

ET5080E

Digital Design Using Verilog HDL

Fall '21

Handy Testbench Constructs:

- ✓ while, repeat, forever loops
- ✓ Parallel blocks (fork/join)
- ✓ Named blocks (disabling of blocks)
- ✓ File I/O
- ✓ Functions & Tasks

Administrative Matters

■ Readings

- Chapter 7 (last part, if you haven't read already)
- Chapter 8 (tasks & functions)

Loops in Verilog

- We already saw the **for** loop:

```
reg [15:0] rf[0:15];          // memory structure for modeling register file
reg [5:0] w_addr;             // address to write to

for (w_addr=0; w_addr<16; w_addr=w_addr+1)
    rf[w_addr[3:0]] = 16'h0000;    // initialize register file memory
```

- There are 3 other loops available:
 - While loops
 - Repeat loop
 - Forever loop

while loops

- Executes until boolean condition is not true
 - If boolean expression false from beginning it will never execute loop

```
reg [15:0] flag;  
reg [4:0] index;  
  
initial begin  
    index=0;  
    found=1'b0;  
    while ((index<16) && (!found)) begin  
        if (flag[index]) found = 1'b1;  
        else index = index + 1;  
    end  
    if (!found) $display("non-zero flag bit not found!");  
    else $display("non-zero flag bit found in position %d",index);  
end
```

Handy for cases where
loop termination is a
more complex function.

Like a search

repeat Loop

- Good for a fixed number of iterations
 - Repeat count can be a variable but...
 - ✓ It is only evaluated when the loops starts
 - ✓ If it changes during loop execution it won't change the number of iterations
- Used in conjunction with `@(posedge clk)` it forms a handy & succinct way to wait in testbenches for a fixed number of clocks

initial begin

```
inc_DAC = 1'b1;  
repeat(4095) @(posedge clk); // bring DAC right up to point of rollover  
inc_DAC = 1'b0;  
inc_smpl = 1'b1;  
repeat(7)@(posedge clk);      // bring sample count up to 7  
inc_smpl = 1'b0;
```

end

forever loops

- We got a glimpse of this already with clock generation in testbenches.
- Only a **\$stop**, **\$finish** or a specific **disable** can end a **forever** loop.

initial begin

clk = 0;

forever #10 clk = ~ clk;
end

Clock generator is by far the most common use of a forever loop

Sequential vs Parallel → (**begin/end**) vs (**fork/join**)

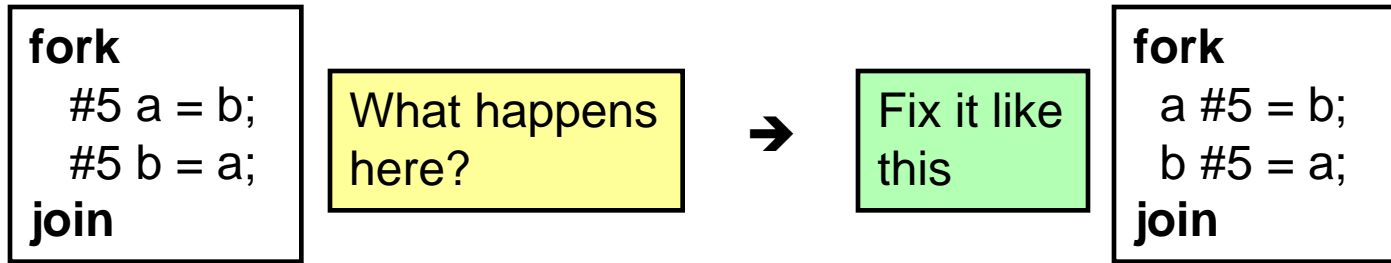
- **begin/end** are used to form compound sequential statements. We have seen this used many times.
- **fork/join** are used to form compound parallel statements.
 - Statements in a parallel block are executed simultaneously
 - All delay or event based control is relative to when the block was entered

Can be useful when you want to wait for the occurrence of 2 events before flow passes on, but you don't know the order the 2 events will occur

