Tutorial – 1
(Intro to Processor Architecture)

Akshit Gureja
K Pavan Kumar

Overview

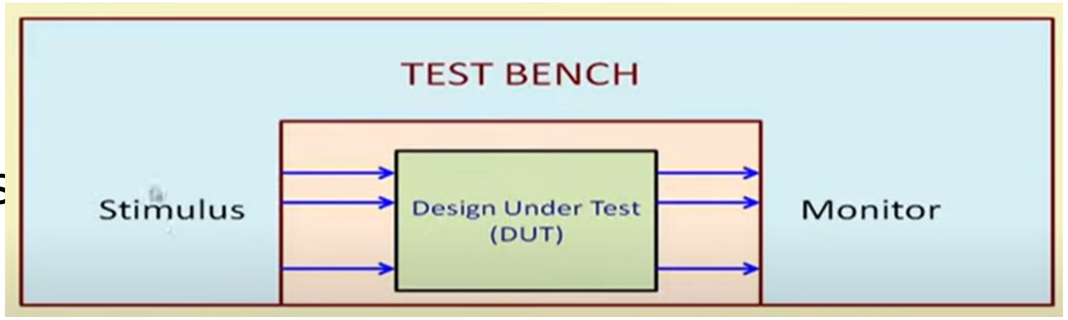
- Verilog Revision
- Sequential Circuits
- Y86-Architecture
- ALU
- Memory Modules
- Assignment

Short recap on Verilog

- Verilog is a hardware description language.
- Icarus Verilog is a command-line compiler for Verilog HDL.
- Testbench consists of the testing infrastructure and the test cases for testing code.
- GTKwave is a waveform visualizer tool that allows to examine the input and output signal stated across the simulation period.
- **Commands:**

➢ iverilog -o output example.v example_tes

➢ vvp output

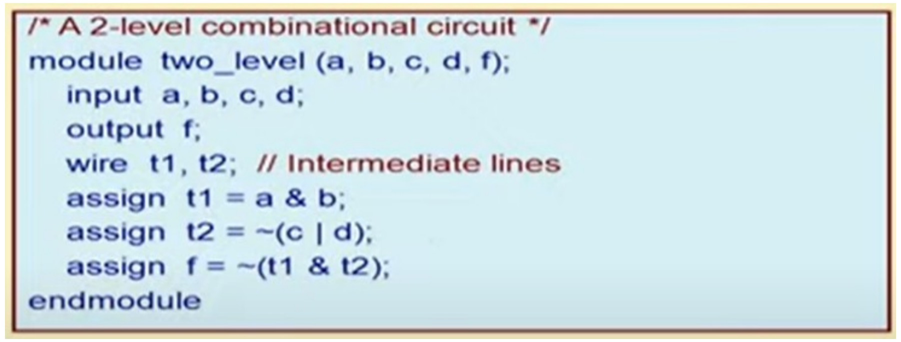


Continued…

- In Verilog, the basic unit of hardware is a module. A module cannot be defined within other modules but can be instantiated.

Basic Structure:

```
module module_name(list_of_ports);
    Input/output declaration;
    Local net declarations;
    Parallel statements;
Endmodule
```



```
/* A 2-level combinational circuit */
module  two_level (a, b, c, d, f);
   input  a, b, c, d;
   output  f;
   wire  t1, t2;  // Intermediate lines
   assign  t1 = a & b;
   assign  t2 = ~(c | d);
   assign  f = ~(t1 & t2);
endmodule
```

Behavioral vs Structural

- In structural verilog code, you describe the hardware In terms of primitives such as gates.
- In behavioral verilog code, you defined the behavior of the module in the form of a function

