synchronous RTL Design:
  - Every circuit element is either a register block or a combinatorial block.
  - All registers receive the same clock signal (important!).
  - Every cyclic path (i.e. feedback) contains at least one register.
- Two most common synchronous sequential circuits
  - Pipelines
  - Finite State Machines

- Recall that all Register Transfer Level circuits always have the same general architecture as shown below:
- In synchronous designs the Registered Processes must all be clocked by the same global clock



- Combinatorial logic does all the decision making.
- Registers store the result of that decision making.

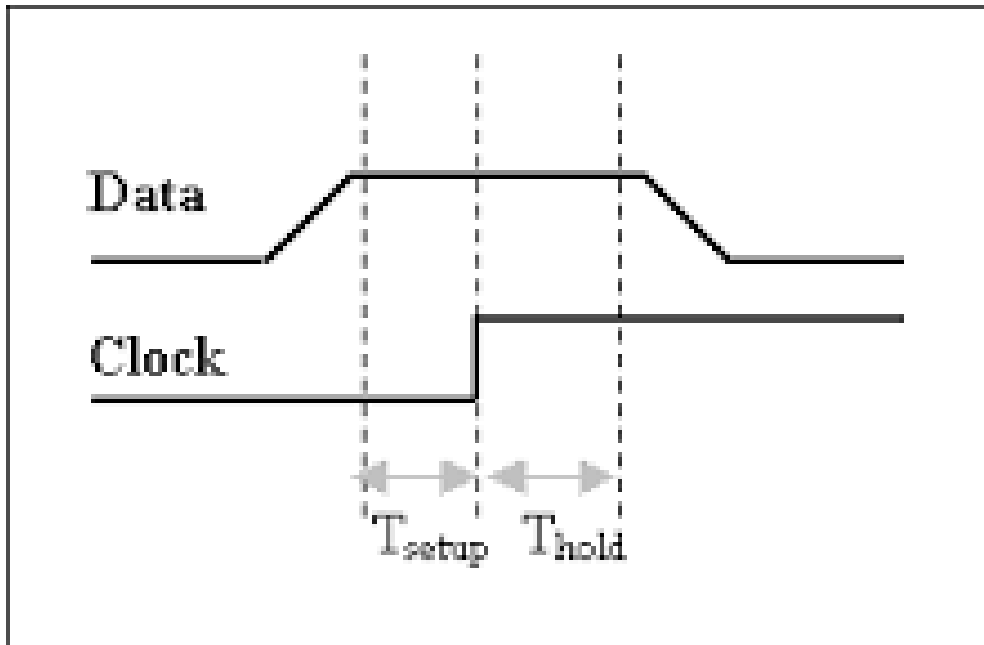
# Delays in Synchronous Designs

## ■ Setup Time

- Time data must be stable before clock edge

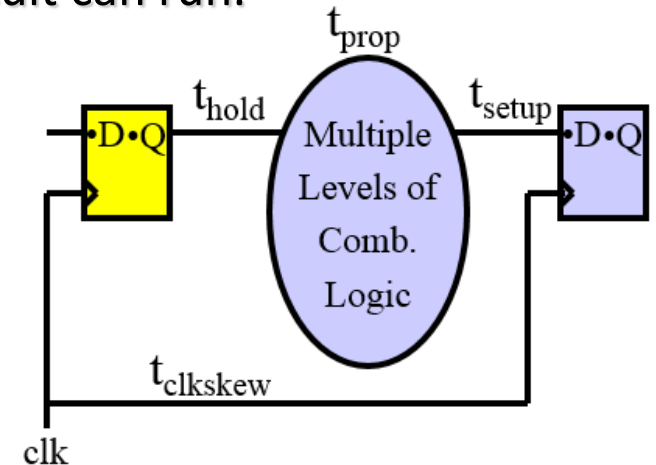
## ■ Hold Time

- Time data must be stable after clock edge



If the combinatorial circuit between two flip-flops becomes too complicated:

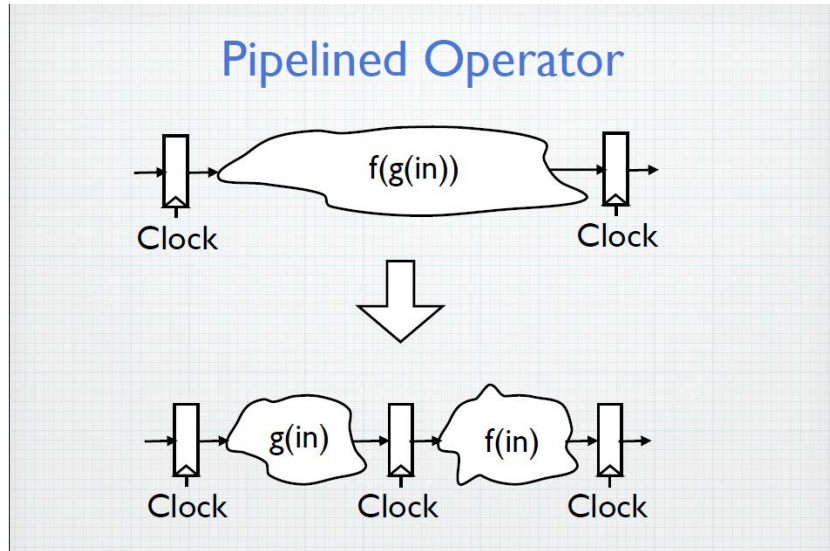
- Consists of multiple levels of decision making.
- Total circuit delay encroaches on global clock period.
- This will limit the maximum clock frequency ( $f_{max}$ ) at which the circuit can run.



