TSEA44: Computer hardware – a system on a chip

Lecture 7: DMA, lab3, testbenches

LINKÖPING UNIVERSITY

Today

- Hints for documentation
- DMA
- Lab3
- Testbenches

LINKÖPING UNIVERSITY

Lab reports

- Lab1: Section 3.7.2 is good reading
  - Specifies what to include (code, diagrams, state graphs)
  - Specifies things to discuss in the report
- Same type of section found for the other lab tasks also
- Include all code you have written/modified
  - Assume the reader have access to the original lab setup

Creating schematics

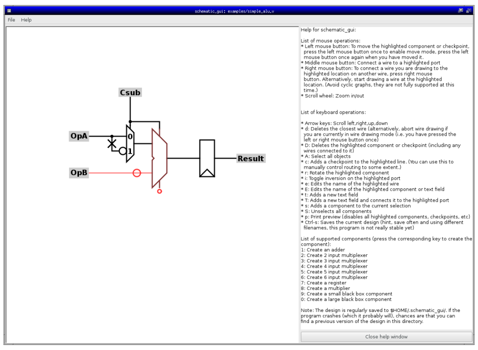
- Alternatives
  - Openoffice/libreoffice diagram tool (I use this for slides)
  - Inkscape (potentially very nice looking, very cumbersome though)
  - Dia (decent if you have RTL library for it)
  - TikZ (if you really like latex)
  - MS Paint (I'm only kidding)
  - Hand drawn schematics from whiteboard/paper
    - Quality problems…
  - Visio (if you have a license for it)

schematic_gui

- Previous examiners (Andreas Ehliar) hobby project
- http://github.com/ehliar/schematic_gui
  - Tutorial at https://github.com/ehliar/schematic_gui/blob/master/tutorial/tutorial.md
- Accessible also on computers in the lab

  module load TSEA44

  schematic_gui



Help for schematic_gui:

List of mouse operations:
* Left mouse button: To move the highlighted component or checkpoint, press the left mouse button once to enable move mode, press the left mouse button once again when you have moved it.
* Middle mouse button: Connect a wire to a highlighted port
* Right mouse button: To connect a wire you are drawing to the highlighted location on another wire, press right mouse button. Alternatively, start drawing a wire at the highlighted location. (Avoid cyclic graphs, they are not fully supported at this time.)
* Scroll wheel: Zoom in/out

List of keyboard operations:

* Arrow keys: Scroll left,right,up,down
* d: Deletes the closest wire (alternatively, abort wire drawing if you are currently in wire drawing mode (i.e. you have pressed the left or right mouse button once)
* D: Deletes the highlighted component or checkpoint (including any wires connected to it)
* A: Select all objects
* c: Adds a checkpoint to the highlighted line. (You can use this to manually control routing to some extent.)
* r: Rotate the highlighted component
* i: Toggle inversion on the highlighted port
* e: Edits the name of the highlighted wire
* E: Edits the name of the highlighted component or text field
* t: Adds a new text field
* T: Adds a new text field and connects it to the highlighted port
* s: Adds a component to the current selection
* S: Unselects all components
* p: Print preview (disables all highlighted components, checkpoints, etc)
* Ctrl-s: Saves the current design (hint, save often and using different filenames, this program is not really stable yet)

List of supported components (press the corresponding key to create the component):
1: Create an adder
2: Create 2 input multiplexer
3: Create 3 input multiplexer
4: Create 4 input multiplexer
5: Create 5 input multiplexer
6: Create 6 input multiplexer
7: Create a register
8: Create a multiplier
9: Create a small black box component
0: Create a large black box component

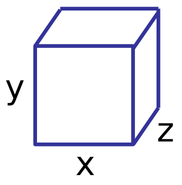
Note: The design is regularly saved to $HOME/.schematic_gui/. If the program crashes (which it probably will), chances are that you can find a previous version of the design in this directory.

Packed arrays, how to use them

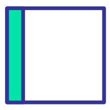
left to right, right first

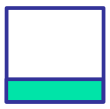
logic [11:0] tm1[0:7][0:7];

logic [0:7][0:7][11:0] tm2;