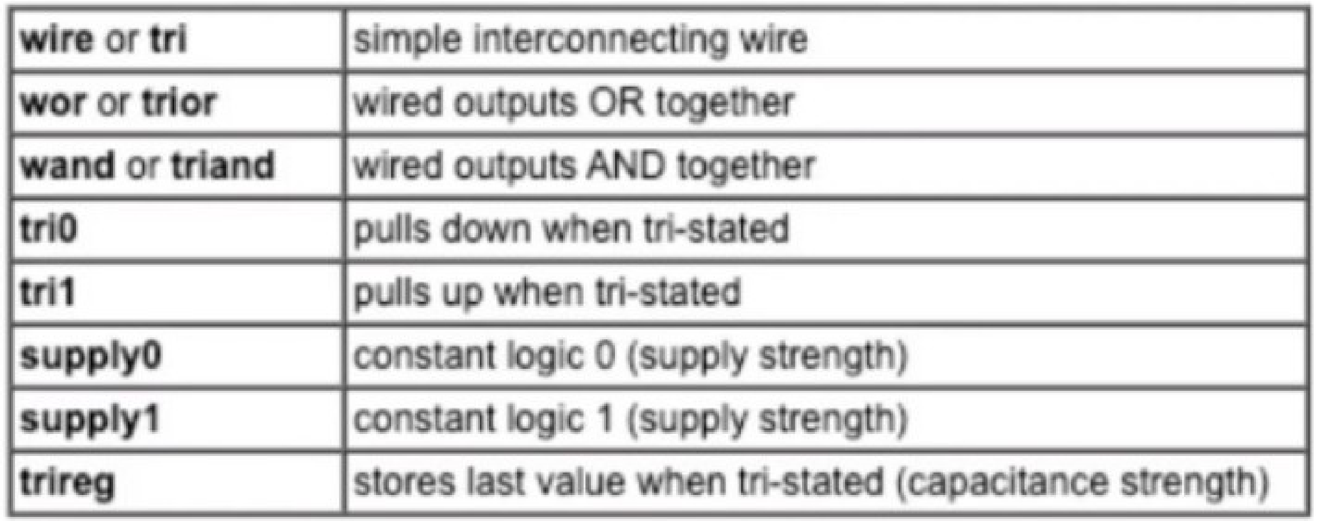
Note that "assign" statement represents continuous assignment, whereby the variable on LHS gets updated whenever the expression on RHS changes.

LHS must be a "net" type variable , RHS can be either a "net" or "reg" type.

Data types in Verilog

1. Net

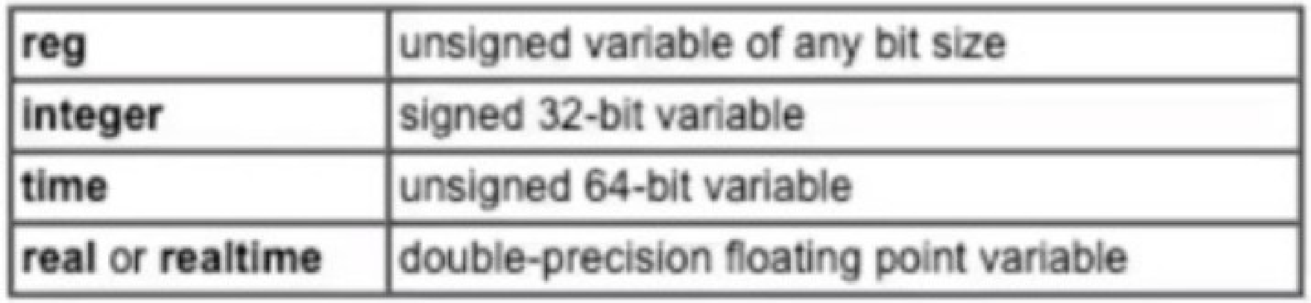
- Nets are continuously driven, cannot be used to store a value.
- Used to model connections between continuous assignments and instantiations.
- Default value of net data type is Z (High impedance value)



wire or **tri**	simple interconnecting wire
wor or **trior**	wired outputs OR together
wand or **triand**	wired outputs AND together
tri0	pulls down when tri-stated
tri1	pulls up when tri-stated
supply0	constant logic 0 (supply strength)
supply1	constant logic 1 (supply strength)
trireg	stores last value when tri-stated (capacitance strength)

2. Reg

- Retains the last value assigned to it, often used to represent a storage element, but sometimes it can translate to combinational circuits as well.
- It can be assigned a value in synchronism with clk signal or even otherwise.
- Default value of a "reg" data type is x(Unknown logic state)
- Often used in procedural assignments.



reg	unsigned variable of any bit size
integer	signed 32-bit variable
time	unsigned 64-bit variable
real or **realtime**	double-precision floating point variable

Vectors

"Net" or "Reg" type variables can be declared as vectors of multiple bitwidths.

Vectors are declared by specifying a range [range1:range2], where range1 is MSB and range2 is LSB.

Part of vectors can be addressed and used in an expression.

Example:

```
reg [31:0] IR;
reg [5:0] opcode;
Opcode = IR[31:26];
reg [63:0] register_banks[15:0];   // 16  64-bit registers
```

• Connectivity during instantiation

a. Positional association

The parameters of the module being instantiated are listed in same order as in original description.

b. **Explicit association**

The parameters of the module are listed in any order.

```
-------------------------------------------------------------------------------
module example(A,B,C,D,E,F,Y);
…
endmodule
--------------------------------------------------------------------------
----
module testbench;
reg X1, X2, X3, X4, X5, X6;  wire OUT;
…
example DUT(.OUT(Y), .X1(A), .X2(B), .X3(C), .X4(D), .X5(E), .X6(F))
endmodule
```

Parameters: A constant with a given value. If the size is not specified, it is taken to be 32-bit.

Parameter HI = 25, LO = 2'b01;

Primitive Gates:

Verilog provides a set of pre-defined logic gates.

Can be instantiated in a module to create a structured design.

