Budapest University of Technology and Economics
Department of Electron Devices

Lec. 2
Introduction to HDL/VHDL Coding

Osama Ali

9 of May 2022

Outline

- Introduction to VHDL
- Abstraction levels
- Design hierarchies
- Data types
- Language elements
- Finite State Machine (FSM)
- Simulation results

Introduction to VHDL

Languages for designing hardware

- Higher-level computer languages are used to describe algorithms
  - Sequential execution
- Hardware Description Languages (HDL) are used to describe hardware
  - Not for programming, but for designing hardware
  - Most popular: VHDL, Verilog
  - Parallel (concurrent) execution
    - Instructions are all executed at the same time

What is VHDL?

- Standardization
  - VHDL: IEEE Std-1076-1987, 1993, 2000 (Digital systems)
  - VHDL-**AMS**: IEEE Std-1076.1-1999, 2001 (Analog, digital, and mixed-signal systems)
- VHDL features
  - Formal language for modelling a digital circuit
  - Modelling concepts are derived from the operational characteristics of digital circuits
  - Source code is interchangeable among the different tools
- Main steps of a VHDL-based design procedure:
  - Code writing, compiling, simulation, & synthesis
  - Synthesis is a process where a VHDL is compiled and mapped into an implementation technology such as an FPGA or an ASIC. Not all constructs in VHDL are suitable for synthesis.

VHDL vs. Verilog

VHDL	Verilog
All abstraction levels	All abstraction levels
Complex grammar	Simple grammar
Lots of data types	Few data types
User-defined type definition	No user defined type definition
User-defined libraries, packages, configurations	No user-defined libraries, packages, configurations
Full design parameterization	Simple design parameterization
Very consistent language (e.g. strong typing rules). Code behaves exactly the same in every simulator	Less consistent language. If the designer doesn't follow ad-hoc coding styles, it executes differently on different platforms
Case insensitive	Case sensitive

Abstraction levels in VHDL

- VHDL is rich in language abstractions, in addition to which the language can be used to describe different abstraction levels
  - Dataflow
  - Structural
  - Behavioural
- Abstraction levels are means of **concealing/hidden** details
- The design of VHDL components described on higher abstraction levels can be **technology-independent**
  - It is usually a requirement to determine the abstraction level at which the information is to be described
  - If a short development time is required, a high abstraction level should be chosen as the way of modelling

