

# Lecture Outline

## Digital VLSI Design

### Lecture 2: Verilog HDL

Semester A, 2018-19  
Lecturer: Dr. Adam Teman



Disclaimer: This course was prepared in its entirety by Adam Teman. Many materials were copied from sources freely available on the internet. Where possible, these sources have been cited, however, some references may have been cited incorrectly or overlooked. If you feel that a picture, graph, or code example has been copied from you and either needs to be cited or removed, please feel free to email [adamen@barilan.ac.il](mailto:adamen@barilan.ac.il) and I will address this as soon as possible.



## Verilog Syntax



## Simple Examples



## Verilog FSM Implementation

A simple 4-bit counter example



## Coding Style for RTL – Part 1



# Digital VLSI Design

## Lecture 2: Verilog HDL

Semester A, 2018-19

Lecturer: Dr. Adam Teman



Emerging Nanoscaled  
Integrated Circuits and Systems Labs



Bar-Ilan University  
אוניברסיטת בר-אילן

Disclaimer: This course was prepared, in its entirety, by Adam Teman. Many materials were copied from sources freely available on the internet. When possible, these sources have been cited; however, some references may have been cited incorrectly or overlooked. If you feel that a picture, graph, or code example has been copied from you and either needs to be cited or removed, please feel free to email [adam.teman@biu.ac.il](mailto:adam.teman@biu.ac.il) and I will address this as soon as possible.

# What is a hardware description language?

- **HDL is NOT another programming language:**

- Textual representation of *Hardware constructs*
- All statements are executed in *parallel*
- *Code ordering* is flexible.

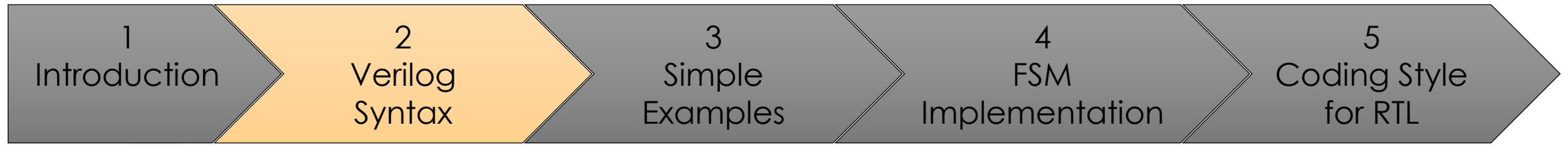
Example:

- $a=1;b=2;c=a+b \rightarrow c==3$
- $c=a+b;a=1;b=2 \rightarrow c==3$
- Execution of code is triggered by *Events*
- *Sensitivity lists* are used to define when a code section is executed
- **Different simulators may yield different results**
  - => Coding style is required

# Abstraction levels

- **Three coding styles:**
  - Structural code (GTL (Gate Level), Netlist)
  - RTL (Register Transfer Level)
  - Behavioral (Testbench)
- **DUT (Device Under Test)**
  - Represents Hardware
  - Usually RTL or GTL
- **Testbench**
  - Represents System
  - Usually Behavioral
  - Using higher order languages (“e”/SystemVerilog)

anics

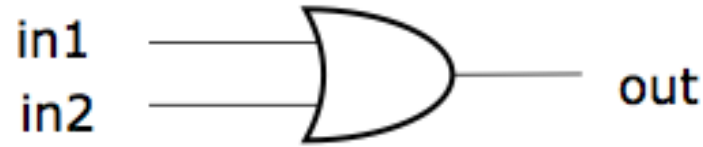


# Verilog Syntax

# Basic Constructs

- **Primitives:**

- not, and, or, etc.



```
or(out, in1, in2);
```

- **Signals:**

- 4 states: 0, 1, X, Z
- Wires: do not keep states
- Registers: keep states (i.e., outputs)
- Can represent buses or group of signals

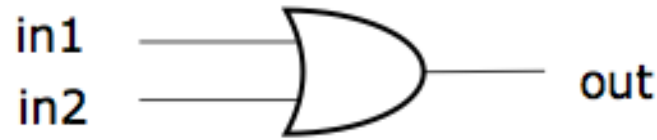
```
wire in1,in2;  
reg out;  
wire [7:0] data;  
reg [31:0] mem [0:7]; //width (bits)=32, depth (words)=8
```

Primitive name	Functionality
and	Logical And
or	Logical Or
not	Inverter
buf	Buffer
xor	Logical Exclusive Or
nand	Logical And Inverted
nor	Logical Or Inverted
xnor	Logical Exclusive Or Inverted

# Basic Constructs

- **Operators:**

- Similar to primitives
- `&`, `|`, `~`, `&&`, `||`, etc.



```
out = in1 | in2;
```

- **Constants:**

- The format is: `W' Bval`
- Examples:
  - `1'b0` – single bit binary 0 (or decimal 0)
  - `4'b0011` - 4 bit binary 0011 (or decimal 3)
  - `8'hff` = 8 bit hexadecimal ff (or decimal 255)
  - `8'd255` = 8 bit decimal 255