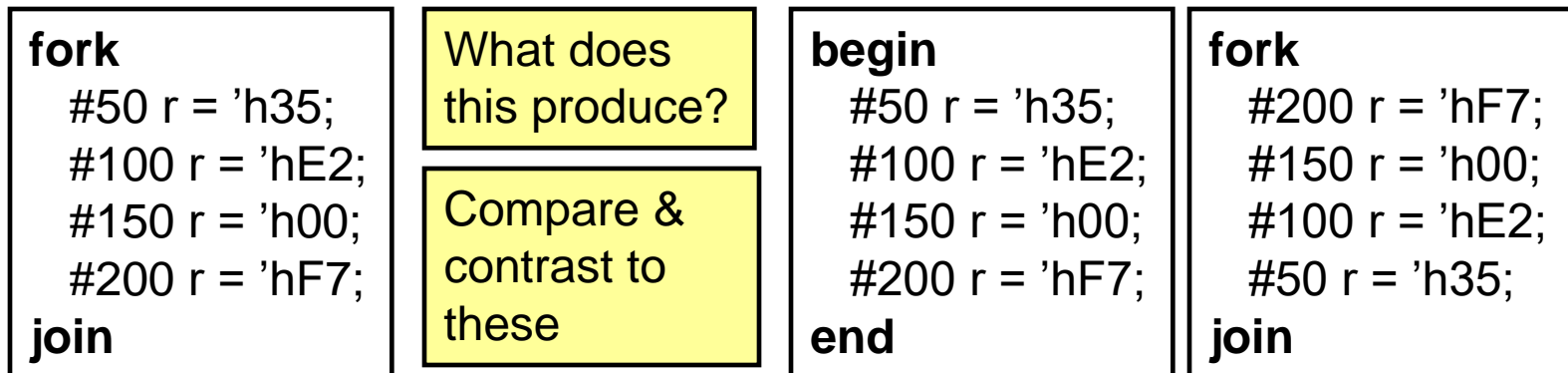
```
begin  
  fork  
    @Aevent  
    @Bevent  
  join  
    areg = breg;  
end
```

fork / join (continued)

- Can be a source of races



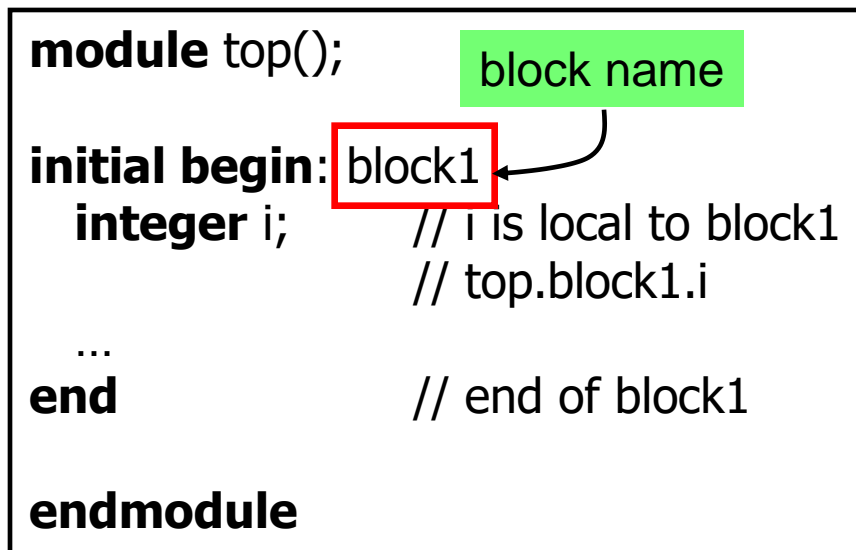
Intra-assignment timing control works because the intra-assignment delay causes the values of a and b to be evaluated *before* the delay, and the assignments to be made *after* the delay



