Fundusze Europejskie
Wiedza Edukacja Rozwój

Rzeczpospolita Polska

Unia Europejska
Europejski Fundusz Społeczny

Politechnika Śląska jako Centrum Nowoczesnego Kształcenia
opartego o badania i innowacje

POWR.03.05.00-IP.08-00-PZ1/17

Projekt współfinansowany przez Unię Europejską ze środków Europejskiego Funduszu Społecznego

Digital Circuits Design

**Faculty of Automatic Control, Electronics and Computer Science,
Informatics, Bachelor Degree**

Lecture 13.

Hardware Description Language

HDL

- The HDL is not a programming language!
- The hardware description language is understood as a description of the operation and/or **construction** of the electronic circuit
- The source is called a **description** or **model**

- **HDLs describe hardware, so they describes mostly „concurrency”!!!**

HDL

- Created for simulation
- Also used for synthesis

HDL

- What is the result of this fragment of program?

Let a=2; b=4;

```
…
a= b;
b= a;
```

- a=b=4   in the case of sequential execution of instruction
- a=4, b=2 in the case of parallel execution of instruction

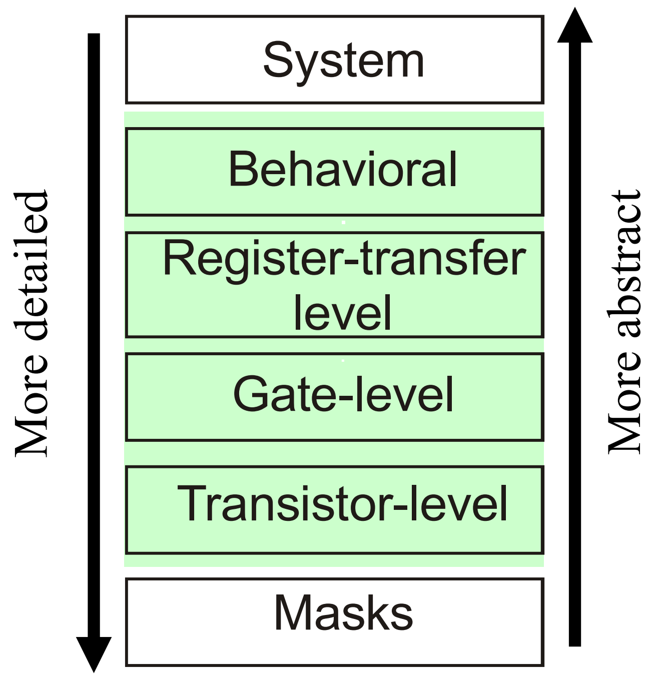
HDL

- The method of synthesis depends on the whole context of the description and not just the instruction used!

```
always @(posedge CLK)
    case (A)
        2'b00  : Y <= 4'b0001;
        2‘b01  : Y <= 4'b0010;
        2'b10  : Y <= 4'b0100;
        2'b11  : Y <= 4'b1000;
        default: Y <= 4'b0001;
    endcase
```

```
always @(A)
    case (A)
        2'b00  : Y <= 4'b0001;
        2‘b01  : Y <= 4'b0010;
        2'b10  : Y <= 4'b0100;
        2'b11  : Y <= 4'b1000;
        default: Y <= 4'b0001;
    endcase
```