The output signal is wire by default unless explicitly declared as register type..

The `timescale derivative

Often used in simulation, delay values in one module need to be specified in terms of some time unit.

`timescale<reference-time-unit> /<time – precision>

`timescale 10ns/1 ns

Procedural Assignment

Two kinds of procedural assignments are supported in Verilog.

1. The initial block

Statements under an "initial" start at time 0 and execute only once.

If there exists multiple blocks, all blocks will start to execute concurrently at time 0.

2. Always block

Used to model a block of activity that is repeated indefinitely in digital ckt.

```
always @(event_expression)
begin
.....
end
```

Note that only "reg" type variables can be assigned within "initial" or "always" block.
Any kind of variable can appear in event expression.

Blocking vs Non-blocking statement

- In verilog, one (or) more sequential statements can be present inside an "initial" or "always" block.
- Multiple assignment statements inside a " begin .. end" block may either execute sequentially (blocking) or con-currently (non-blocking)
- Blocking:

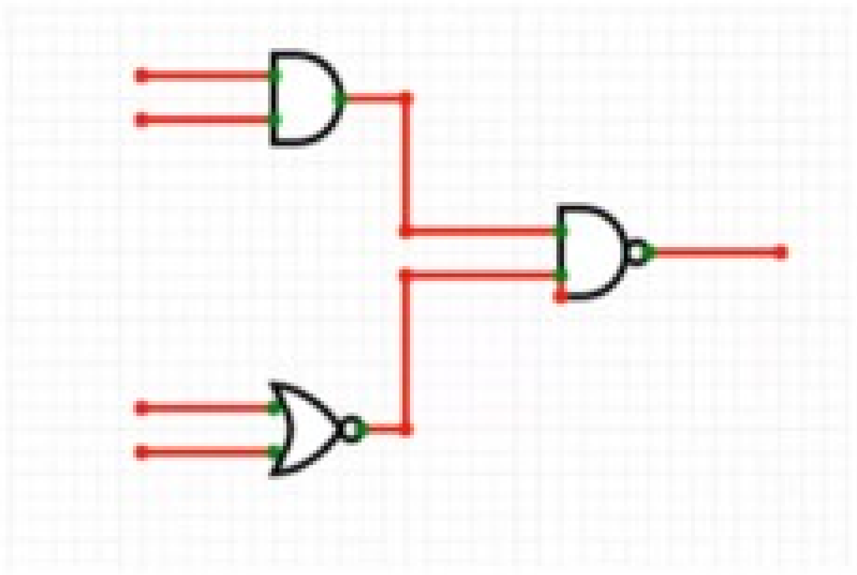
```
a = b + c;
z = x + y;
```

- Non-blocking:

```
a <= b + c;
z <= x + y;
```

Combinational Circuits

```
/* A 2-level combinational circuit */
module two_level (a, b, c, d, f);
    input a, b, c, d;
    output f;
    wire t1, t2;  // Intermediate lines assign
    t1 = a & b; assign t2 = ~(c | d);
    assign f = ~(t1 & t2);
endmodule
```