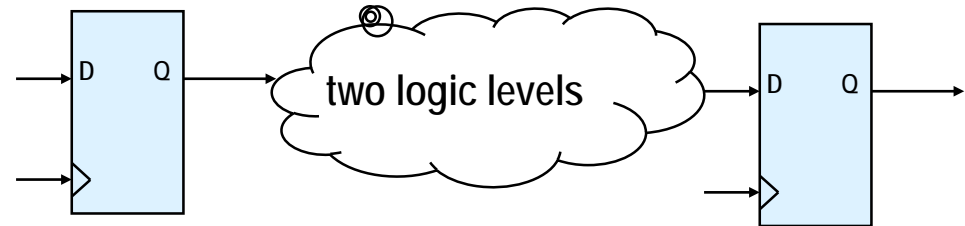
- $$F_{max} = 1 / (t_{hold} + t_{prop} + t_{setup} + t_{clkskew})$$

# Pipelining Concept

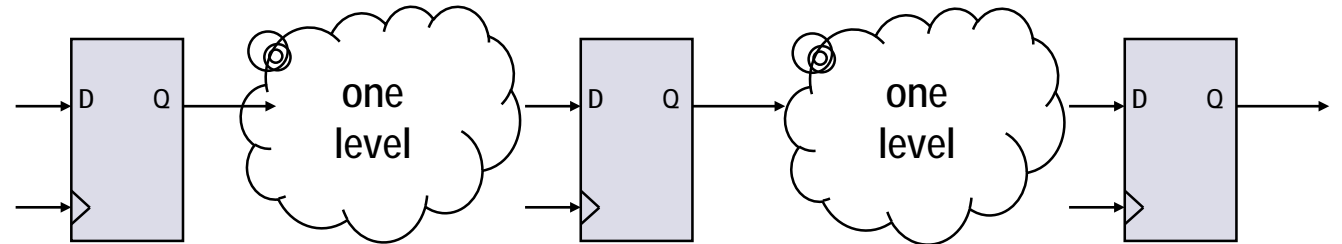
- Inserting flip-flops into a datapath is called “pipelining”.
- Pipelining increases performance by reducing the number of logic levels (LUTs for FPGAs, gates for ASICs) between flip-flops.
- All Altera FPGA device families support pipelining. The basic slice structure is a logic level (four-input LUT) followed by a flip-flop.



$$f_{MAX} = n \text{ MHz}$$



$$f_{MAX} \approx 2n \text{ MHz}$$



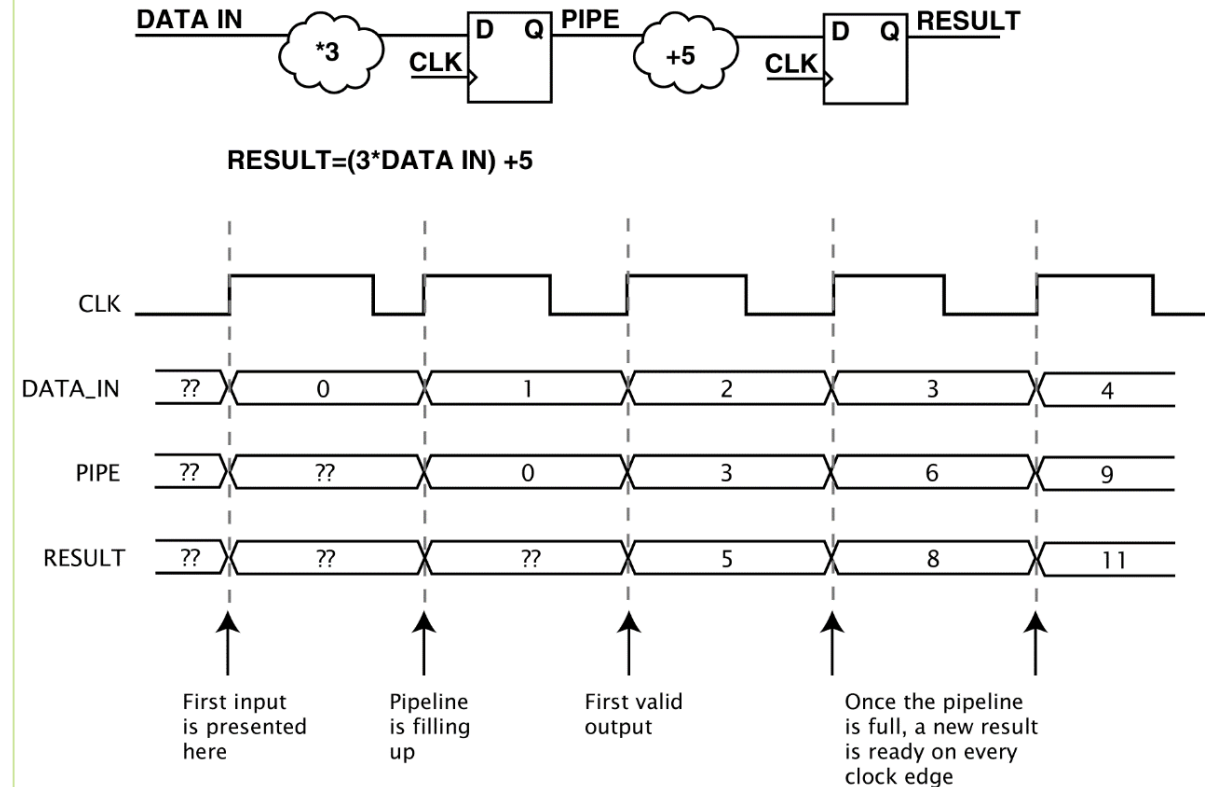
# Latency in Pipelines

Each pipeline stage adds one clock cycle of delay before the first output will be available