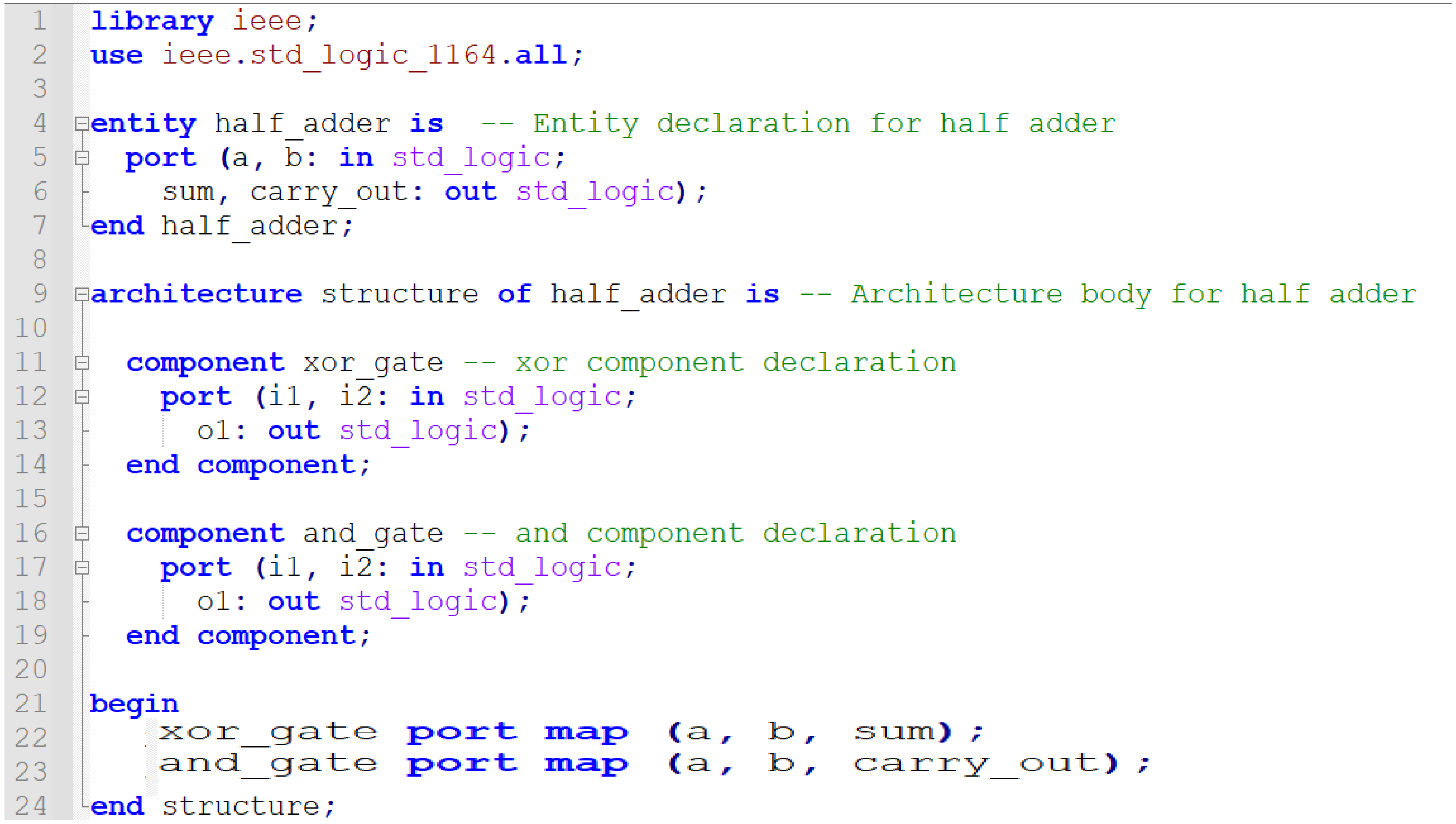
Dataflow

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

entity half_adder is
  port (a, b: in std_logic;
    sum, carry_out: out std_logic);
  end half_adder;

architecture dataflow of half_adder is
  begin
  sum <= a xor b;
  carry_out <= a and b;
end dataflow;
```

Behavioural

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

entity half_adder is
  port (a, b: in std_logic;
    sum, carry_out: out std_logic);
  end half_adder;

architecture behavior of half_adder is
  begin
    ha: process (a, b)
    begin
      if a = '1' then
        sum <= not b;
        carry_out <= b;
      else
        sum <= b;
        carry_out <= '0';
      end if;
    end process ha;

end behavior;
```

Structural



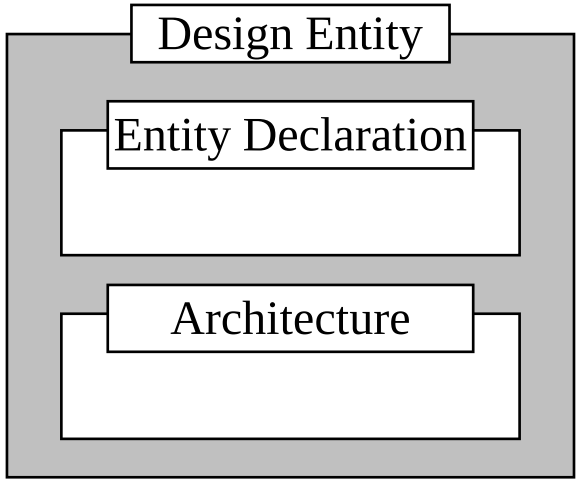
```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity half_adder is  -- Entity declaration for half adder
  port (a, b: in std_logic;
    sum, carry_out: out std_logic);
end half_adder;

architecture structure of half_adder is -- Architecture body for half adder

  component xor_gate -- xor component declaration
    port (i1, i2: in std_logic;
      o1: out std_logic);
  end component;

  component and_gate -- and component declaration
    port (i1, i2: in std_logic;
      o1: out std_logic);
  end component;

begin
    xor_gate port map (a, b, sum);
    and_gate port map (a, b, carry_out);
end structure;
```