Named Blocks

- Blocks (**begin/end**) or (**fork/join**) can be named
 - Local variables can be declared for the named block
 - Variables in a named block can be accessed using hierarchical naming reference
 - Named blocks can be disabled (i.e. execution stopped)

```
module top();  
  
initial begin: block1  
    integer i;           // i is local to block1  
                        // top.block1.i  
    ...  
end                  // end of block1  
  
endmodule
```



disable Statement

- Similar to the “break” statement in C
 - Disables execution of the current block (not permanently)

```
begin : break
  for (i = 0; i < n; i = i+1) begin : continue
    @(posedge clk)
    if (a == 0) // "continue" loop
      disable continue;
    if (a == b) // "break" from loop
      disable break;
    statement1
    statement2
  end
end
```

What occurs if (a==0)?

What occurs if (a==b)?

How do they differ?

File I/O – Why?

- If we don't want to hard-code all information in the testbench, we can use input files
- Help automate testing
 - One file with inputs
 - One file with expected outputs
- Can have a software program generate data
 - Create the inputs for testing
 - Create “correct” output values for testing
 - Can use files to “connect” hardware/software system

Opening/Closing Files

- **\$fopen** opens a file and returns an integer descriptor
 - integer fd = \$fopen(“filename”);**
 - integer fd = \$fopen(“filename”, r);**
 - If file cannot be open, returns a 0
 - Can output to more than one file simultaneously by writing to the OR (|) of the relevant file descriptors
 - ✓ Easier to have “summary” and “detailed” results
- **\$fclose** closes the file
 - \$fclose(fd);**

Writing To Files

- Output statements have file equivalents
 - ✓ **\$fmonitor()**
 - ✓ **\$fdisplay()**
 - ✓ **\$fstrobe()**
 - ✓ **\$fwrite()** // write is like a display without the \n
- These system calls take the file descriptor as the first argument
 - ✓ **\$fdisplay**(fd, "out=%b in=%b", out, in);

Reading From Files

- Read a binary file: **\$fread**(destination, fd);
 - Can specify start address & number of locations too
 - Good luck! I have never used this.
- Very rich file manipulation (see IEEE Standard)
 - ✓ \$fseek(), \$fflush(), \$ftell(), \$rewind(), ...
- Will cover a few of the more common read commands next

Using **\$fgetc** to read characters

```
module file_read()
parameter EOF = -1;
integer file_handle,error,indx;
reg signed [15:0] wide_char;
reg [7:0] mem[0:255];
reg [639:0] err_str;

initial begin
  indx=0;
  file_handle = $fopen("text.txt","r");
  error = $ferror(file_handle,err_str);
  if (error==0) begin
    wide_char = 16'h0000;
    while (wide_char!=EOF) begin
      wide_char = $fgetc(file_handle);
      mem[indx] = wide_char[7:0];
      $write("%c",mem[indx]);
```

```
      indx = indx + 1;
    end
  end
  else $display("Can't open file...");
  $fclose(file_handle);
end
endmodule
```

The quick brown fox jumped
over the lazy dogs

text.txt

When finished the array *mem* will contain the characters of this file one by one, and the file will have been echoed to the screen.

Why wide_char[15:0] and why signed?

Using **\$fgets** to read lines

```
module file_read2()

integer file_handle,error,indx,num_bytes_in_line;
reg [256*8:1] mem[0:255],line_buffer;
reg [639:0] err_str;

initial begin
    indx=0;
    file_handle = $fopen("text2.txt","r");
    error = $ferror(file_handle,err_str);
    if (error==0) begin
        num_bytes_in_line = $fgets(line_buffer,file_handle);
        while (num_bytes_in_line>0) begin
            mem[indx] = line_buffer;
            $write("%s",mem[indx]);
            indx = indx + 1;
            num_bytes_in_line = $fgets(line_buffer,file_handle);
        end
    end
    else $display("Could not open file text2.txt");
```

\$fgets() returns the number of bytes in the line. When this is a zero you know you hit EOF.