Abstraction levels



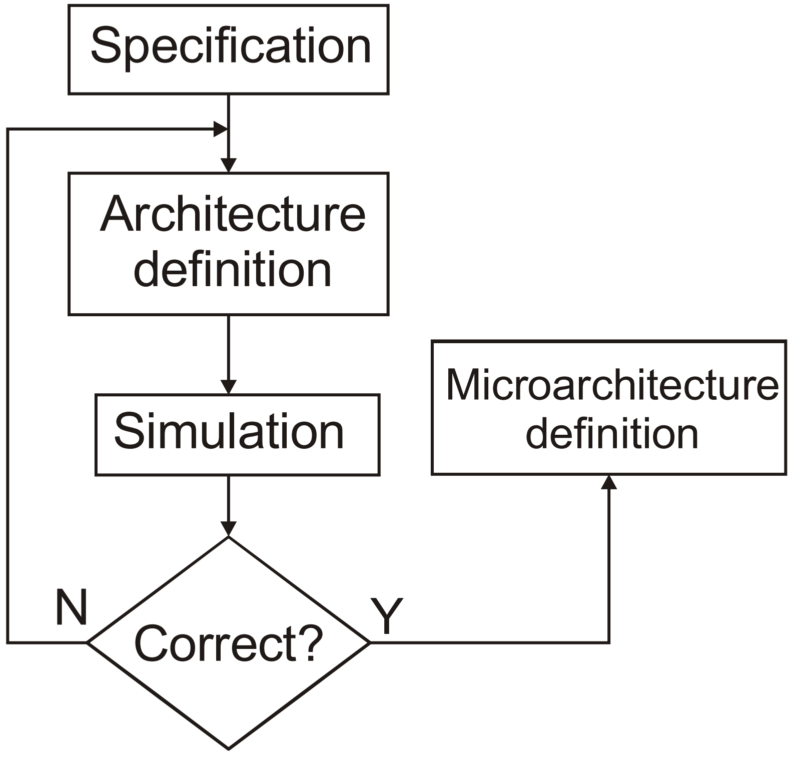
- Behavioral – a description of „how it works”

- Register-transfer level (RTL) – a description of the structure of the design

- Gate-level – the circuit is described by means of logic gates

- Transistor-level – the circuit is described by means of transistors

DCD design flow – specification



Verilog - HDL

- Synthesizable elements of the Verilog
  - module

Module

- The module is only a „template” on which objects are created
- The body of the module can be described in different abstraction levels:
  - Functional-level,
  - RTL,
  - Gate-level,
  - Transistor-level.
- Modules can be (should be) parameterized
- The modules definitions cannot be nested!

Module

```
module <mod_name>(<port_list>);
<port_declaration>;
<further_decl>;   //parameters,
                  //nets, registers
                  //variables

<module_body>;  //instances,
                 //continuous assignments,
                 //function and tasks


endmodule
```

where:

`<mod_name>` is the name of the module

`<port_list>` is list of ports (inputs, outputs and bidirectional ports)

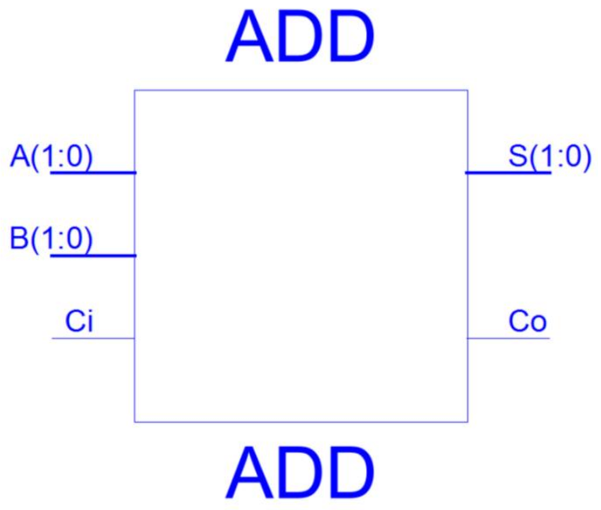
Module – port declaration

- Port modes
  - `input`
  - `output`
  - `inout`
- Ports can be 1-bit (scalar) or multi-bit (vector)
- The number of bits is defined by means of the structure[a:b]
- Example

```
input [1:0] A,B; //A and B 2-bit
input Ci; //Ci 1-bit
output [1:0]S;
output Co;
```

Module – example

```
module ADD(A, B, Ci, S, Co);
    input [1:0] A,B; //A i B 2-bit
    input Ci; //Ci 1-bit
    output [1:0]S;
    output Co;

        //body

endmodule
```