Design hierarchies

Mechanisms to reduce complexity

- Language abstractions are useful to describe complex matters without having to describe small details
- Black box principle
- In some cases there is no need to know how the component is internally structured
  - The designer is usually only interested in
    - Inputs and outputs
    - Specification function
    - Delay times

Primary language abstraction



- The primary abstraction is the design entity
- It is the basic unit of hardware description
- The design entity can represent a **cell, chip, board, or subsystem**

- Aspects of modelling a system: *interface* and *function*
- An entity declaration defines the interface between the entity and the environment outside of the design entity
- An entity can be linked to several architectures
    - E.g., one architecture may model an entity at a **behavioural level**, while another architecture may be a **structural model**

Design entity and component

- Entity is a component of a design
  - Component reusability
  - Ports
    - The inputs and outputs of the circuits
    - They are special programming objects
    - Ports are signals **(But)?**
    - Ports are the means used by the circuit to communicate with the external world, or with other circuits
    - Each port must be declared to be of a particular type

Syntax of the entity declaration

```
entity <identifier_name> is
port( [signal] ~~<identifier>:[<mode>] <type_indication>;~~
        ...
        [signal] <identifier>:[<mode>] <type_indication>);
end [entity] [<identifier_name>];
```

The word **signal** is normally left out of the port declaration, as it does not add any information

Mode **in** and the name of the **entity** after **end** can also be left out

<mode> = **in, out, inout**

- **in**: Component only read the signal
- **out**: Component only write to the signal
- **inout**: Component read or write to the signal - bidirectional signals

Example for simplification of an entity declaration

```
entity gate1 is
port(signal a: in bit;
     signal b: in bit;
     signal c: out bit);
end gate1;

entity gate1 is  -- Identical with the above example
port( a,b: bit;
       c: out bit);
end;
```

Syntax of the architecture

architecture <architecture_name> **of** <entity_identifier> **is**

[<architecture_declarative_part>]

begin

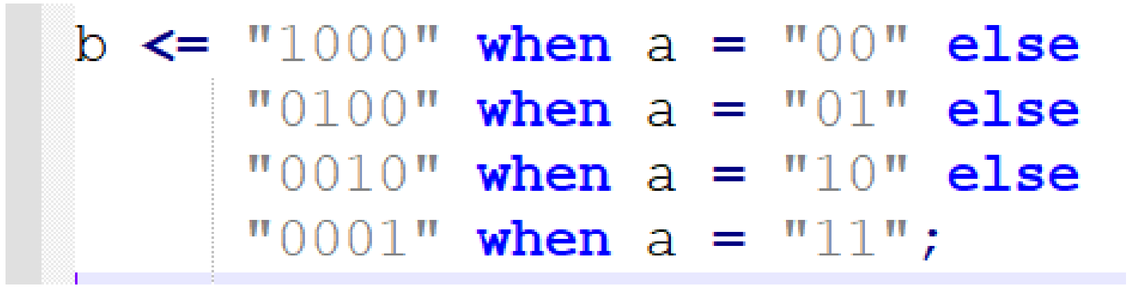
  <architecture_statement_part>

end [**architecture**] [<architecture_name>];