# Procedural Blocks

- **Initial block**

- Will be executed only once, at first time the unit is called (Only in testbench)

```
initial begin
    a = 1'b0;
    b = 1'b0;
end
```

- **Always block**

- Statements will be evaluated when a change in **sensitivity list** occurs
- **Example1** - sync reset, rising edge triggered flop:
- **Example 2** - async reset, rising edge triggered, load enable flop:

```
always @(posedge clock)
    if (!nreset)
        q <= 1'b0;
    else
        q <= d;
```

```
always @(posedge clock or negedge nreset)
    if (!nreset)
        q <= 1'b0;
    else if (load_enable)
        q <= d;
```



# Procedural Blocks

- There are two types of **always** blocks
- **Sequential**
  - Asserted by a **clock** in the **sensitivity list**.
  - Translates into flip-flops/latches.
- **Combinational**
  - Describes purely combinational logic, and therefore, the **sensitivity list** has (non-clock) **signals**.
  - The Verilog 2001 standard allows using **\*** instead of a sensitivity list to reduce bugs.

```
always @(posedge clock or negedge nreset)
    if (!nreset)
        q <= 1'b0;
    else if (load_enable)
        q <= d;
```

```
always @(a or b or c)
    out = a & b & c;
```

```
always @(*)
    out = a & b & c;
```

# Assignments

Verilog has three types of assignments:

- **Continuous assignment**
  - Outside of `always` blocks
- **Blocking procedural assignment “=”**
  - RHS is executed and assignment is completed before the next statement is executed.
- **Non-blocking procedural assignment “<= ”**
  - RHS is executed and assignment takes place at the end of the current time step (not clock cycle)
- **To eliminate mistakes, follow these rules:**

```
assign muxout = (sel&in1) | (~sel&in0);  
assign muxout = sel ? in1 : in0;
```

```
// assume initially a=1;  
a = 2;  
b = a;  
// a=2; b=2;
```

```
// assume initially a=1;  
a <= 2;  
b <= a;  
// a=2; b=1;
```

- Combinational `always` block: Use **blocking** assignments (=)
- Sequential `always` block: Use **non-blocking** assignments (<=)
- **Do not mix** blocking and non-blocking in the same `always` block
- **Do not assign to the same variable** from more than one `always` block

# Hierarchy

- **Modules**

- Used to define a hardware block

```
module mux4 (out, in, sel);  
  input [3:0] in;  
  input [1:0] sel;  
  output out;  
  reg out;  
  
  always @*  
    case (sel)  
      2'b00: out = in[0];  
      2'b01: out = in[1];  
      2'b10: out = in[2];  
      2'b11: out = in[3];  
      default: out = 1'bx;  
    endcase  
endmodule
```

Module Header

Module Body

- **Instances**

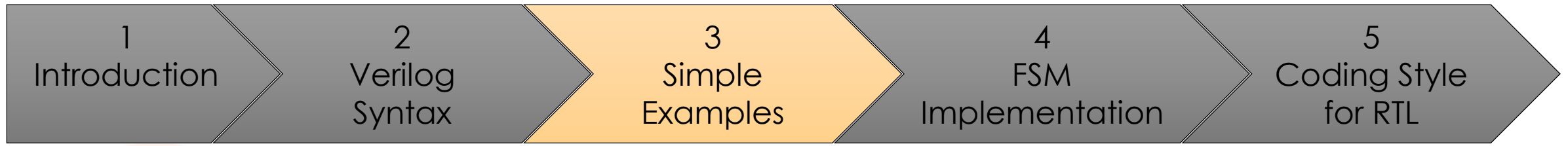
- Referencing a block at a different level

```
mux4 M0 (.out(outa),.in(a),.sel(sel));  
mux4 M1 (.out(outb),.in(b),.sel(sel));
```

# System Tasks

- System tasks are used to provide interface to simulation data
- Identified by a `$name` syntax
- Printing tasks:
  - `$display`, `$strobe`: Print once the statement is executed
  - `$monitor`: Print every time there is a change in one of the parameters
  - All take the “c” style `printf` format

```
$display("At %t Value of out is %b\n",$time,out);
```



# Simple Examples

# Hello World

- Your first Verilog module:

```
module main;  
  initial  
    begin  
      $display("Hello world!");  
      $finish;  
    end  
endmodule
```

# Combinatorial Logic

- Three ways to make a Mux

- Using an **assign** Statement:

```
wire out;  
assign out = sel ? a : b;
```

- Using an **always** Block:

```
reg out;  
always @ (a or b or sel)  
    if (sel)  
        out=a;  
    else  
        out=b;
```

- Using a **case** statement:

```
reg out;  
always @ (a or b or sel)  
    begin  
        case (sel)  
            1'b0: out=b;  
            1'b1: out=a;  
        endcase  
    end
```