Using **\$fscanf** to read files

```
module file_read3()
integer file_handle,error,indx,num_matches;
reg [15:0] mem[0:255][1:0];
reg [639:0] err_str;

initial begin
    indx=0;
    file_handle = $fopen("text3.txt","r");
    error = $ferror(file_handle,err_str);
    if (error==0) begin
        num_matches = $fscanf(file_handle,"%h %h",mem[indx][0],mem[indx][1]);
        while (num_matches>0) begin
            $display("data is: %h %h",mem[indx][0],mem[indx][1]);
            indx = indx + 1;
            num_matches = $fscanf(file_handle,"%h %h",mem[indx][0],mem[indx][1]);
        end
    end
    else $display("Could not open file text3.txt");
```

12f3	13f3
abcd	1234
3214	21ab
text3.txt	

Loading Memory Data From Files

- This is very useful (memory modeling & testbenches)
 - \$readmemb("<file_name>",<memory>);
 - \$readmemb("<file_name>",<memory>,<start_addr>,<finish_addr>);
 - \$readmemh("<file_name>",<memory>);
 - \$readmemh("<file_name>",<memory>,<start_addr>,<finish_addr>);
- **\$readmemh** → Hex data...**\$readmemb** → binary data
 - But they are reading ASCII files either way (just how numbers are represented)

// addr	data
@0000	10100010
@0001	10111001
@0002	00100011

example "binary" file

// addr	data
@0000	A2
@0001	B9
@0002	23

example "hex" file

//data
A2
B9
23

address is optional for the lazy

Example of \$readmemh

```
module rom(input clk; input [7:0] addr; output [15:0] dout);

reg [15:0] mem[0:255];    // 16-bit wide 256 entry ROM
reg [15:0] dout;

initial
    $readmemh("constants",mem);

always @(negedge clk) begin
    //////////////////////////////////////
    // ROM presents data on clock low //
    //////////////////////////////////////
    dout <= mem[addr];
end

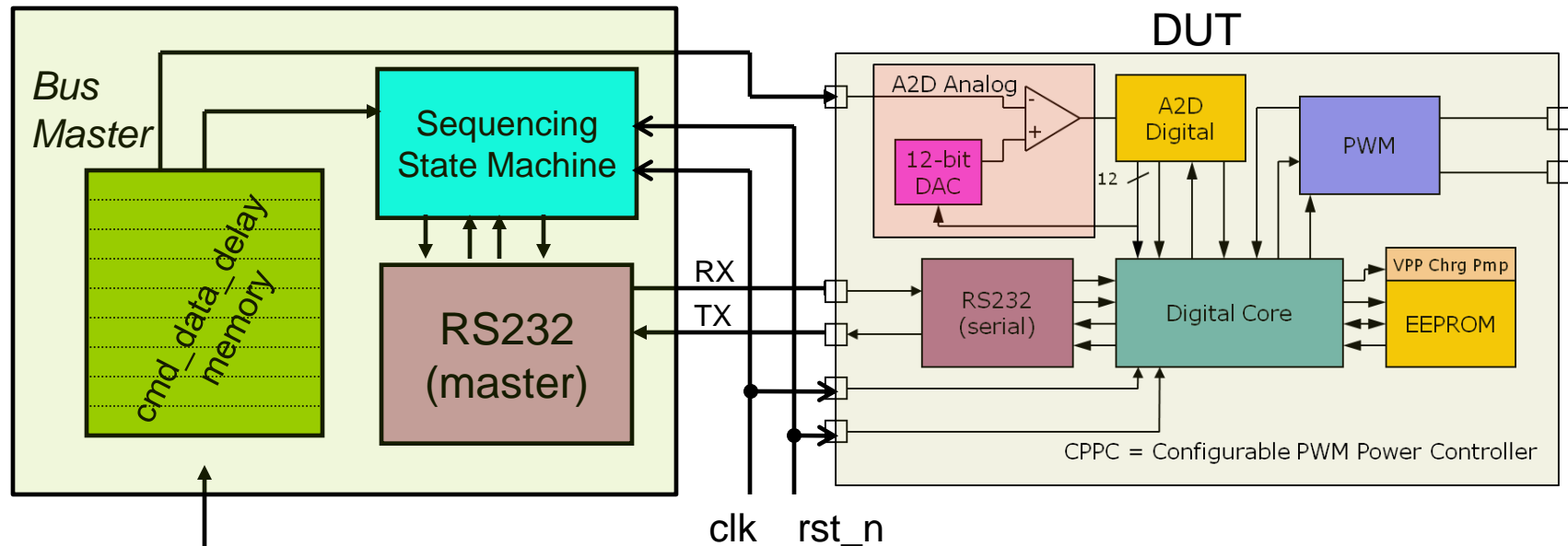
endmodule
```