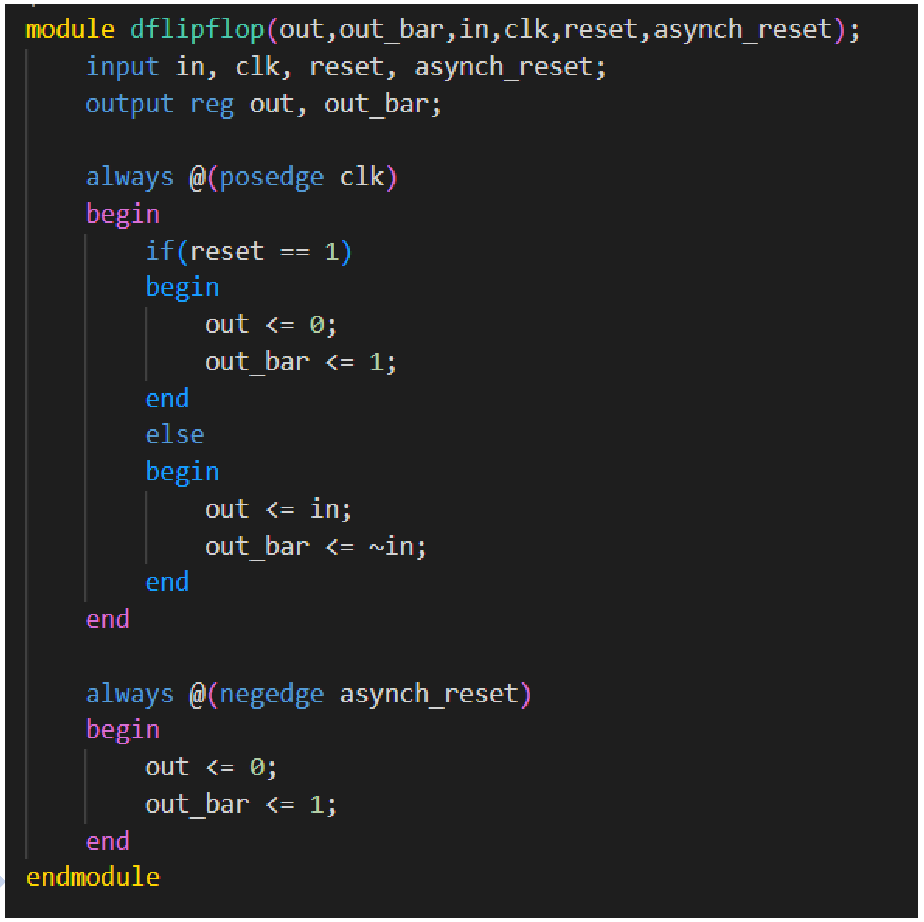


Inputs for the gates are
a,b,c,d respectively.

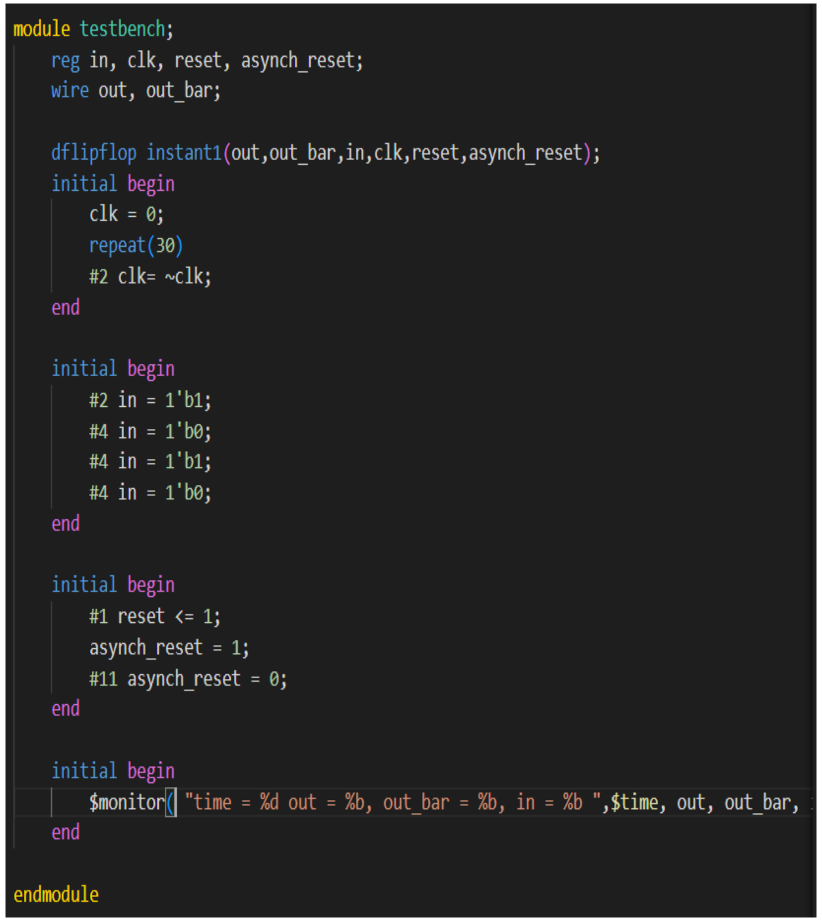
Sequential Circuits

- Many circuits of practical importance are required to have some sort of memory element
- This could be to store the output, store the input, feedback purposes etc.
- Combinational circuits cannot store any information and output varies every time the input is changed.
- This is why we need sequential circuits, which can be defined as "combinational circuits with feedback". The output of these circuits thus depends on the present as well as the past state inputs.

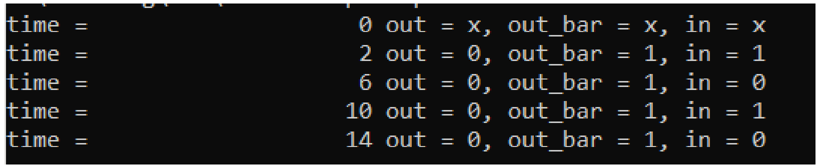
Sequential Circuits



```verilog
module dflipflop(out,out_bar,in,clk,reset,asynch_reset);
    input in, clk, reset, asynch_reset;
    output reg out, out_bar;

    always @(posedge clk)
    begin
        if(reset == 1)
        begin
            out <= 0;
            out_bar <= 1;
        end
        else
        begin
            out <= in;
            out_bar <= ~in;
        end
    end

    always @(negedge asynch_reset)
    begin
        out <= 0;
        out_bar <= 1;
    end
endmodule
```