# Sequential Logic

- A simple D-Flip Flop:

```
reg q;  
always @(posedge clk)  
    q<= d;
```

- An asynch reset D-Flip Flop:

```
reg q;  
always @(posedge clk or negedge reset_)  
    if (~reset_)  
        q<= 0;  
    else  
        q<= d;
```

- Be careful not to infer latches!!!:

```
reg q;  
always @(en)  
    if (en)  
        q<= d;
```



# Arithmetic

- Verilog supports standard arithmetic operators:
  - `+`, `-`, `*`, `<<` (shift left), `>>` (shift right), etc.
  - Be careful about division... (not synthesizable!)
  - Concatenate signals with the `{,}` operator
- But...
  - By default, Verilog treats all vectors as **unsigned** binary numbers.
  - To do signed (two's complement) operations, declare the reg/wire as **signed**:
  - To make a constant signed, add an **s**: `10'sh37C`

```
assign a = 4'b1100;  
assign b = 4'b1010;  
assign c = {a,b}; //c=8'b11001010
```

```
wire signed [9:0] a,b;  
wire signed [19:0] result = a*b;
```

# reg vs. wire

- **Oh no... Don't go there!**

- A **reg** is not necessarily an actual register, but rather a “driving signal”... (huh?)
- This is truly the most ridiculous thing in Verilog...
- But, the compiler will complain, so here is what you have to remember:

1. Inside **always** blocks (both sequential and combinational) only **reg** can be used as LHS.
2. For an **assign** statement, only **wire** can be used as LHS.
3. Inside an **initial** block (Testbench) only **reg** can be used on the LHS.
4. The **output** of an instantiated module can only connect to a **wire**.
5. **Inputs** of a **module** cannot be a **reg**.

```
reg r;  
always @*  
    r = a & b;
```

```
wire w;  
assign w = a & b;
```

```
reg r;  
initial  
    begin  
        r = 1'b0;  
        #1  
        r = 1'b1;  
    end
```

```
module m1 (out)  
    output out;  
endmodule  
  
reg r;  
m1 m1_instance(.out(r));
```

```
module m2 (in)  
    input in;  
    reg in;  
endmodule
```

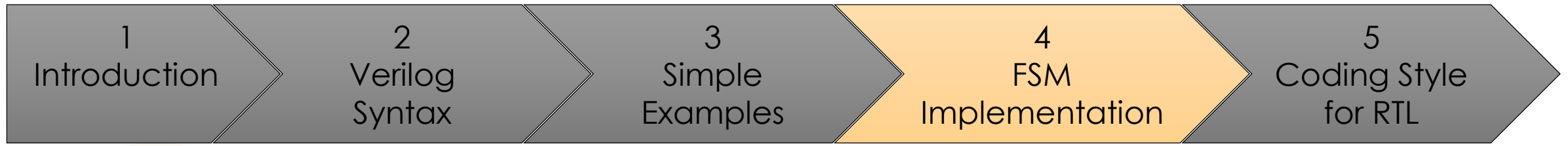
# Testbench constructs

- Create a clock:

```
`define CLK_PERIOD 10

initial
  begin //begins executing at time 0
    clk = 0;
  end

always //begins executing at time 0 and never stops
  #(CLK_PERIOD/2) clk = ~clk;
```



# Verilog FSM Implementation

A simple 4-bit counter example

# FSM Example

- **A 4-bit counter**

- Receives 4 inputs:

- `clk` – the system clock
    - `rst_n` – an active low reset
    - `act` – the activate signal
    - `up_dwn_n` – count up (positive) or count down (negative)