- Also called “filling the pipeline”

After the pipeline is filled, a new output is available every clock cycle

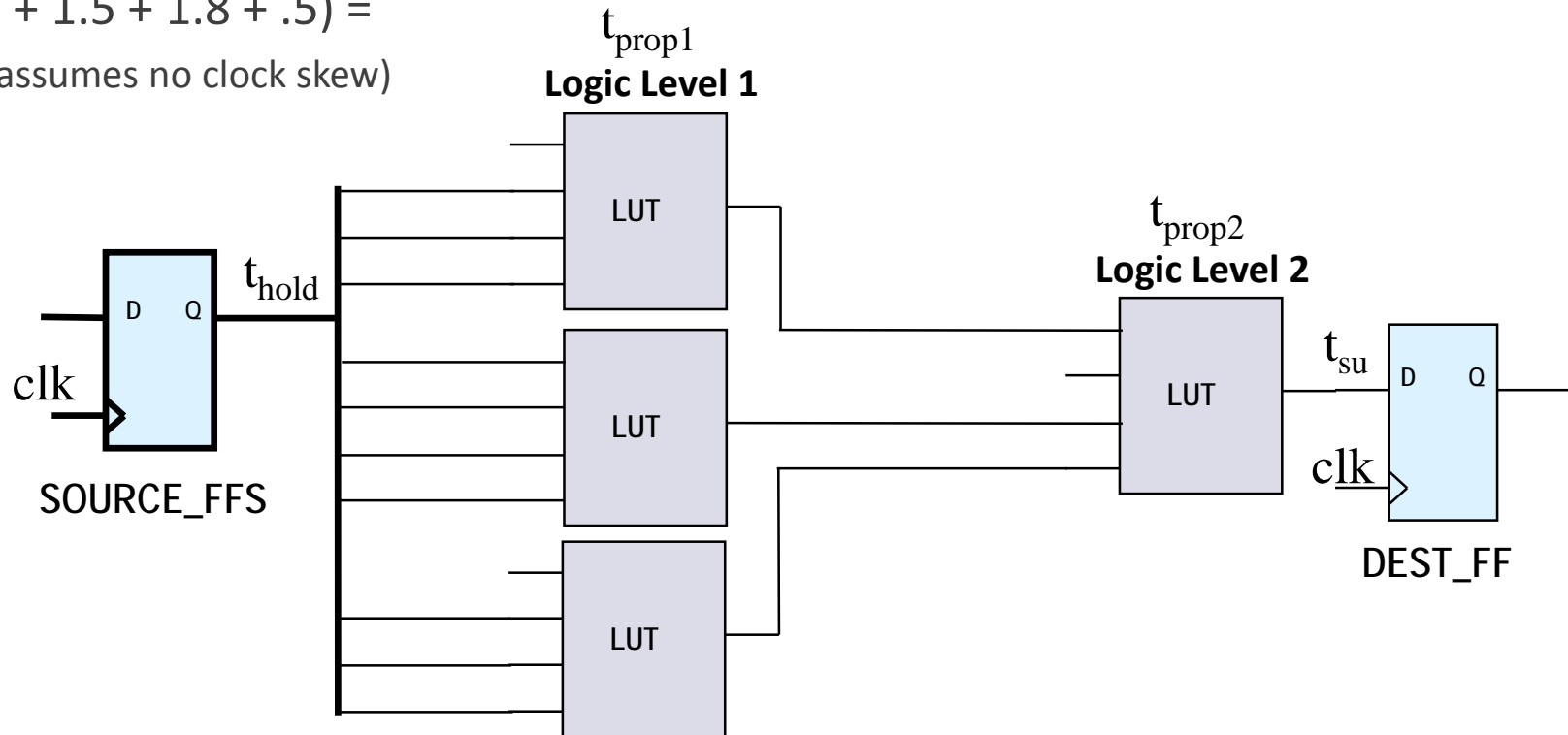
- **Example of a pipelined circuit. Follow the data as it goes through the multiplication and then the addition:**