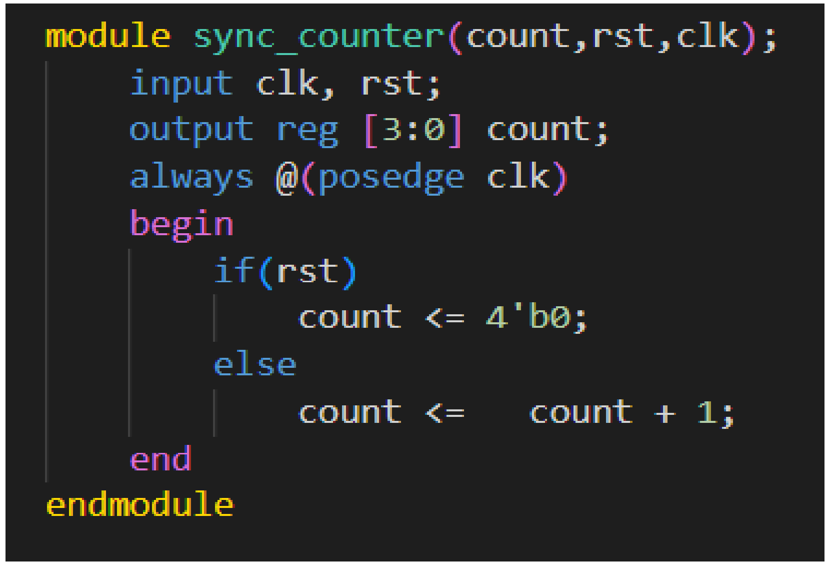


```verilog
module testbench;
    reg in, clk, reset, asynch_reset;
    wire out, out_bar;

    dflipflop instant1(out,out_bar,in,clk,reset,asynch_reset);
    initial begin
        clk = 0;
        repeat(30)
        #2 clk= ~clk;
    end

    initial begin
        #2 in = 1'b1;
        #4 in = 1'b0;
        #4 in = 1'b1;
        #4 in = 1'b0;
    end

    initial begin
        #1 reset <= 1;
        asynch_reset = 1;
        #11 asynch_reset = 0;
    end

    initial begin
        $monitor( "time = %d out = %b, out_bar = %b, in = %b ",$time, out, out_bar,
    end

endmodule
```

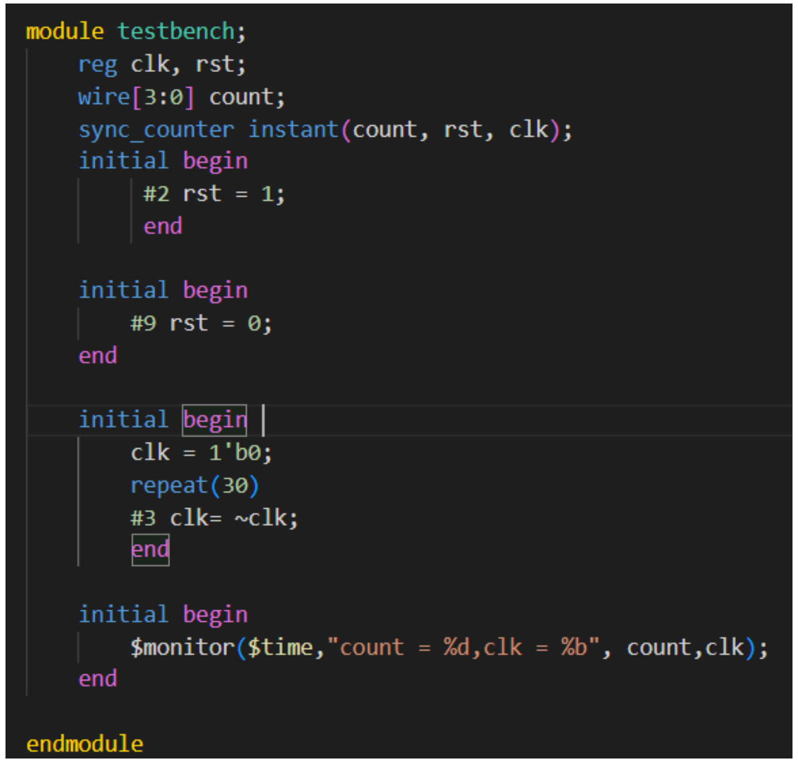


```
time =                    0 out = x, out_bar = x, in = x
time =                    2 out = 0, out_bar = 1, in = 1
time =                    6 out = 0, out_bar = 1, in = 0
time =                   10 out = 0, out_bar = 1, in = 1
time =                   14 out = 0, out_bar = 1, in = 0
```