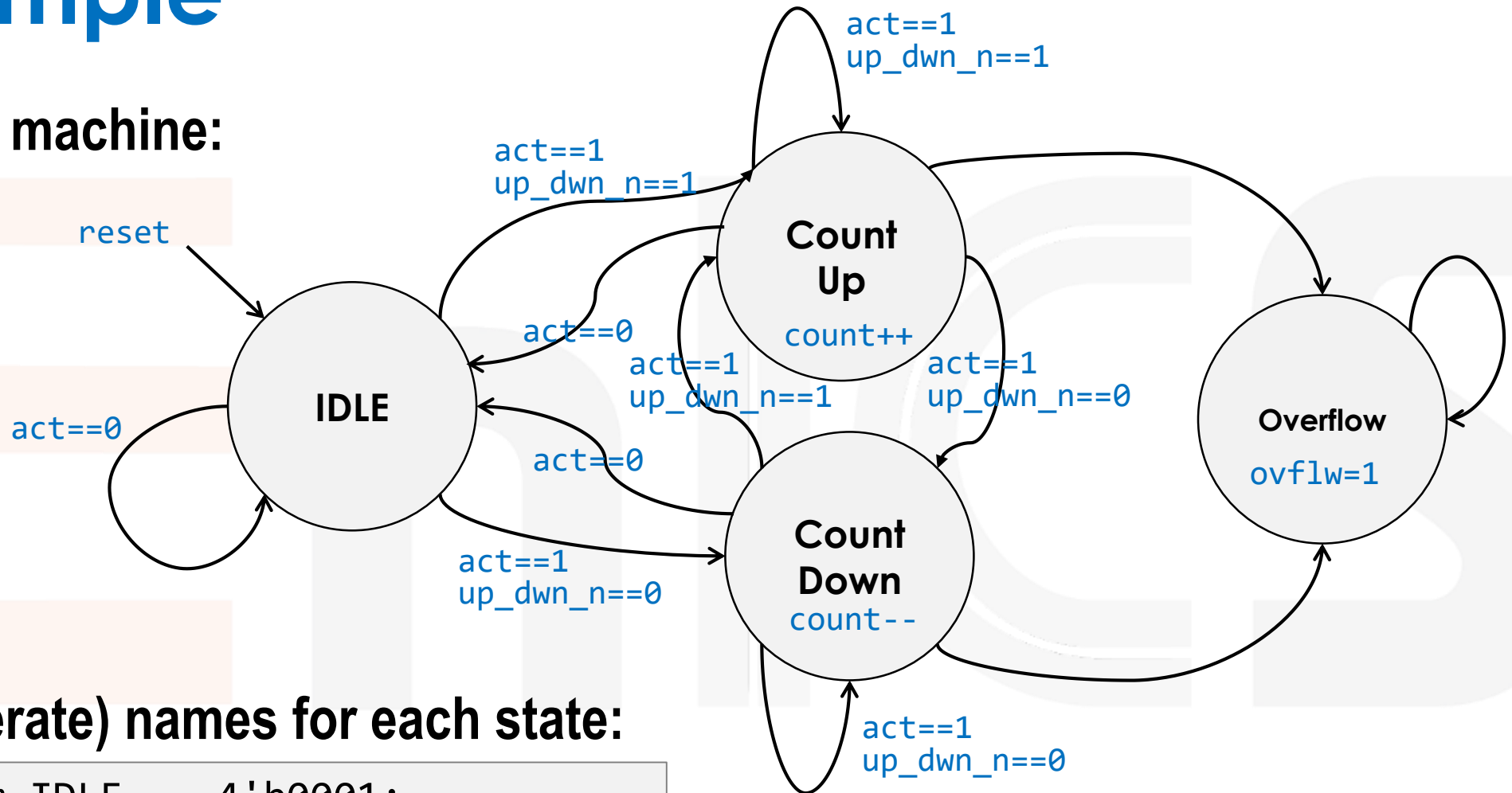
- Outputs 2 signals:

- `count`: the current counted value
    - `ovflw`: an overflow signal

```
module sm
    #(parameter COUNTER_WIDTH = 4)
    (clk,rst_n,act,up_dwn_n,count,ovflw);
    input clk;
    input rst_n;
    input act;
    input up_dwn_n;
    output [COUNTER_WIDTH-1:0] count;
    reg [COUNTER_WIDTH-1:0] count;
    output ovflw;
    reg ovflw;
    reg [3:0] state, next_state;
```

# FSM Example

- Draw the state machine:



- Define (Enumerate) names for each state:

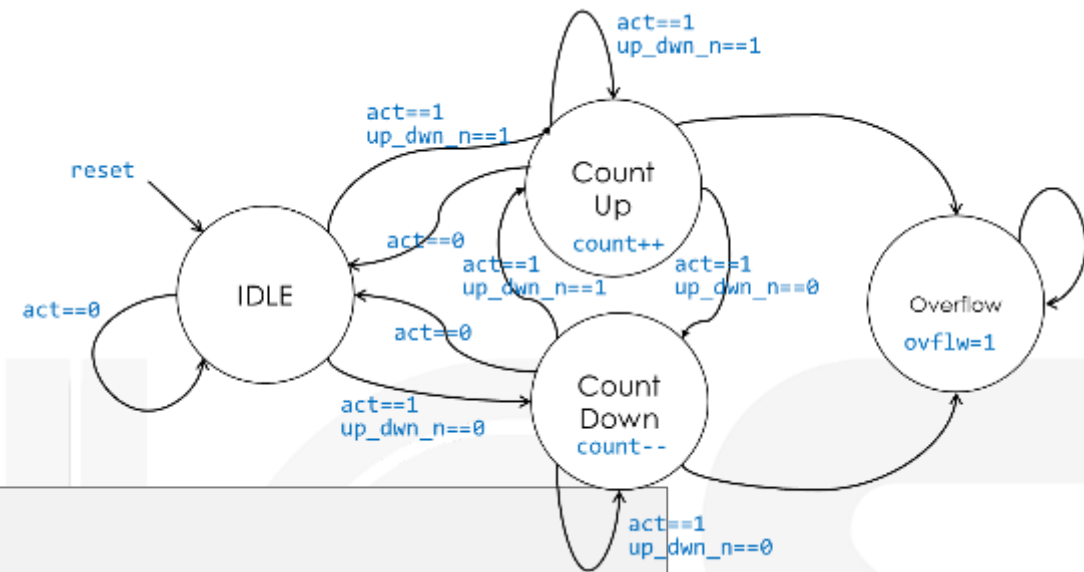
```
localparam IDLE   = 4'b0001;
localparam CNTUP  = 4'b0010;
localparam CNTDN  = 4'b0100;
localparam OVFLW  = 4'b1000;
```