# Pipelining Example 1

## Original circuit, No pipelining:

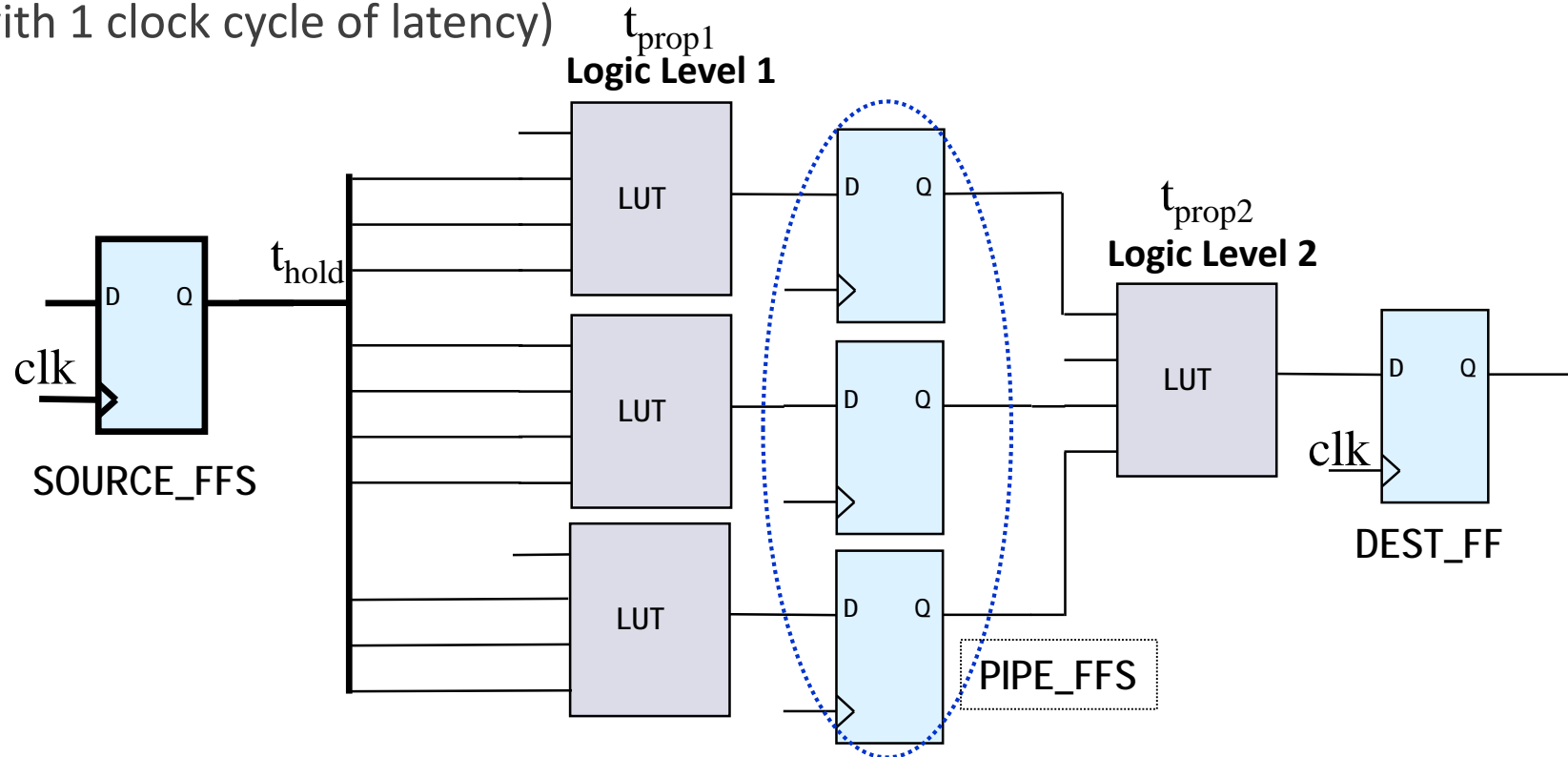
- Two logic levels between SOURCE\_FF and DEST\_FF
- If  $t_{\text{hold}} = .5\text{ns}$      $T_{\text{prop1}} = 1.5\text{ns}$      $T_{\text{prop2}} = 1.8\text{ns}$      $T_{\text{su}} = .5\text{ns}$
- $f_{\text{MAX}} = 1 / (.5 + 1.5 + 1.8 + .5) =$ 
  - $\sim 233\text{ MHz}$  (assumes no clock skew)



# Pipelining Example 2

Pipelined circuit, One pipelining stage:

- One logic level between each set of flip-flops
- $f_{\text{MAX}} = 1/(.5 + 1.8 + .5) = \sim 357 \text{ MHz}$  (  $T_{\text{prop2}} = 1.8 \text{ ns}$  ) limits  $f_{\text{MAX}}$
- $\sim 357 \text{ MHz}$  (But with 1 clock cycle of latency)





# Throughput, Latency, and Area

Throughput/Bandwidth – The amount of data that is processed per clock cycle. Ex. bits/sec

Latency – Time between data input and processed data output. Typical units are clock cycles or time.

Area – Actual silicon gates take up by the design.

**Pipelining increases latency, area, and throughput**

Example 1

Bandwidth: 233 MHz, Latency: 1 clock

Example 2

Bandwidth: 357 Mhz, Latency: 2 clocks, More Gates

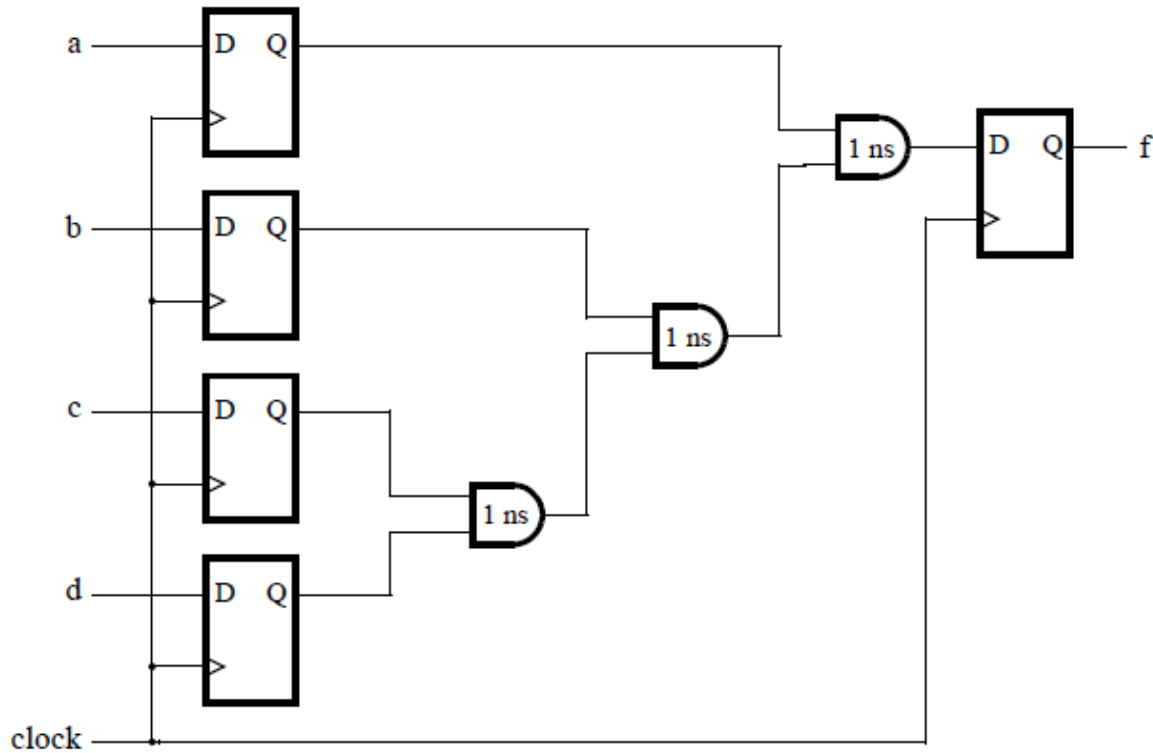
| Instr. No.  | Pipeline Stage |    |    |     |     |     |     |
|-------------|----------------|----|----|-----|-----|-----|-----|
|             | IF             | ID | EX | MEM | WB  |     |     |
| 1           | IF             | ID | EX | MEM | WB  |     |     |
| 2           |                | IF | ID | EX  | MEM | WB  |     |
| 3           |                |    | IF | ID  | EX  | MEM | WB  |
| 4           |                |    |    | IF  | ID  | EX  | MEM |
| 5           |                |    |    |     | IF  | ID  | EX  |
| Clock Cycle | 1              | 2  | 3  | 4   | 5   | 6   | 7   |