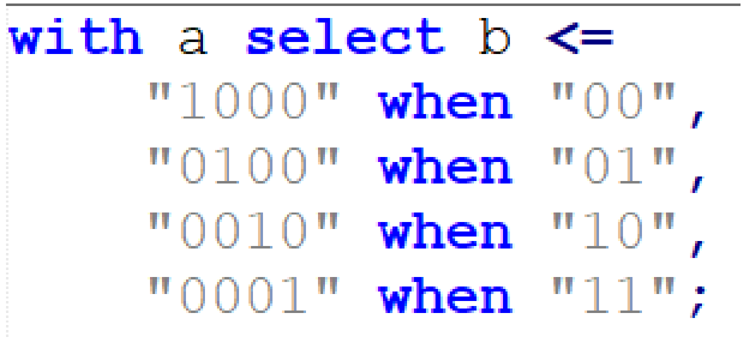
- The architecture declaration part must be defined before first begin and can consist of, for example:
  - Types
  - Subprograms
  - Components
  - Signal declarations

CSA: Concurrent Signal Assignment

- Each architecture consists of concurrent statements, which are executed concurrently with respect to simulated time
    - The order of execution of the statements is dependent upon the flow of values and not on the textual order of the program
- CSA: Concurrent Signal Assignment
    - This is a major difference between VHDL and ordinary computer languages Types
        - Simple CSAs
        - Conditional CSAs: ***when* statement**
        - Selected CSAs: ***with – select* statements**



```
b <= "1000" when a = "00" else
     "0100" when a = "01" else
     "0010" when a = "10" else
     "0001" when a = "11";
```



```
with a select b <=
    "1000" when "00",
    "0100" when "01",
    "0010" when "10",
    "0001" when "11";
```

Data types
Objects and containers

Objects

- Objects are containers for values of a specified type
- Once an object is declared of a certain type, operations can be performed on the object within the bounds set in the type declaration
- If objects of different types are mixed or exceed boundaries set by the type declaration, an error is displayed
- Type of objects
  - *Variables*
    - Sequential objects
  - *Signals*
    - Concurrent objects
    - Unlike variables, signals have an associated time value
    - The signal retains this value until it is assigned a new value at a future point in time
  - *Constants*
- Declarations of signals, variables and constants need specify their corresponding type or subtype

Classes of types

- *Scalar (range) types*
  - **Integer** types, **floating** point types, enumeration types
    - E.g. of enumeration types “**type** wireColor **is** (red, black, green);”
- *Composite types*
  - Array, record
  - *Access type “*These types provide access to objects” **Not supported by synthesis tools**

Scalar types: Integer

- Sets of positive or negative whole numbers
- Their range are machine dependent, typically $\pm$2.147.483.648 for **32-bit** systems

-- Declaration in standard.vhd package

type integer **is range** -2147483647 **to** 2147483647**;**

-- User declared integer type

type testInteger **is range** -100 **to** 100;

Scalar types: Floating point

- Defines a collection of numbers that provide an approximation to real numbers
- Problem: it is not possible for hardware to handle **infinitely** long real numbers

-- Declaration in the package standard.vhdl

type real **is range** -1.7e38 **to** 1.7e38

-- Bounds are implementation dependent

type half_hour **is range** 0.0 **to** 29.99;

Composite types: Array

- A named array is a collection of elements that are of the same type
- Arrays may be configured in one or more dimensions
- Each array element is referenced by one or more index value