Instances

- Module is only template to making objects
- An object created from the template is called na instance
- Every instance must have unique name

- Instance

```
<mod_name> <instance_name>(<port_list>);
```

Module instantiation

```
module ADD(A, B, Ci, S, Co);
input A, B, Ci;
output S, Co;
...
endmodule
```

Connections by names

```
module Top;
reg a, b, ci;
wire s, co;

ADD SUM1(.A(a),.B(b),.Ci(ci),.S(s),.Co(co));
...
endmodule
```

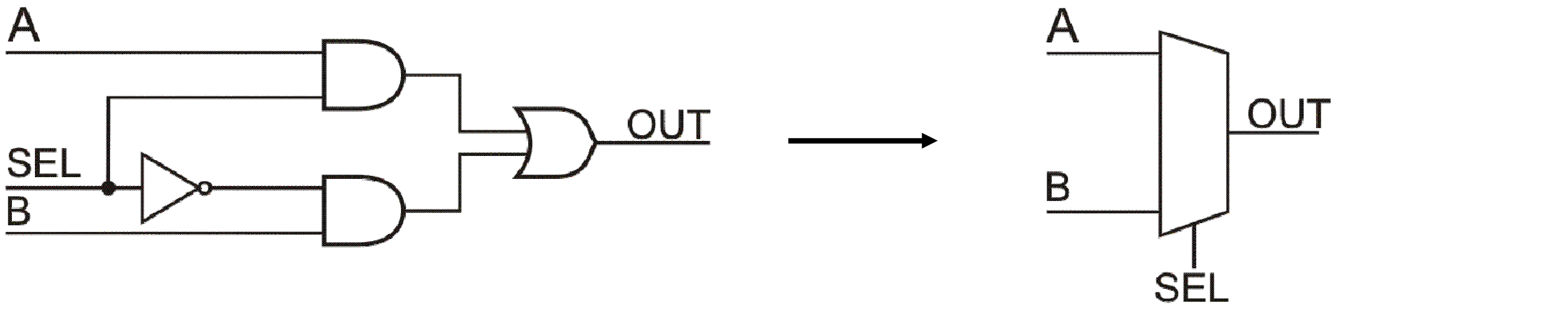
Connections by ordered list

```
module Top;
reg a, b, ci;
wire s, co;

ADD SUM1(a,b,ci,s,co);
...
endmodule
```

Example

```
module riddle(OUT, A, B, SEL);
//port declarations
input A,B,SEL;
output OUT;

//instances
not b1(sel_n, SEL);
and b2(sel_a, A, SEL);
and b3(sel_b, sel_n, B);
or b4(OUT, sel_a, sel_b);

endmodule
```

```
module mux(OUT, A, B, SEL);
//port declarations
input A,B,SEL;
output OUT;

assign OUT= (SEL) ? A : B;

endmodule
```



Lexical conventions

- Verilog is case-sensitive
- Keywords are in lower case
- Expressions (except for keywords ending in an expression like *endmodule*) are finished by semicolon (;)

- Identifiers (gives a unique names)
  - Can include letters (a-z, A-Z), digits (0-9), underline character (_), dollar character ($)
  - Must begin with letters or underline
  - Can have up to 1024 characters

- Comments
  - One-line comment begins with „//”
    ```
    //Comment
    ```
  - Muli-line comment: begins with „/*”, ends with „*/”
    ```
    /* Comment A
       comment B */
    ```

Lexical conventions – numbers

- Value system

Value	Condition
0	Logic low
1	Logic high
z	High-impedance state
x	Unknown value

- Number representation
  `<sign><size> ’ <base><value>`
  - `size` – the number of bits (decimal)
  - `base` – podstawa systemu liczenia
    - `‘b (‘B)` – binary
    - `‘o (‘O)` – octal
    - `‘d (‘D)` – decimal
    - `‘h (‘F)` – hexadecimal
  - `value` – digits: *0-f* depending on the `base`