# FSM Example

- **Combinational block**
  - Compute the next state:

```
always @*
  case (state)
    IDLE: begin
      if (act)
        if (up_dwn_n)
          next_state = CNTUP;
        else
          next_state = CNTDN;
      else
        next_state = IDLE;
    end
```

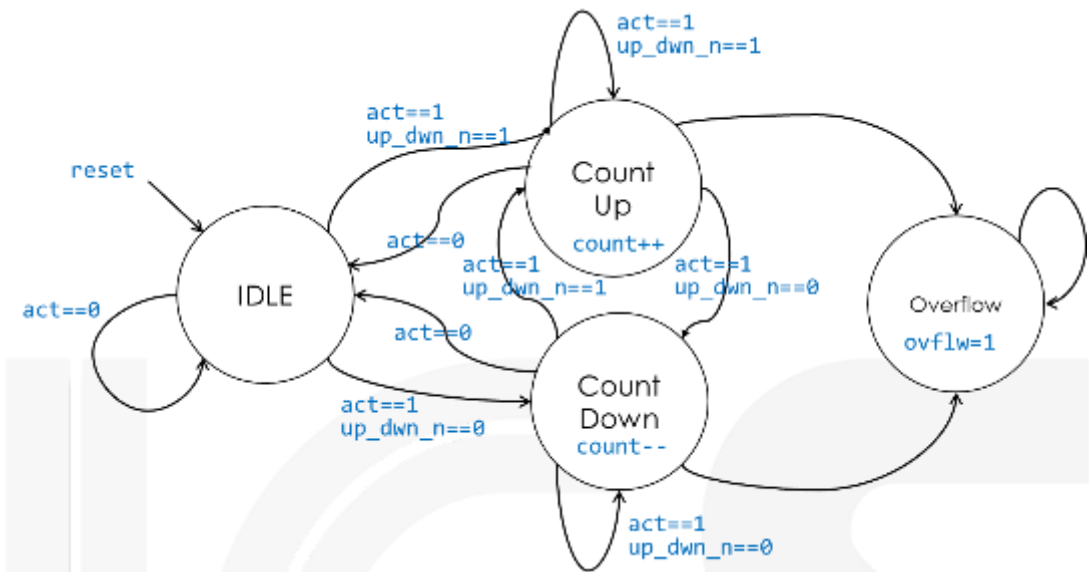
```
CNTUP: begin
  if (act)
    if (up_dwn_n)
      if (count==(1<<COUNTER_WIDTH)-1)
        next_state = OVFLW;
      else
        next_state = CNTUP;
    else
      if (count=='b0)
        next_state=OVFLW;
      else
        next_state=CNTDN;
  else
    next_state=IDLE;
end
```



# FSM Example

- **Combinational block**
  - compute the next state:

```
CNTDN: begin
  if (act)
    if (up_dwn_n)
      if (count==(1<<COUNTER_WIDTH)-1)
        next_state = OVFLW;
      else
        next_state = CNTUP;
    else
      if (count== 'b0)
        next_state=OVFLW;
      else
        next_state=CNTDN;
    else
      next_state=IDLE;
  end
end
```



```
OVFLW: begin
  next_state=OVFLW;
end

default: begin
  next_state = 'bx;
  $display("%t: State machine not
            initialized\n",$time);
end
endcase
```