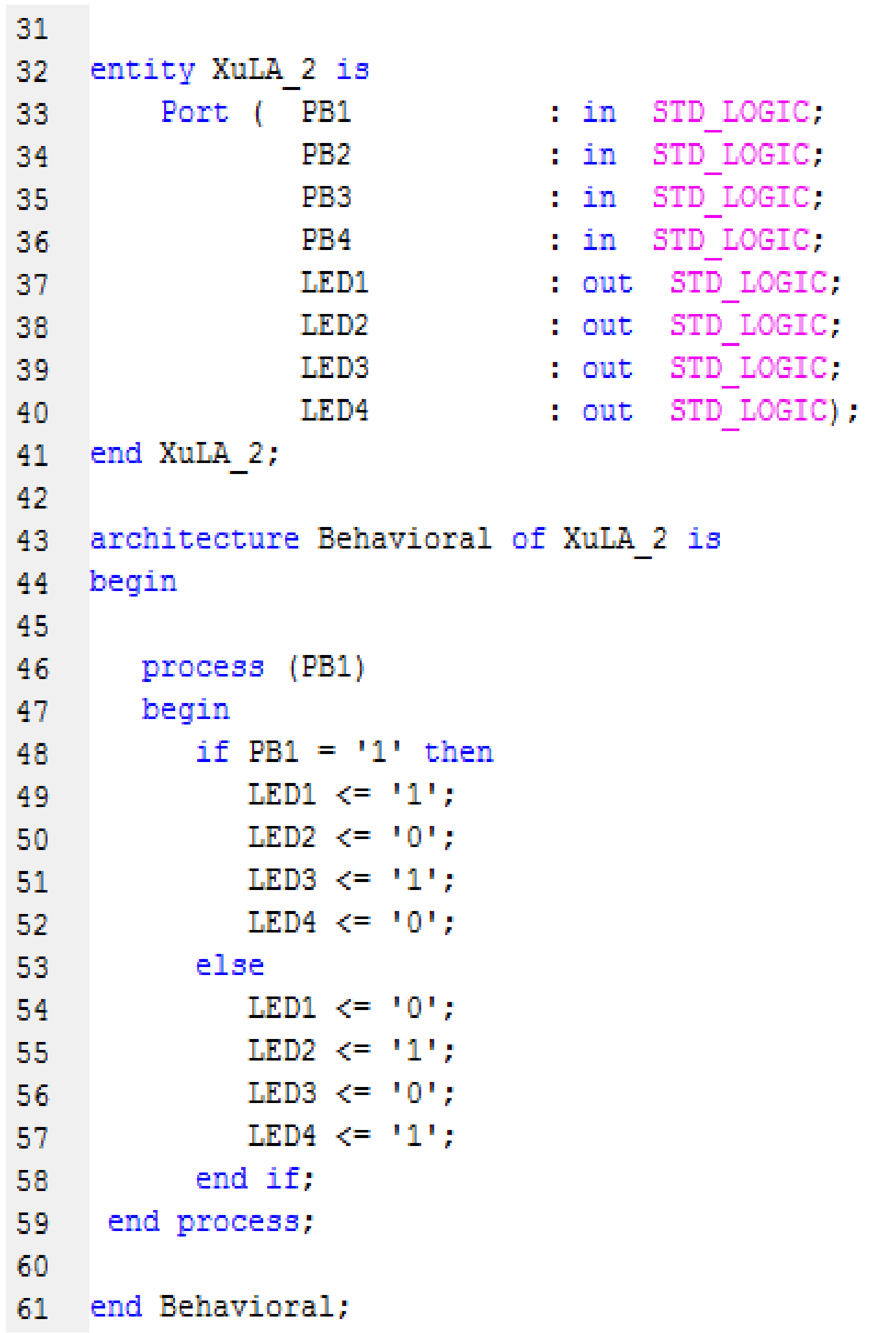
```
        **type** array10 **is array** (0 **to** 9) **of** integer;
-- Std_ulogic_vector is defined as:
        **type** std_ulogic_vector **is array** (natural **range** <>) **of** std_ulogic;
-- Std_logic_vector is defined as:
        **type** std_logic_vector **is array** (natural **range** <>) **of** std_logic;
-- Bit_vector is defined as:
        **type** bit_vector **is array** (natural **range** <>) **of** bit;
```

Comments, spaces & labels

- Comments follow (two consecutive dashes or hyphens) '--' and instruct the analyser to ignore the rest of the line
  - The **parser** ignores anything after the two dashes and up to the end of the line in which the dashes appear
  - There are no multiline comments in VHDL
- VHDL is not sensitive to white spaces (**spaces** and **tabs**)
  - Tabs improve readability, but it is best not to rely on a tab as a space in case the tabs are lost or deleted in conversion