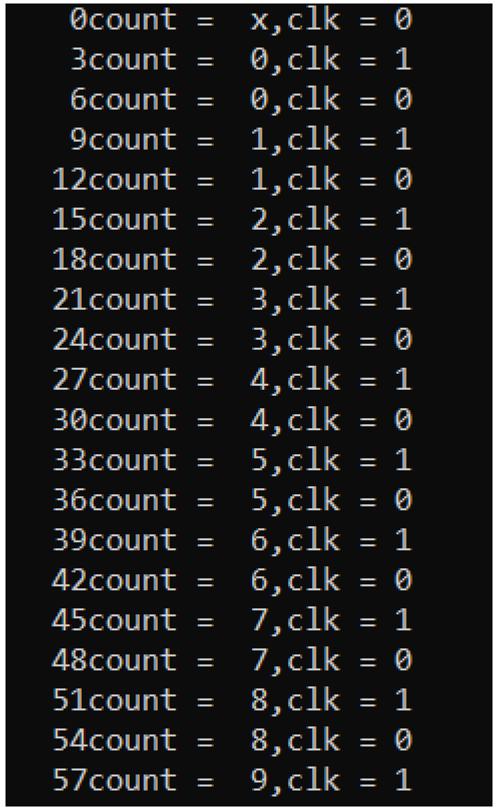
- **Synchronous Counter**



```verilog
module sync_counter(count,rst,clk);
    input clk, rst;
    output reg [3:0] count;
    always @(posedge clk)
    begin
        if(rst)
            count <= 4'b0;
        else
            count <=   count + 1;
    end
endmodule
```

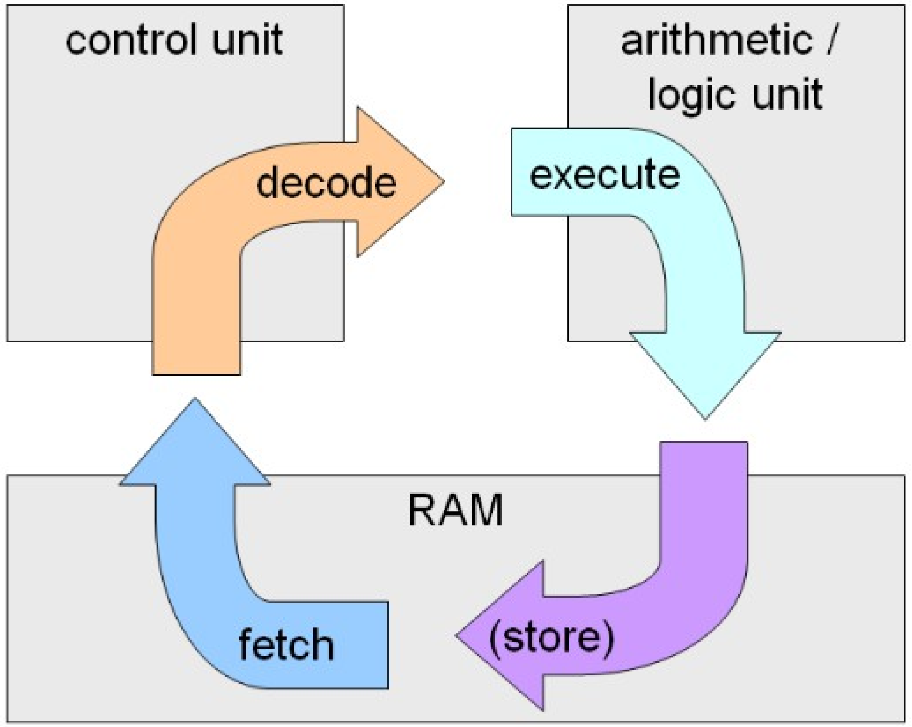


```verilog
module testbench;
    reg clk, rst;
    wire[3:0] count;
    sync_counter instant(count, rst, clk);
    initial begin
        #2 rst = 1;
        end

    initial begin
        #9 rst = 0;
    end

    initial begin
        clk = 1'b0;
        repeat(30)
        #3 clk= ~clk;
        end

    initial begin
        $monitor($time,"count = %d,clk = %b", count,clk);
    end

endmodule
```



```
   0count =  x,clk = 0
   3count =  0,clk = 1
   6count =  0,clk = 0
   9count =  1,clk = 1
  12count =  1,clk = 0
  15count =  2,clk = 1
  18count =  2,clk = 0
  21count =  3,clk = 1
  24count =  3,clk = 0
  27count =  4,clk = 1
  30count =  4,clk = 0
  33count =  5,clk = 1
  36count =  5,clk = 0
  39count =  6,clk = 1
  42count =  6,clk = 0
  45count =  7,clk = 1
  48count =  7,clk = 0
  51count =  8,clk = 1
  54count =  8,clk = 0
  57count =  9,clk = 1
```

How does a processor work?



How does a processor work?