Basic five-stage pipeline in a RISC machine (IF = Instruction Fetch, ID = Instruction Decode, EX = Execute, MEM = Memory access, WB = Register write back). In the fourth clock cycle (the green column), the earliest instruction is in MEM stage, and the latest instruction has not yet entered the pipeline.

# Sample Problems

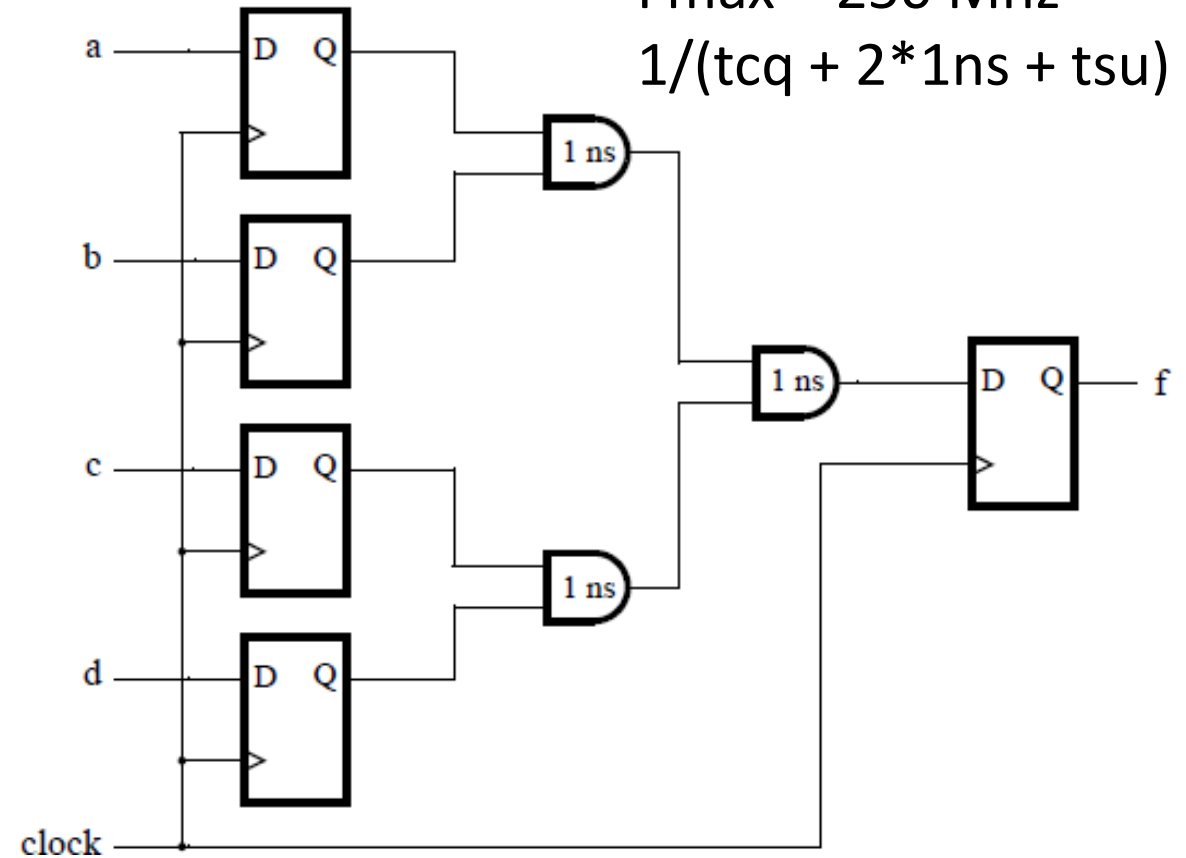
- Clock to Q and Setup Times = 1 ns

$$F_{\max} = 200 \text{ Mhz}$$
$$1/(tcq + 3 \cdot 1\text{ns} + tsu)$$



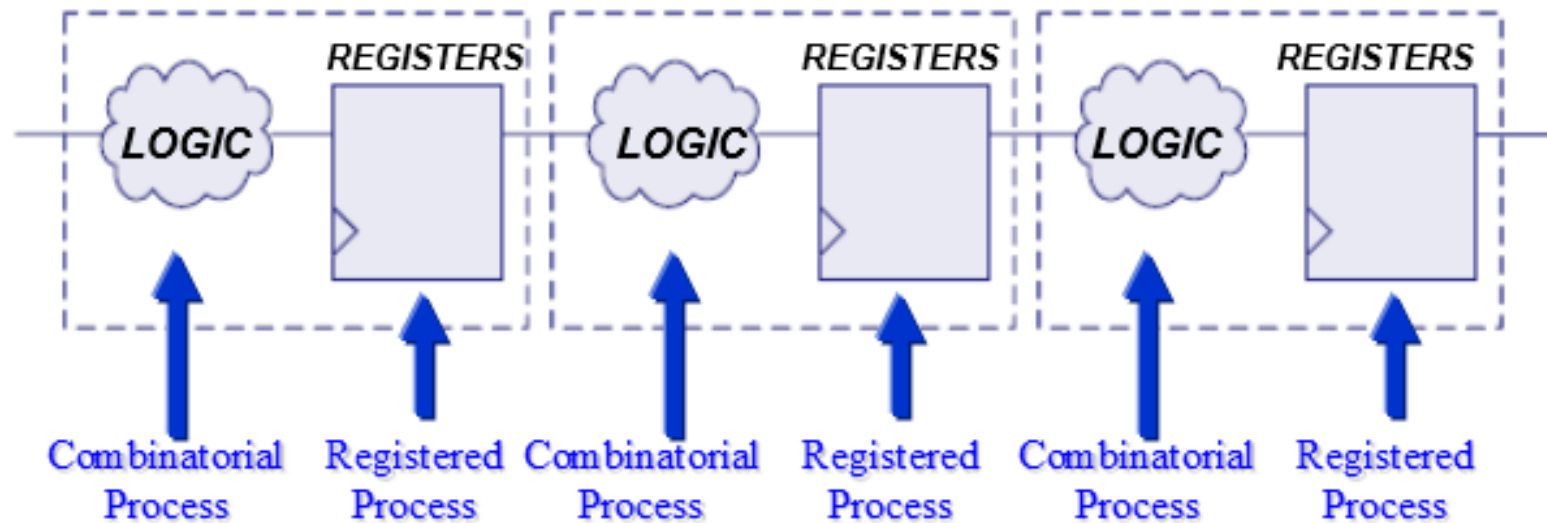
- Clock to Q and Setup Times = 1 ns

$$F_{\max} = 250 \text{ Mhz}$$