# FSM Example

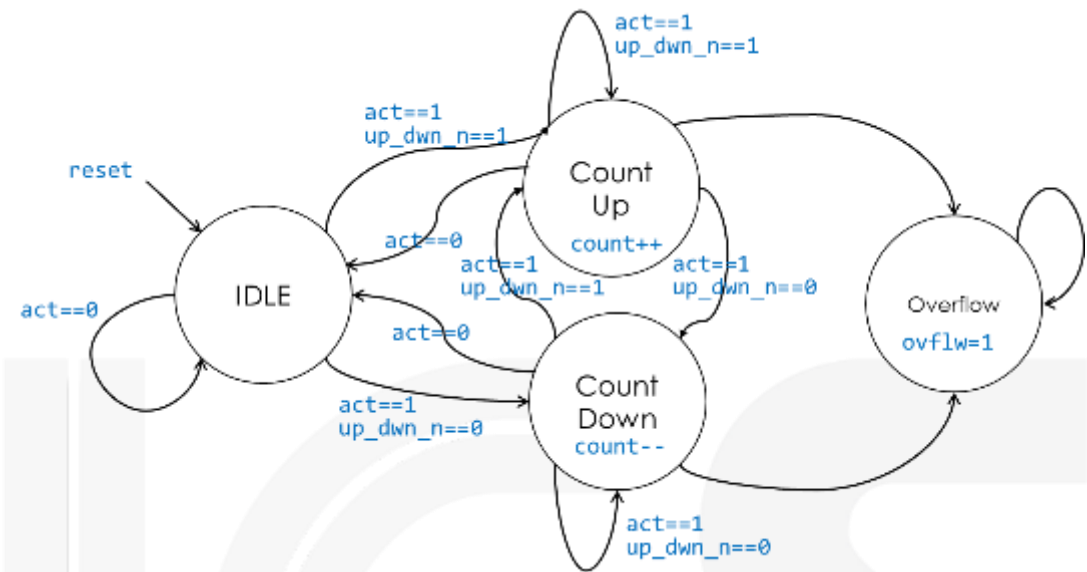
- **Sequential block**

- Define the state registers:

```
always @(posedge clk or negedge rst_n)
  if (!rst_n)
    state <= IDLE;
  else
    state <= next_state;
```

- Define the counter registers:

```
always @(posedge clk or negedge rst_n)
  if (!rst_n)
    count <= 'b0;
  else
    if (state==CNTUP)
      count <= count+1'b1;
    else if (state==CNTDN)
      count <= count-1'b1;
```



- Finally assign the output (Moore):

```
assign ovflw = (state==OVFLW) ? 1'b1 : 1'b0;

endmodule
```

# Testbench Example

- **Definition of signals and parameters**

```
module sm_tb;  
    parameter WIDTH = 5;  
    reg clk;  
    reg rst_n;  
    reg act;  
    reg up_dwn_n;  
    wire [WIDTH-1:0] count;  
    wire ovflw;
```

- **Instantiate the state machine:**

```
sm #(WIDTH) DUT (.clk(clk),.rst_n(rst_n),  
                .act(act),.up_dwn_n(up_dwn_n),  
                .count( count),.ovflw(ovflw));
```

ICS

# Testbench Example

- Set initial values, value monitoring and reset sequence:

```
initial begin
  clk = 1'b1;
  rst_n = 1'b0; // Activate reset
  act = 1'b0;
  up_dwn_n = 1'b1;

  // Monitor changes
  $monitor("%t: rst_n=%b act=%b up_dwn_n=%b count=%d\n", $time, rst_n, act, up_dwn_n, count, ovflw);
  // After 100 time steps, release reset
  #100 rst_n = 1'b1;
end
```

- Define a clock:

```
always
  #5 clk = ~clk;
```

- Set stimuli:

```
initial begin
  // @100, Start counting up
  //      until overflow
  #100 act = 1'b1;
        up_dwn_n = 1'b1;
  // Reset (10 cycles pulse)
  #1000 rst_n = 1'b0;
        act = 1'b0;
  #100 rst_n = 1'b1;
  // Do a count-up to 4 and
  //      then count-down to ovflw
  #100 act = 1'b1;
        up_dwn_n = 1'b1;
  #40 up_dwn_n = 1'b0;
end
endmodule
```

# The Chip Hall of Fame

- We started with the 4004, but Intel's real success came with

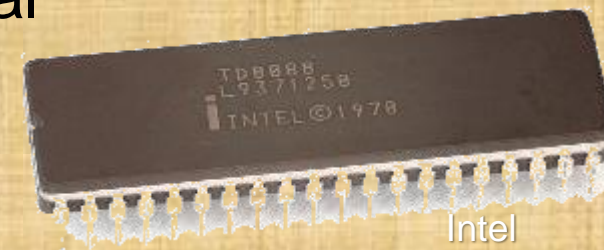
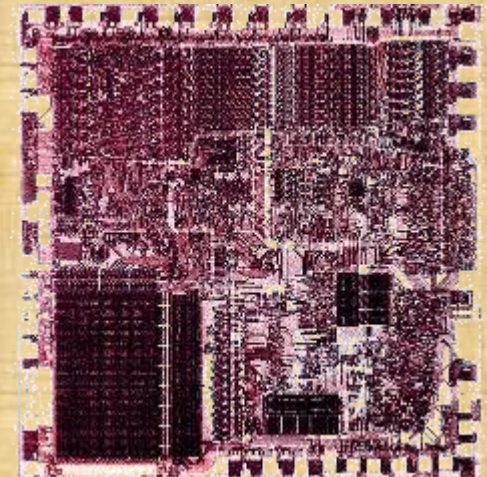
## The Intel 8088 Microprocessor

- The chip that established the x86 architecture, by being chosen to power the IBM PC.
- Release date: July 1979
- Transistor Count: 29,000      Process: 3 um CMOS
- Frequency: 10 MHz      16-bit x86 architecture
- A “castrated” version of the 8086, since it had an 8-bit external bus, instead of 16-bits.
- And guess what... it was designed in [Haifa](#)!