- The default numbers – 32-bits
  `[‘ <base>]<value>`

Lexical conventions – numbers

- Examples

```
4’b1010                //4-bit binary
12’habc                //12-bit hexadecimal
12’b1111_0000_1011     //12-bit binary; _ increasing readability
16’d255                //16-bit decimal
12’h13x                //12-bit hexadecimal(12’b0001_0011_xxxx)
6’hx                   //6-bit hexadecimal
-6’d3                  //6-bit U2 decimal
6’d-4                  //error!!!
4’b10??                //? can replace z; 4’b10zz
‘hc3                   //hexadecimal 32-bit!!!
2530                   //decimal 32-bit!!!
```

Data types – wires and registers

- Wires
  - Represent physical connections between elements
  - Require continuous assignments

```
wire a,b; //nets (wires) a and b
wire c = 1’b0; //wire wit constant value
```

- Registers
  - Represent memory element (similar to variables in C)
  - Store the value until reassigned
  - Are used in procedural assignments

```
reg CLR; //example of declaration
```

Data types – vectors

- Wires and registers can be declared as vectors:

  [a:b] where: *a>b* or *a<b* or *a=b*, but *a* is always MSB

```
wire a; //1-bit (scalar)
wire [7:0] DBUS; //8-bit vector DBUS
wire [3:0] A,B; //4-bit wectors A and B
reg CLK; //scalar
reg [0:40] AD;  //41-bit register AD
```

- It is possible to select a single bit from the vector:

```
DBUS[3] //bit 3 of the DBUS
AD[0:3] //4 most signifficant bits of AD

DATA[15-:4] /*starting bit 15, 4 bits with lower indices
DATA[15:12]*/

DATA[8+:4] /*starting bit 8, 4 bits with higher indices
DATA[11:8]*/
```

Data types – arrays

- Arrays are allowed for `reg` and `wire` (scalar or vectors)
- Multi-dimensional arrays can also be declared with any number of dimensions
- Daclaration `<basic_type> <name>[a:b];`
- Elements of arrays `<name>[<index>]`
- Examples

```
integer count[0:7]; //8-element array
time chk_point [1:100]; //100-element (used in simulation)
reg bool[31:0]; //32-element 1-bit
reg [3:0] port [7:0]; //8-element array of 4-bit each
wire [15:0] wekt_s [7:0]; //8-el. 16-bit
integer tabl [0:9] [255:0]; //two-dimmensional
```

- Elements of arrays

```
count[3]=0;
tabl[8][6]=24356; // two-dimmensional array
port[5]=0;
port[5][1]=1’b1; //bit 1; element 5
```

Data types – memories

- Memories are modeled as a one-dimensional array of registers
- Each word can be one or more bits

- Examples

```
reg kbit[0:1023]; //1kb memory: 1x1024
reg [7:0] kB[0:1023]; //8kb memory: 8x1024
```

- Elements of memory

```
kB[7]  //element 7 (index) of the memory
```

Data types – strings

- A string is a sequence of characters that are enclosed by double quotes
- Must be contained on a single line
- Strings are treated as a sequence of one-byte ASCII values

```
reg [8*12:1] s;
    ...
    s = “Ala ma kota”;
```

Parameters

- Parameters enable constant defining

```
parameter IDLE  = 3’b000,
          START = 3’b001; //states of FSM

parameter W = 8;        //the bus width
parameter CNTMAX = 10; //max range
parameter PI = 3.141;  //float

parameter A = 32;
parameter B = 8;
parameter AB = (A+B)/2; //Determined as an expr.
```

Parameters

```
      module ADD(A,B,S,Co);
      parameter W = 4;
      input [W-1:0] A,B;
      output [W-1:0] S;
      output Co;
         assign {Co,S} = A + B;
      endmodule
```

- How to change the parameter
  - **defparm – can occur anywhere in the project (not recommended - low readability of the code)**

```
module ADD_TEST;
defparm S1.W = 8;          //redefinition W = 8 in the S1 instance
...
ADD S1(.A(A), .B(B), .S(S), .Co(Co));
...
endmodule
```