$$1/(tcq + 2 \cdot 1\text{ns} + tsu)$$



# FPGA Design with Timing Constraints

- Recall that all Register Transfer Level circuits have the same general architecture as shown below:
- A major component in determining  $F_{max}$  is the routing delays incurred in connecting components together in the FPGA fabric.
- How can one “constrain” the design in order to increase  $F_{max}$ ?



$$F_{max} = 1 / (t_{clk2q} + t_{prop} + t_{setup})$$

# Types of Constraints

- Placement Constraints

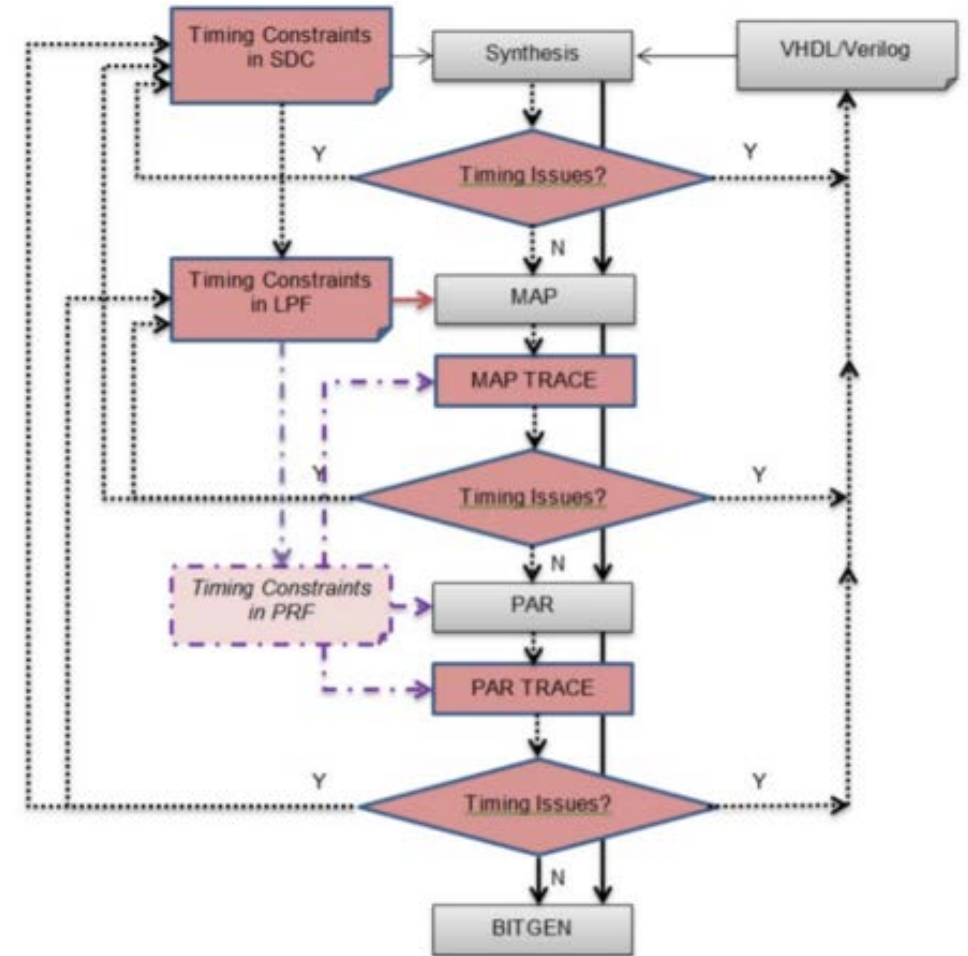
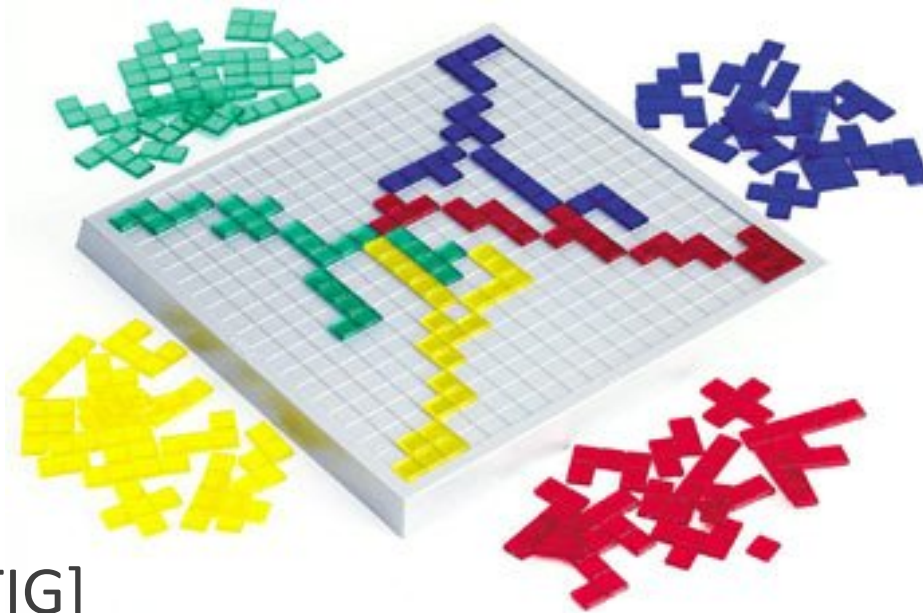
- Pin Constraints
- Area Groups