Process

- A *process* is a concurrent statement
    - All statements in a process are executed sequentially until the process is suspended via a *wait* statement
    - Within a process, procedures and functions can partition the sequential statements
- A process can be a single signal assignment statement or a series of sequential statements
- Upon initialization, all processes are executed once
- Processes are executed in a data-driven manner, and activated
    - by events on signals in the process *sensitivity* list or
    - by waiting for the occurrence of specific events using the *wait* statement
- The **sensitivity list** – being next to the process keyword – is a list of those input signals to the component to which the process is sensitive



```

entity XuLA_2 is
    Port (  PB1           : in  STD_LOGIC;
            PB2           : in  STD_LOGIC;
            PB3           : in  STD_LOGIC;
            PB4           : in  STD_LOGIC;
            LED1          : out  STD_LOGIC;
            LED2          : out  STD_LOGIC;
            LED3          : out  STD_LOGIC;
            LED4          : out  STD_LOGIC);
end XuLA_2;

architecture Behavioral of XuLA_2 is
begin

   process (PB1)
   begin
      if PB1 = '1' then
         LED1 <= '1';
         LED2 <= '0';
         LED3 <= '1';
         LED4 <= '0';
      else
         LED1 <= '0';
         LED2 <= '1';
         LED3 <= '0';
         LED4 <= '1';
      end if;
 end process;

end Behavioral;
```

An infinite loop

- When a process is completed, it enters in *suspend* mode until the next change in its sensitivity list
- If there is no sensitivity list, then the process will run forever

```
process
begin
        signalName <= ‘1’;
end process;
```

The wait statement

wait_statement::=
  [label :] **wait** [sensitivity_clause] [condition_clause] [timeout_clause] ;

- It is permissible to have several *wait* commands in the same process

wait for *time expression*;

  - This causes suspension of the process for a period of time given by the evaluation of the time expression

  wait for 10 ns;

    - The simulator waits for 10 ns before continuing execution of the process
    - The starting time point of the waiting is significant and not the actual changes of any signal value

The wait statement (continued)

wait on *signal1, signal2,* …;

- The process is interrupted until the value of one of the signals changes
- An event on any of the signals, causes the process to resume execution at the next statement after the *wait* statement

wait on a, b;

- It suspends execution until a change (event) occurs on either signal *a* or *b*

wait until *condition*;

- The Boolean expression is evaluated whenever one of the signals in the expression changes, and the process continues execution when the expression evaluates to true

wait until a=‘1’;

- It is satisfied when signal *a* has an event (changes value), and the new value is ‘1’, i.e. a rising edge for signal *a*

wait until signalName=‘1’ **for** 10 ns;

- The wait condition is satisfied when *a* changes value or after a wait of 10 ns (regarded as an or condition)

wait;

- The process is permanently suspended

Sensitivity list vs. wait statement

- If a process has a sensitivity list, then it cannot contain a *wait* statement
- Using the sensitivity list is identical to a *wait on* statement at the end of the process, if the group of signals are the same
  - The both processes are executed at first time
  - The both processes are triggered each time that signal *s1* or *s2* changes value

```
p0: process (s1, s2)
begin
        if s1>s2 then
        q<='1';
        else
            q<='0';
        end if;
end process;
```

```
p1: process
begin
        if s1>s2 then
            q<='1';
         else
            q<='0';
         end if;
        wait on s1, s2;
end process;
```

Examples of wait statement

signal a: **in** bit; c1, c2, c3, c4, c5, c6, c7: **out** bit;

```
-- Example 1
process (a)
begin
   c1<= not a;
end process;
```

```
-- Example 2
process
begin
   c2<= not a;
   wait on a;
end process;
```

```
-- Example 3
process
begin
   wait on a;
   c3<= not a;
end process;
```

```
-- Example 4
process
begin
   wait until a='1';
   c4<= not a;
end process;
```

```
-- Example 5
process
begin
  c5<= not a;
  wait until a='1';
end process;
```