There are essentially 6 stages of a processor

1. Fetch
2. Decode
3. Execute
4. Memory
5. Write Back
6. PC Update

Y86-Architecture

Inspired from the x86-64 instruction set and is largely a subset of the same.

Includes only 8-byte integer operations

Has fewer data types, instructions, operations and addressing modes (second index register and scaling are not supported)

A gentler introduction to assembly level programming than x86.

Each instruction in Y86-64 program can read and modify some part of the processor state.

Arithmetic Logic Unit (ALU)

- As evident from the name, Arithmetic and logical operations are to be performed.
- Arithmetic operations include, 64-bit addition, subtraction.
- Logical operations include 64-bit AND and XOR operations.

Addition:

- Consider 64-bit full adder to perform addition, the 64-bit full adder can be implemented using 64 1-bit full adders. So, instantiate 1-bit full adder 64 times!!
- Here, concept of **generate** block comes to rescue.
- Generate block is used when modules are to be instantiated multiple times.

- Example when multiple instances are to be made:

```
module circuitA(a,b,c,d,e);
…..
endmodule

 module mainmodule(circuit parameters);
……
genvar i;
generate
     for(i = 0; i < N; i = i + 1)
     begin
               circuitA  instance(parameters) ;
     end
….
endmodule
```

- Subtraction: Subtraction of two 64-bit numbers can be achieved as follows.

C = A – B

C = A + (-B)

C = A + 2’s complement of B

Blocks used for addition can be utilized.

- Implement AND and EX-OR accordingly.
- Inputs to ALU: Control signal, input1, input2
- Outputs: calculated value, overflow flag

2’s complement form:

- MSB is signed bit. 0 for positive numbers and 1 for negative numbers.

Representing a positive number:

The binary representation of a positive number is calculated by repeatedly dividing by 2. (The usual way of computing).

For example, consider representing +15 in 8-bit (In the project, we deal with 64-bit) 2’s complement form.

+15 => **0**0001111

Representing a negative number:

The binary representation of a negative number is 2’s complement of the corresponding positive number.

For example, -15 => (2’s complement of +15)