Testbench Example (contrived but valid)

```
module test_and;
integer file, i, code;
reg a, b, expect, clock;
wire out;
parameter cycle = 20;
and #4 a0(out, a, b);           // Circuit under test

initial begin : file_block
    clock = 0;
    file = $fopen("compare.txt", "r" );
    for (i = 0; i < 4; i=i+1) begin
        @(posedge clock)           // Read stimulus on rising clock
        code = $fscanf(file, "%b %b %b\n", a, b, expect);
        #(cycle - 1)               // Compare just before end of cycle
        if (expect !== out)
            $strobe("%d %b %b %b %b", $time, a, b, expect, out);
    end // for
    $fclose(file); $stop;
end // initial
always #(cycle /2) clock = ~clock; // Clock generator
endmodule
```

Another Testbench Example using File I/O



`$readmemh("master",mem)`

// addr	data
@0000	42C_0800
@0001	000_000D
@0002	000_0A5A

Command

Delay

Expected Data

Testing a DUT that interfaces to outside world through a SPI bus

Many functions this chip performs can be accessed through RS232.

How do you test it?

functions

- Declared and referenced within a module
- Used to implement combinational behavior
 - Contain no timing controls or tasks
- Inputs/outputs
 - Must have at least one input argument
 - Has only one output (no inouts)
 - Function name is implicitly declared return variable
 - Type and range of return value can be specified (1-bit wire is default)

When to use functions?

- Usage rules:
 - May be referenced in any expression (RHS)
 - May call other functions
- Requirements of procedure (implemented as function)
 - No timing or event control
 - Returns a single value
 - Has at least 1 input
 - Uses only behavioral statements
 - Only uses blocking assignments (combinational)
- Mainly useful for conversions, calculations, and selfchecking routines that return boolean. (testbenches)

Function Example

```
module word_aligner (word_out, word_in);  
  output      [7: 0]  word_out;  
  input       [7: 0]  word_in;  
  assign word_out = aligned_word(word_in); // invoke function  
  
  function    [7: 0]  aligned_word;      // function declaration  
    input     [7: 0]  word;  
    begin  
      aligned_word = word;  
      if (aligned_word != 0)  
        while (aligned_word[7] == 0) aligned_word = aligned_word << 1;  
      end  
    endfunction  
endmodule
```

size of return value

input to function

Does this synthesize?

Function Example [2]

```
module arithmetic_unit (result_1, result_2, operand_1, operand_2,);
```

```
  output                [4: 0] result_1;
```

```
  output                [3: 0] result_2;
```

```
  input                 [3: 0] operand_1, operand_2;
```

```
  assign result_1 = sum_of_operands (operand_1, operand_2);
```

```
  assign result_2 = larger_operand (operand_1, operand_2);
```

function call



```
function [4: 0] sum_of_operands(input [3:0] operand_1, operand_2);
```

```
  sum_of_operands = operand_1 + operand_2;
```

```
endfunction
```

function output



function inputs



```
function [3: 0] larger_operand(input [3:0] operand_1, operand_2);
```

```
  larger_operand = (operand_1 >= operand_2) ? operand_1 : operand_2;
```

```
endfunction
```

```
endmodule
```

Re-Entrant (recursive) functions

- Use keyword **automatic** to enable stack saving of function working space and enable recursive functions.

```
module top;  
  
//Define the factorial function  
function automatic integer factorial;  
input [31:0] oper;  
  
begin  
  if (oper>=2)  
    factorial = factorial(oper-1)*oper;  
  else  
    factorial = 1;  
end  
  
endfunction
```

```
initial begin  
  result = factorial(4);  
  $display("Result is %d",result);  
end  
  
endmodule
```

Is this how you would do it?