2018 Inductee to the IEEE Chip Hall of Fame



The IBM PC



Intel

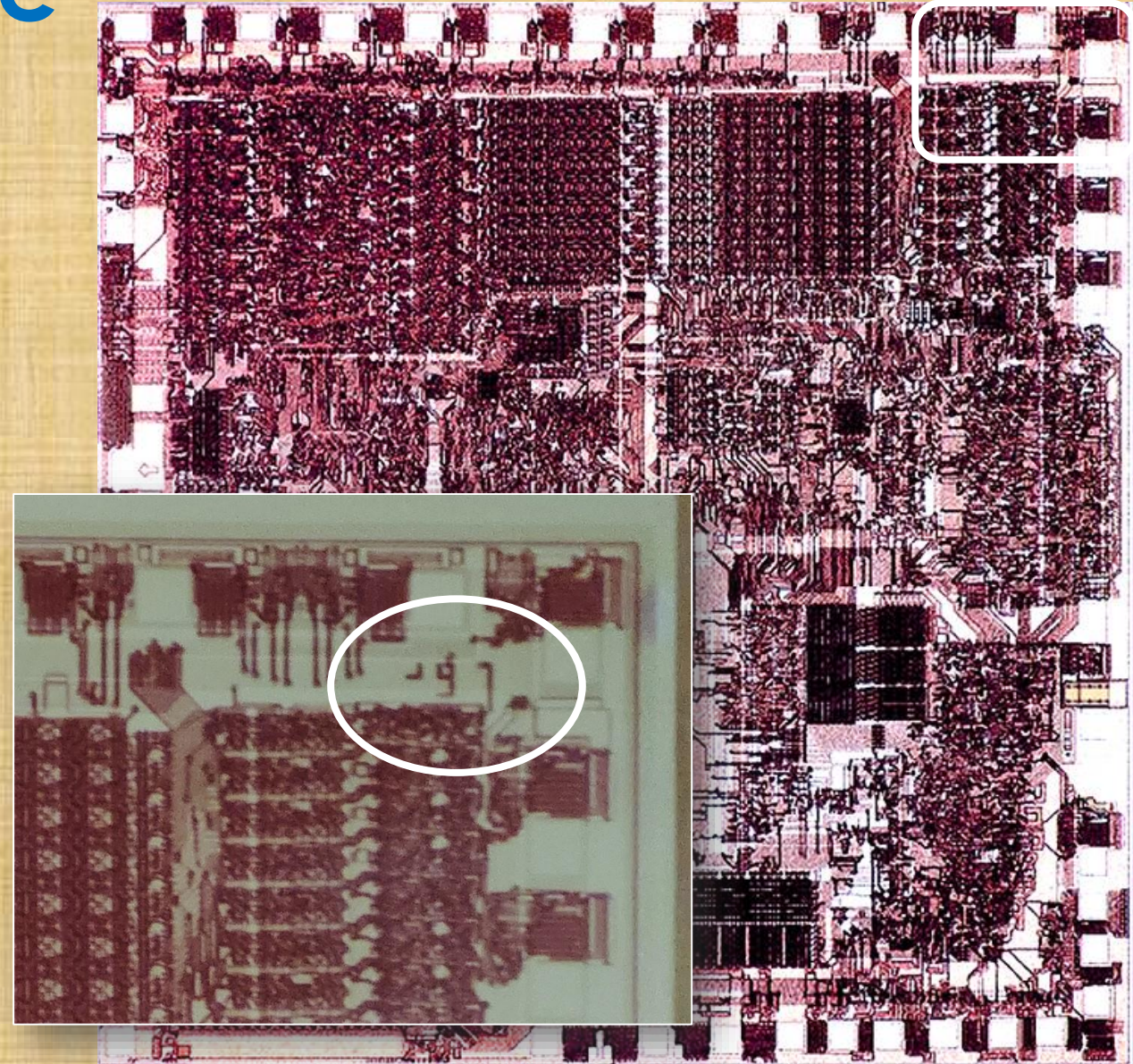


# The Chip Hall of Fame

- Actually, being designed in Haifa, I know something about this chip that Gordon Moore may not have known...
- Let's zoom in on the top right corner.
- Is that Hebrew?

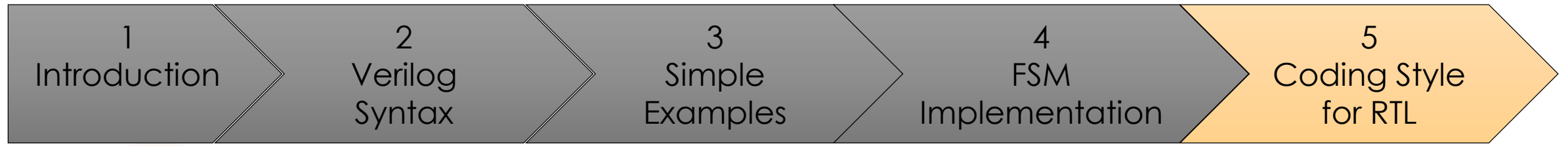
**Mr. Rafi Retter**

*Designer of the Intel 8080*



2018 Inductee to the IEEE Chip Hall of Fame





# Coding Style for RTL – Part 1

# HDL is NOT another programming language!

- **Well, at least it shouldn't be...**
  - Verilog is a relatively “rich” programming language, with commands and constructs that let you do many things.
  - In fact, it was originally designed exclusively as a verification language.
  - However, when designing hardware,  
you cannot actually do whatever you want!
  - Therefore, it is important to follow some simple (but strict!) rules  
and adhere to a *coding style*.
- **In the following slides:**
  - I will introduce you to *a few* guidelines and rules.
  - *But*, we will revisit this later, after you have some hands-on experience.