- Timing Constraints

- Global Timing
  - Period
  - Offset In
  - Offset Out
- Path Specific
  - Multicycle Path
  - False Path

- Timing Ignore [TIG]

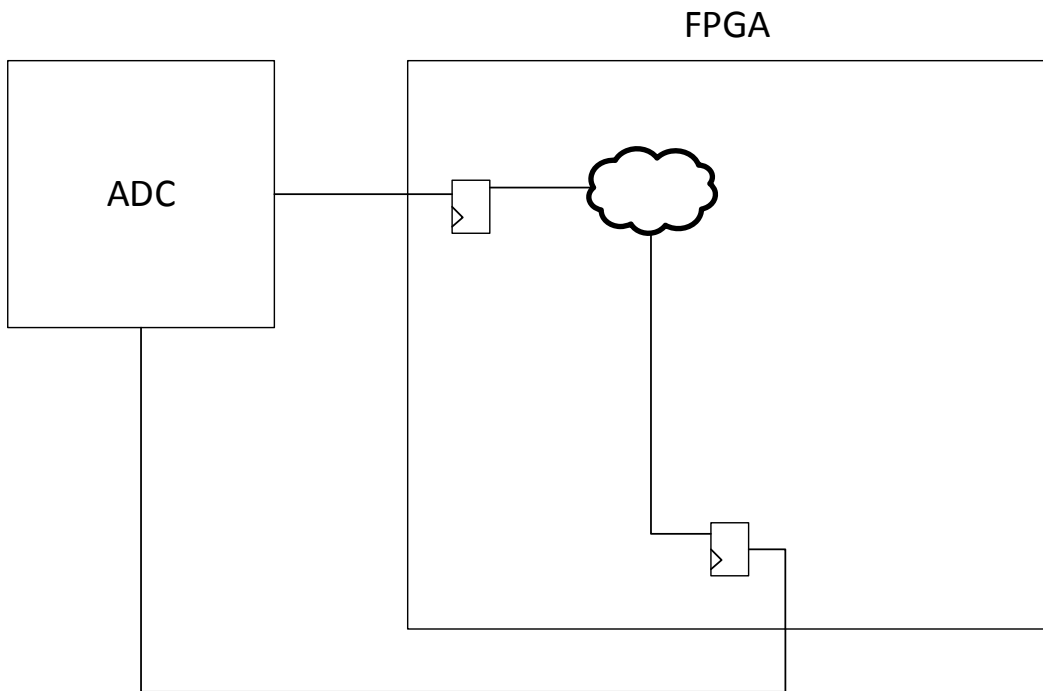
**“place and route algorithms use constraints to optimize where each gate is placed”**