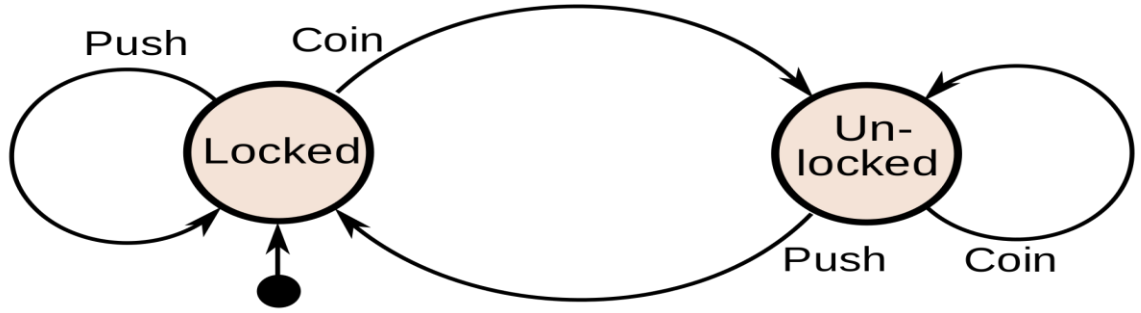
```
-- Example 6
process
begin
  c5<= not a;
  wait for 10 ns;
end process;
```

```
-- Example 7
process
begin
  c5<= not a;
  wait until a='1' for 10 ns;
end process;
```

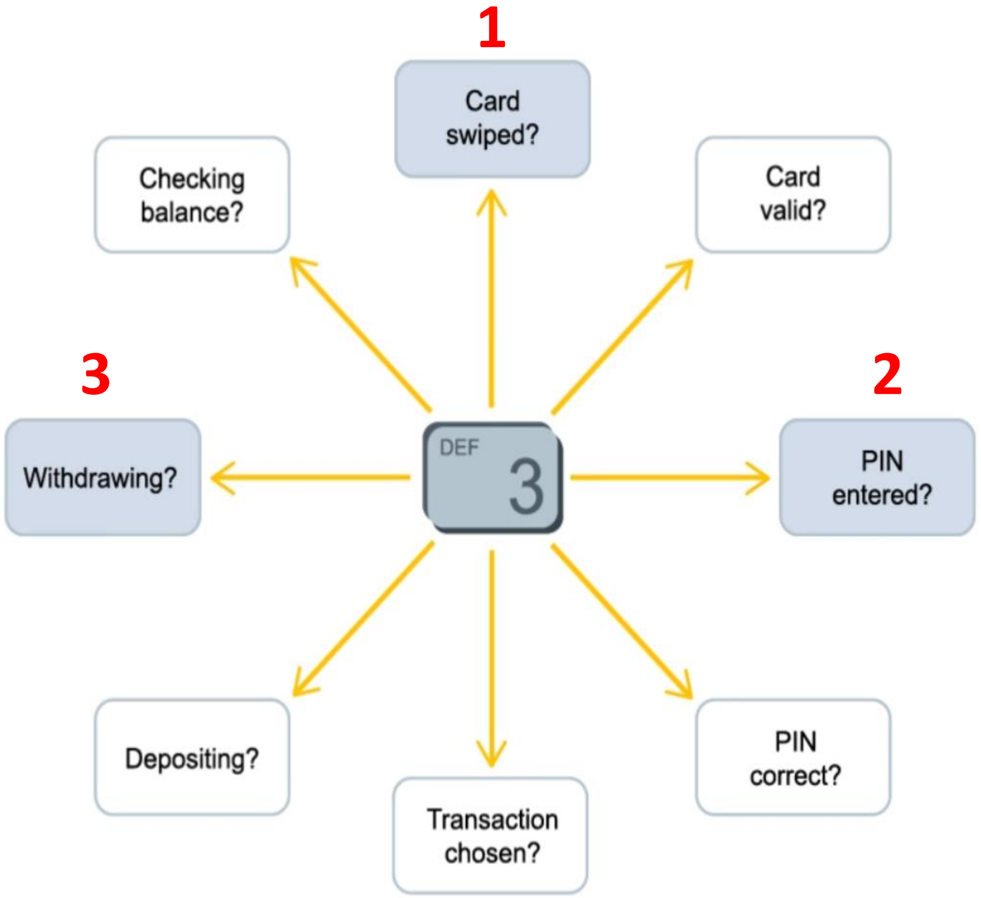
- Note that the process does not wait for event at the signal assignment