  - **By means of instatiating – defining a value when creating an instance**

```
ADD #(8)S1(.A(A), .B(B), .S(S), .Co(Co));  //redefinition W=8 by
                                              ordered list

ADD #(.W(8))S1(.A(A), .B(B), .S(S), .Co(Co)); //W=8 by name
```

- How to prepare the initial value of the number with bit size

```
A = W’d5                    //error
A = {W{1‘b1}}                       //A=1..111
A = {{W-3{1'b0}},3'd5};             //A=0..101
```

Operators

- Arithmetics
  - `A + B   //addition`
  - `A - B   //subtraction`
  - `A * B   //multiplication`
  - `D / E   //division` ★
  - `D % E   //modulus` ★
  - `**      //exponentiation` ★

- Bitwise

  `// X = 4'b1010, Y = 4'b1101`
  - `~X       //bitwise negation; 4'b0101`
  - `X & Y    //bitwiese and; 4'b1000`
  - `X | Y    //bitwise or; 4'b1111`
  - `X ^ Y    //bitwise EXOR; 4'b0111`

- Logical

  `// A=2, B=0`
  - `!A //logical negation; 0 (false)`
  - `!B //logical negation; 1 (true)`
  - `A && B  //logical and; 0 (false)`
  - `A || B  //logical or; 1 (true)`

```
//A=1, B=2
A & B //0
A && B //1 (true)
```

Operators

- RelationalS

```
 // A = 4, B = 3
 // X = 4'b1010, Y = 4'b1101, // Z = 4'b1xxz
 – A == B  //equality;             0
 – X != Y  //inequality;           1
 – X == Z  //equality;             x (unknown)
 – A <= B  //less than or equal;   0
 – A > B   //greater than;         1
 – Y >= X  //greater than or equal;1
 – Y < Z   //less than;            x (unknown)
```

- Shift

```
 // X = 4'b1100
 – Y = X >> 1  //right by 1 bit; 4'b0110
 – Y = X << 1  //left by 1 bit; 4'b1000
 – Y = X << 2  //left by 2 bits; 4'b0000
```

- Conditional: `<condition> ? <expr_1> : <expr_0>`

```
 assign ABUS = EN ? ADDR : 16'bz;  //tri-state buffer
 assign Y = SEL ? I1 : I0; //Mux 2 1
 assign Y = S1 ? (S0 ? I3 : I2):(S0 ? I1 : I0); //nested
```

Operators

- Reduction
  - Unary operator - executes a logical operation on all operand bits giving a single result bit
  - It can be used with ~ (negation)

```
// X = 4'b1010
– &X  //1 & 0 & 1 & 0 gives 1'b0
– |X  //1 | 0 | 1 | 0 gives 1'b1
– ^X  //1 ^ 0 ^ 1 ^ 0 gives 1'b0
– ~&X //NAND
– ~|X //NOR
– ~^X //EX-NOR
```

- Concatenation: `{<argument_1>,...,<argument_n>}`

```
// A = 1'b1, B = 2'b00, C = 2'b10
– Y = {B,C};         // Y = 4'b0010
– Y = {A,B,3'b101};  // Y = 6'b100101
– Y = {A,C[1],B[0]}; // Y = 3'b110
– Y = {A,A,A,A};
– Y = {4{A}};        // Y = 4’b1111
```

Continuous assignment – `assign`

- The new value to the output (left-hand-side) is assigned after every change of input (right-hand-side)
- L-value must be wire

- Examples

```
//out, i1 and i2 are of wire type
assign out = i1 & i2;

//implicated continuous assignment
wire out = i1 & i2;

//vectors
assign addr[15:0] = ad1[15:0] ^ ad2[15:0];

//concatenation
assign {Co,Sum[3:0]} = A[3:0] + B[3:0] + Ci;
```

to be continued ...

Based on:

Robert Czerwiński „Digital Circuits Design” lecture