# Proper Pin Constraints in Board Layout Phase

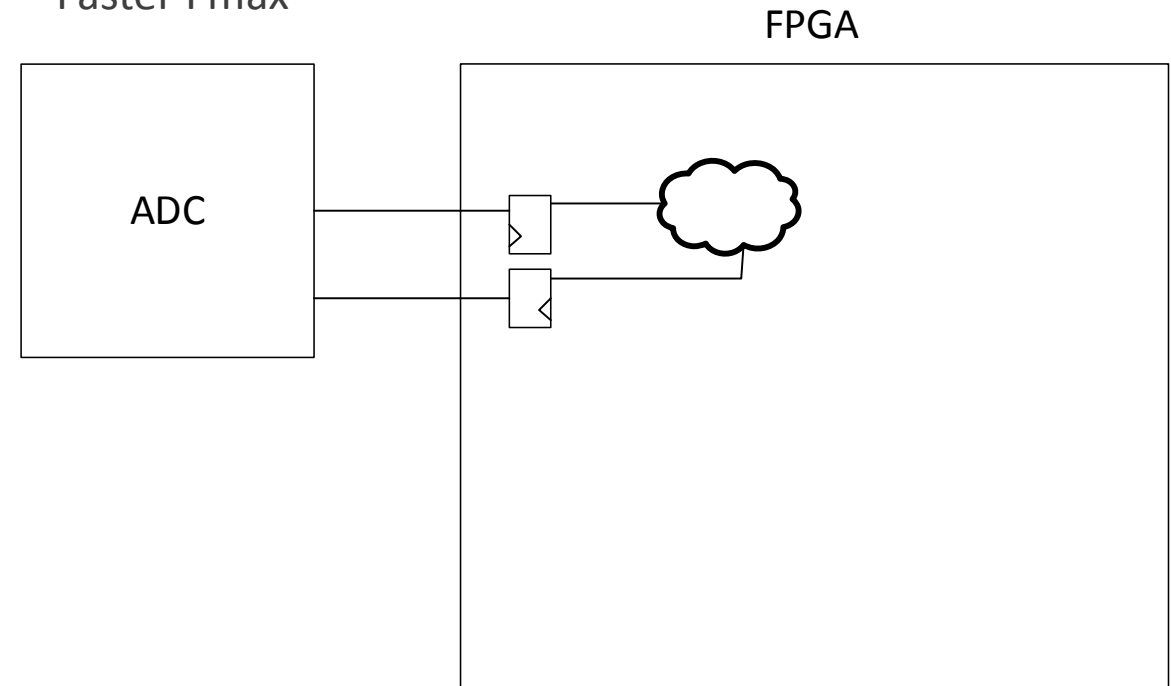
Poor layout gives rise to:

- Large routing delays
- Clock skew
- Slow Fmax



Good layout gives rise to:

- Smaller routing delays
- Less clock skew
- Faster Fmax

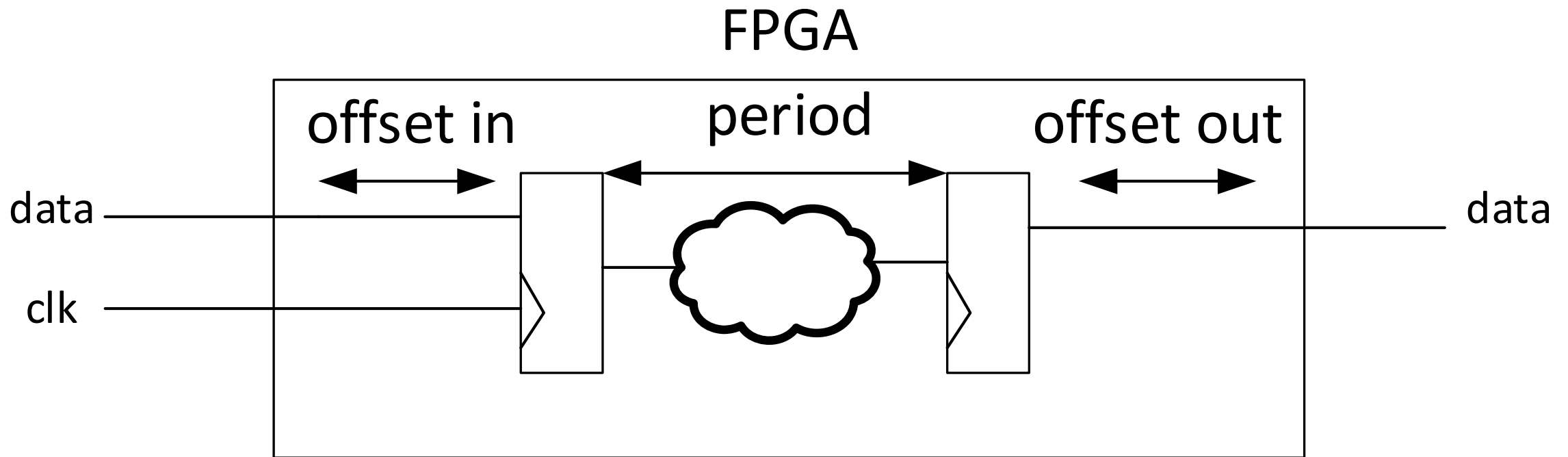


Ex. `set_location_assignment PIN_N2 -to clk`



# Global Timing Constraints

- Period delays are used to constrain a clock frequency
- Offset in and offset out delays are used when communicating with off-chip devices such as analog to digital converters. Can force the FPGA to 'skew' the data in time to compensate for on-chip or off-chip routing delays



# Other Constraints

## ■ Multicycle Path

- When using synchronous enable pulses the registers are effectively updated at a much slower rate than the nominal clock frequency
- Ex. 50 MHz clock with 50 kHz synchronous enable pulses
  - Can have 1000x slack on the path since the data is changing 1000 times slower than the clock

## ■ False Path

- There may be scenarios where certain logic sections can never be executed
- These false paths should not be included in the routers algorithm as they will force the design to be over constrained and possibly fail timing.

## ■ Timing Ignore [TIG]

- There may be occasions where one does not want the router to apply any timing constraints on certain regions of the design. One can tell the router to simply ignore timing for that section.



# Multicycle Path