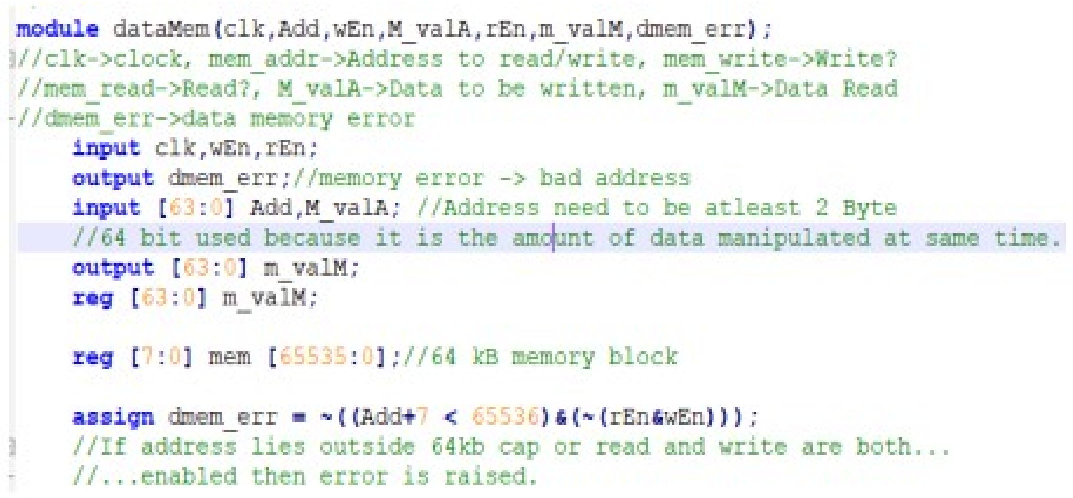
=> 11110000 + 1

=> **1**1110001

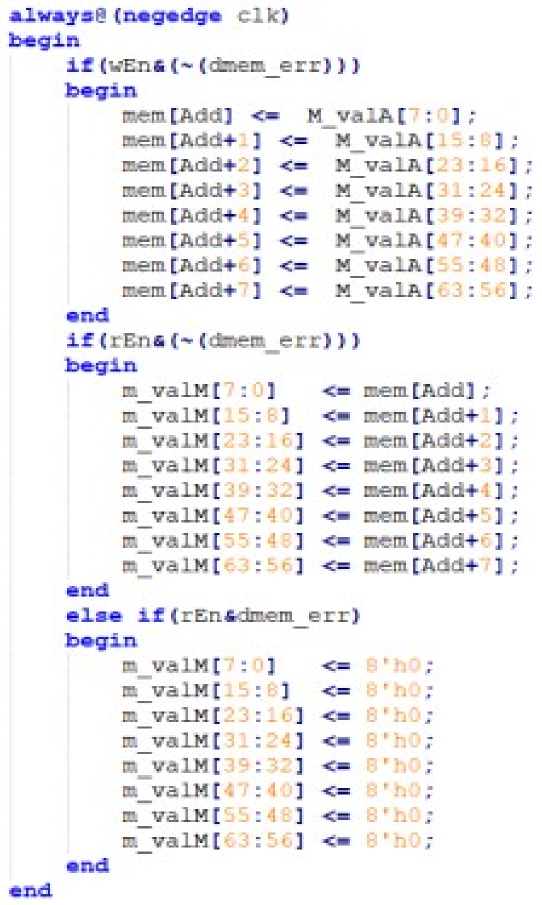
Memory Modules

- Our processor will need a memory module to store the given instructions and also to store the program data, maintain the stack, store addresses etc.
- Furthermore we need to decide if the memory can be overwritten or not. We wouldn't want to overwrite the instructions and data provided at the start of program execution.
- This leads us to begin thinking in terms of RAM (Random Access Memory) and ROM (Read-Only Memory)
- But which memory is exactly used for storing the instructions ???

Implementation of Random Access Memory



```
module dataMem(clk,Add,wEn,M_valA,rEn,m_valM,dmem_err);
//clk->clock, mem_addr->Address to read/write, mem_write->Write?
//mem_read->Read?, M_valA->Data to be written, m_valM->Data Read
//dmem_err->data memory error
    input clk,wEn,rEn;
    output dmem_err;//memory error -> bad address
    input [63:0] Add,M_valA; //Address need to be atleast 2 Byte
    //64 bit used because it is the amount of data manipulated at same time.
    output [63:0] m_valM;
    reg [63:0] m_valM;

    reg [7:0] mem [65535:0];//64 kB memory block

    assign dmem_err = ~((Add+7 < 65536)&(~(rEn&wEn)));
    //If address lies outside 64kb cap or read and write are both...
    //...enabled then error is raised.
```



```
always@(negedge clk)
begin
    if(wEn&(~(dmem_err)))
    begin
        mem[Add]  <=  M_valA[7:0];
        mem[Add+1] <=  M_valA[15:8];
        mem[Add+2] <=  M_valA[23:16];
        mem[Add+3] <=  M_valA[31:24];
        mem[Add+4] <=  M_valA[39:32];
        mem[Add+5] <=  M_valA[47:40];
        mem[Add+6] <=  M_valA[55:48];
        mem[Add+7] <=  M_valA[63:56];
    end
    if(rEn&(~(dmem_err)))
    begin
        m_valM[7:0]   <= mem[Add];
        m_valM[15:8]  <= mem[Add+1];
        m_valM[23:16] <= mem[Add+2];
        m_valM[31:24] <= mem[Add+3];
        m_valM[39:32] <= mem[Add+4];
        m_valM[47:40] <= mem[Add+5];
        m_valM[55:48] <= mem[Add+6];
        m_valM[63:56] <= mem[Add+7];
    end
    else if(rEn&dmem_err)
    begin
        m_valM[7:0]   <= 8'h0;
        m_valM[15:8]  <= 8'h0;
        m_valM[23:16] <= 8'h0;
        m_valM[31:24] <= 8'h0;
        m_valM[39:32] <= 8'h0;
        m_valM[47:40] <= 8'h0;
        m_valM[55:48] <= 8'h0;
        m_valM[63:56] <= 8'h0;
    end
end
```

Assignment –1 (Build a 64-bit ALU in Verilog)

You are expected to build an ALU with the following functionality

- ADD – 64 bits
- SUB – 64 bits
- AND – 64 bits
- XOR – 64 bits

You are not allowed to use +, -, &, ^ directly on the 64-bit inputs for the given 64-bit operations. We expect you all to write each of the above modules from scratch in Verilog (structural).

All input and outputs should be signed and use 2's complement for the subtraction.

Write a final wrapper ALU unit from where you will call the modules mentioned above based on the control input. The ALU unit takes as input the control signal, and two 64-bit inputs, and returns the 64-bit output corresponding to the control signal chosen.

Some further instructions

- Assignment has to be done in teams of 2 (The same teams will be continued for the project)
- Github Classrooms will be set up

Files to be submitted:

- Verilog module for each operation
- Verilog testbench for each operation – try to test for all or a considerable number of input combinations
- Verilog module for the wrapper ALU unit
- Verilog testbench for the wrapper ALU unit
- A report summarizing your approach and results. Clearly mention the control inputs to the ALU and their corresponding functionality, and the tests you performed in the testbenches. Results should contain the screenshots of the waveforms for each operation on 64-bit inputs.

Do not indulge in any malpractices or plagiarism. Any such malpractice will have serious consequences.

Any Questions ?