# Organizing your code

- **Each module should be in a separate file**
  - Name the file `<modulename>.v`
  - Always connect modules by name (the `.dot()` form).
- **Write each input/output on a separate line**
  - Comment what each signal is used for.
- **Separate sequential and combinational logic**

```
module fsm(...)
  input ... ;
  ...
  always@(posedge clk or negedge rst_)
    ... // sequential code
  always@*
    ... // combinational code
  assign ... // simple combinational logic
endmodule
```

**fsm.v**



# Assignment

Just to make sure you got the rules I mentioned before...

- **In a combinational (`always@*`) block:**
  - Always use blocking (`=`) assignment.
  - Recommended to use (`*`) in your sensitivity list.
  - Always use full case statements. Use default to propagate `X`.
- **In a sequential (`always@posedge`) block:**
  - Always use non-blocking (`<=`) assignment.
  - Prefer each flip-flop in a separate `always` block.
  - Prefer to data enable all sampling.
- **Never assign a signal (LHS) from more than one `always` block.**

# Be careful not to infer latches

- A very bad mistake made by rookie HDL designers is to describe latches by mistake.
- If an output is not explicitly assigned by every signal in the sensitivity list, a latch will be inferred to save the previous state of the signal.
- For example, what happens if `sel==11` ?
- The same will happen if an output is not assigned in all branches of an `if-else` block.
  - An `if` must have an `else` !!!
- Oh yeah... Stay away from `tri-states`, as well!

```
module mux4to1 (out, a, b, c, d, sel);  
    output out;  
    input a, b, c, d;  
    input [1:0] sel;  
    reg out;  
  
    always @(sel or a or b or c or d)  
        case (sel)  
            2'b00: out = a;  
            2'b01: out = b;  
            2'b10: out = d;  
            // 2'b11: out = c; // Missing branch!  
        endcase  
endmodule
```

# Stick with one reset type

- The purpose of reset is to bring your design into a well-known state.
  - It is desirable to have *every flip-flop* resettable, whether or not required.
  - We usually use asynchronous reset:

```
always @(posedge clk or negedge rst_)
    if (!rst_)
        state <= idle;
    else
        state <= next_state;
```
  - But synchronous reset is also okay:

```
always @(posedge clk)
    if (!rst_)
        state <= idle;
    else
        state <= next_state;
```
- Just make sure you don't mix them in your design!
- And do not put logic on your reset signal!

```
always @(posedge clk or negedge rst_)
    if (!rst_)
        state <= idle;
    else
        state <= next_state;
```

```
always @(posedge clk)
    if (!rst_)
        state <= idle;
    else
        state <= next_state;
```

```
assign something = a && reset ;

always@*
    case (state)
        1'b1011: if (b || reset )
                    next_state = idle;
```

# Parameterize your design

- “Pretty code” is code that is completely parametrized
- Two approaches to parameterization:
  - Compiler directives: ``define`, ``include`, and ``ifdef`
    - put all ``define` statements in external define files.
  - Parameters or `localparam`
    - `parameters` can be overridden through instantiation
    - `localparam` is better for constants, such as FSM encoding
- You can also use `generate` statements, but be careful of these.
- Always encode FSM states with hard coded values
  - You can choose various methods, such as binary, gray code, one-hot, etc.

# Write readable code

- **Always use indentation!!!**
  - You will lose points if you turn in ugly code!
- **Naming Conventions**
  - Really useful!
  - Common styles:
    - **C style**: words with underscores, e.g., `packet_addr`, `data_in`
    - **Pascal style**: capitalize every word, e.g., `PacketAddr`, `DataIn`
    - **Modula style**: capitalize every word but the first, e.g., `packetAddr`, `dataIn`
  - There is no “one right answer”, but two recommended styles are:
    - NetFPGA VerilogCodingGuidelines  
<https://github.com/NetFPGA/netfpga/wiki/VerilogCodingGuidelines>
    - ETH-Zurich VHDL naming conventions (with emacs highlighting!):  
<https://www.dz.ee.ethz.ch/en/information/hdl-help/vhdl-naming-conventions.html>

# Some helpful documents and references

- Chris Fletcher “**Verilog: wire vs. reg**”
- Greg Tumbush “**Signed Arithmetic in Verilog 2001 – Opportunities and Hazards**”
- NetFPGA wiki “**VerilogCodingGuidelines**”
- MIT 6.111 Lectures <http://web.mit.edu/6.111/www/f2007/>
- Stuart Sutherland, Don Mills “**Standard Gotchas: Subtleties in the Verilog and SystemVerilog Standards That Every Engineer Should Know**”