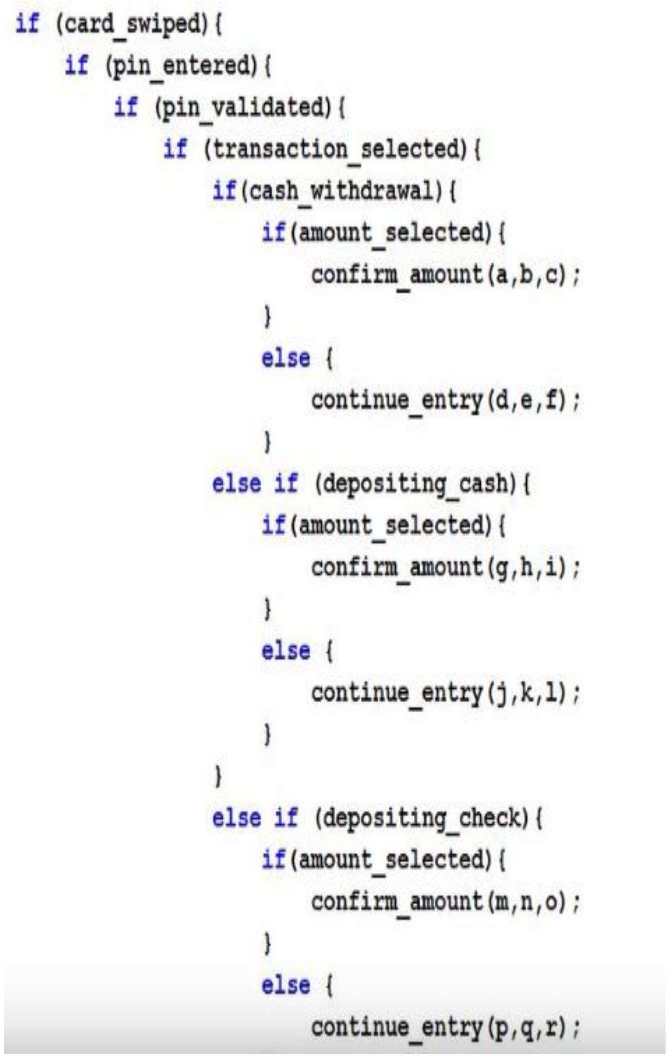
FSM : Finite State Machine

- A sequential circuit is a circuit with memory
- FSM is a mathematical model of the sequential circuit with discrete inputs, discrete outputs and finite number of internal configuration or states.
- So, it is special modelling technique for sequential logic circuit.
- Example of FSM



FSM: example and benefits





```
if (card_swiped){
    if (pin_entered){
        if (pin_validated){
            if (transaction_selected){
                if(cash_withdrawal){
                    if(amount_selected){
                        confirm_amount(a,b,c);
                    }
                    else {
                        continue_entry(d,e,f);
                    }
                else if (depositing_cash){
                    if(amount_selected){
                        confirm_amount(g,h,i);
                    }
                    else {
                        continue_entry(j,k,l);
                    }
                }
                else if (depositing_check){
                    if(amount_selected){
                        confirm_amount(m,n,o);
                    }
                    else {
                        continue_entry(p,q,r);
```