KISS

tasks (much more useful than functions)

Functions:	Tasks:
A function can enable another function, but not another task	A task can enable other tasks and functions
Functions must execute in zero delay	Tasks may execute in non-zero simulation time
Functions can have no timing or even control statements	Tasks may contain delay, event or timing control. (i.e. \rightarrow @, #)
Function must have at least one input argument.	Task may have zero or more arguments of type input, output, or inout
Functions always return a single value. They cannot have output or inout arguments	Tasks do not return a value, but rather pass multiple values through output and input arguments


Tasks can modify global signals too, perhaps naughty, but I do it all the time.

Why use Tasks?

- Tasks provide the ability to
 - Execute common procedures from multiple places
 - Divide large procedures into smaller ones
- Local variables can be declared & used
- Personally, I only use tasks in testbenches, but they are very handy there.
 - Break common testing routines into tasks
 - ✓ Initialization tasks
 - ✓ Stimulus generation tasks
 - ✓ Self Checking tasks
 - Top level test then becomes mainly calls to tasks.

Task Example [Part 1]

```
module adder_task (c_out, sum, clk, reset, c_in, data_a, data_b, clk);  
  output reg    [3: 0]  sum;  
  output reg          c_out;  
  input [3: 0]  data_a, data_b;  
  input          clk, reset, c_in;  
  
  always @(posedge clk or posedge reset) begin  
    if (reset) {c_out, sum} <= 0;  
    else add_values (sum, c_out, data_a, data_b, c_in); // invoke task  
  end  
  // Continued on next slide
```



Calling of task

Task Example [Part 2]

// Continued from previous slide

task add_values; // task declaration

output reg [3: 0] SUM;

output reg C_OUT;

input [3: 0] DATA_A, DATA_B;

input C_IN;

 {C_OUT, SUM} = DATA_A + (DATA_B + C_IN);

endtask

endmodule

} *task outputs*

} *task inputs*

- Could have instead specified inputs/outputs using a port list.

task add_values (**output reg** [3: 0] SUM, **output reg** C_OUT,
 input [3:0] DATA_A, DATA_B, **input** C_IN);

Task Example [2]

```
task leading_1(output reg [2:0] position, input [7:0]  
data_word);
```

```
  reg          [7:0] temp;
```

```
  begin
```

```
    temp = data_word;
```

```
    position = 7;
```

```
    while (!temp[7]) begin
```

```
      temp = temp << 1;
```

```
      position = position - 1;
```

```
    end
```

```
  end
```

```
endtask
```



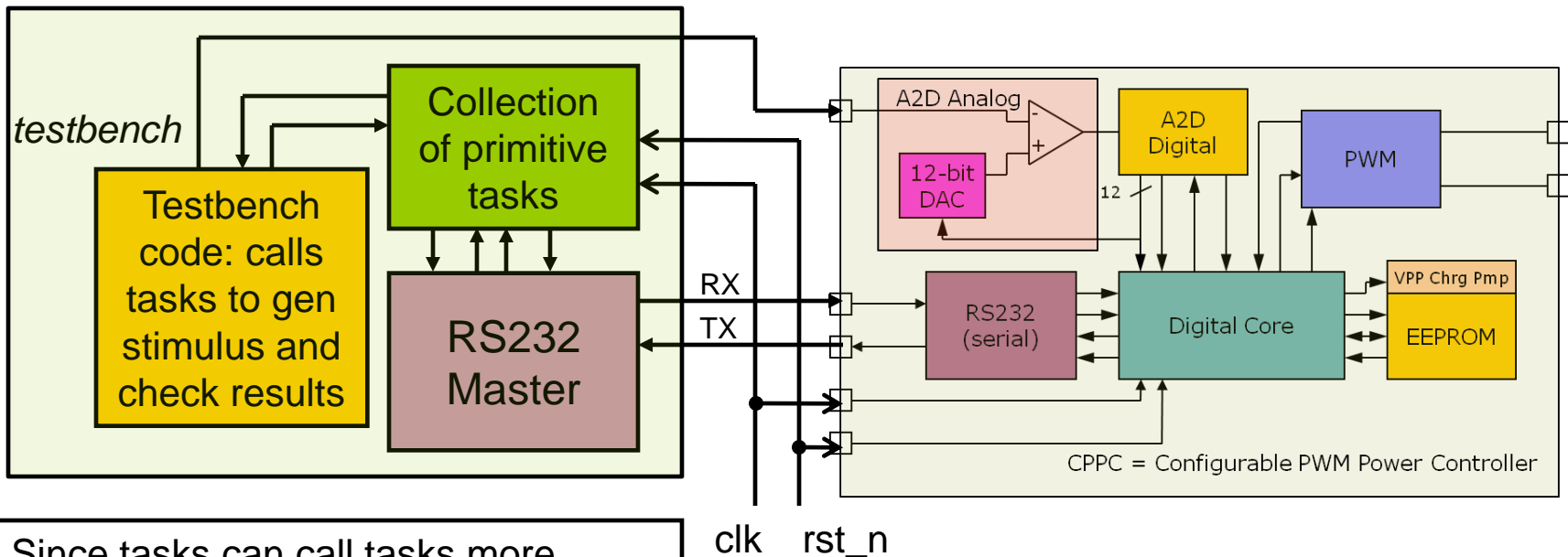
internal task variable

NOTE:

*"while" loops usually
not synthesizable!*

- What does this task assume for it to work correctly?
- How do tasks differ from modules?
- How do tasks differ from functions?

Tasks in testbenches



Since tasks can call tasks more complex macro tasks can be made by calling several primitive tasks.

Primitive: (transmit word on RS232)

Primitive: (wait for frame returned)

Primitive: (read result and compare)

Macro: (a task that call all 3 above)

Now we have tested the command interface of this chip through RS232. How will we test its PWM output?

How can we create the testbench in a different manner making use of tasks?

'Include Compiler Directives

■ **`include** filename

- Inserts entire contents of another file at compilation
- Can be placed anywhere in Verilog source
- Can provide either relative or absolute path names

■ Example 1:

```
module use_adder8(...);  
    `include "adder8.v" // include the task for adder8
```

■ Example 2:

```
module cppc_dig_tb();  
    `include "/home/ehoffman/ece551/project/tb_tasks.v"
```

■ Useful for including tasks and functions in multiple modules

Use of **`include** and **tasks** in testbenches

```
module ctic_dig_tb();

`include "tb_tasks.v"    // include commonly used tb
                        // tasks & parameter definitions

reg [15:0] response;    // holds SPI results from DUT
reg fail=0;            // initially test has not failed

///// instantiate DUT /////
ctic_dig dut(.miso(miso), .mosi(mosi), ... .rdy(rdy));

initial begin
    @(posedge dut.rst_n)    // wait for rst to deassert
    init_master;           // call of task to initialize SPI master
    snd_rcv(rd_eep0,response); // read eeprom address 0
    @(posedge dut.rdy)
    snd_rcv(rd_eep1,response);
    chk_resp(fail,response,16'h2832); // expected response
    if (fail) $stop;
    @(posedge dut.rdy)
    tb_clks(10);           // waits 10 clocks
    ...
end
```

```
parameter rd_eep1 = 16'h3000;
parameter rd_eep1 = 16'h3001;
...
task tb_clks(input integer num_clks);
    repeat(numclks) @(posedge dut.clk);
endtask

task init_master
    ....
endtask
...
tb_tasks.v
```