tm1[0][0] // DC component
tm2[0][0] // -”-

tm2[0] // 

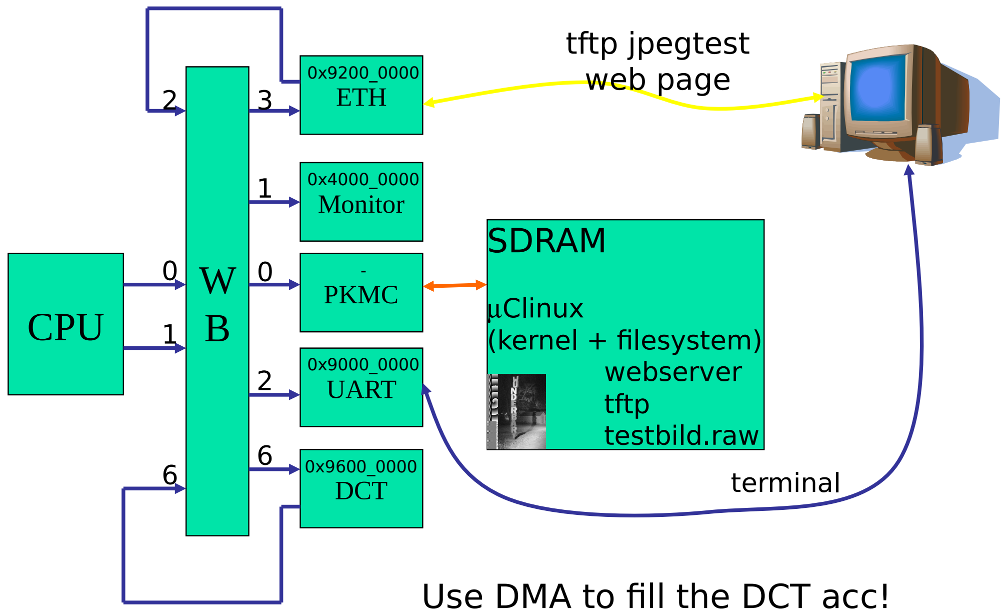
tm2[0:7][0] // 

Array slicing

The size of the part select or slice must be constant, but the position can be variable.

```
logic [31:0] b;
logic [7:0] a1, a2;

a1 = b[x -: 8];          // OK fixed width
a2 =  b[y +: 8];         // OK fixed width
d = b[x:y];              // not OK
```

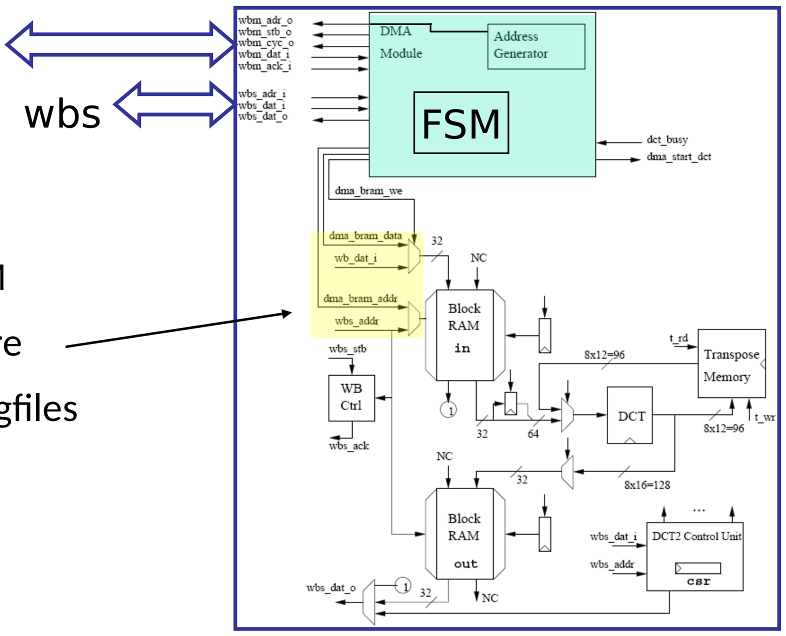
Lab 3 - DMA



Use DMA to fill the DCT acc!

2016-11-24 23:45

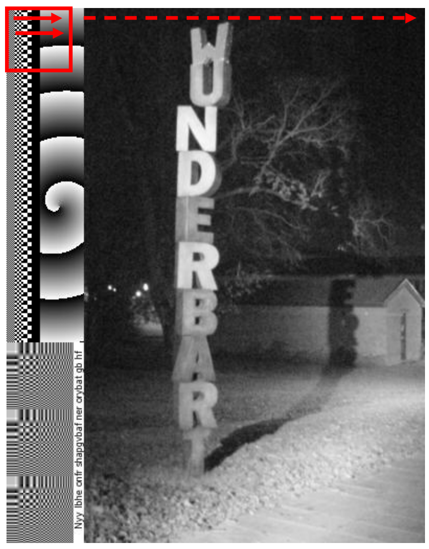
Proposed architecture



- Design FSM
- Change here
- Modify jpegfiles

Address generation

- We want to transfer block by block (8x8)
- Address generator must konw format (width, height) of image

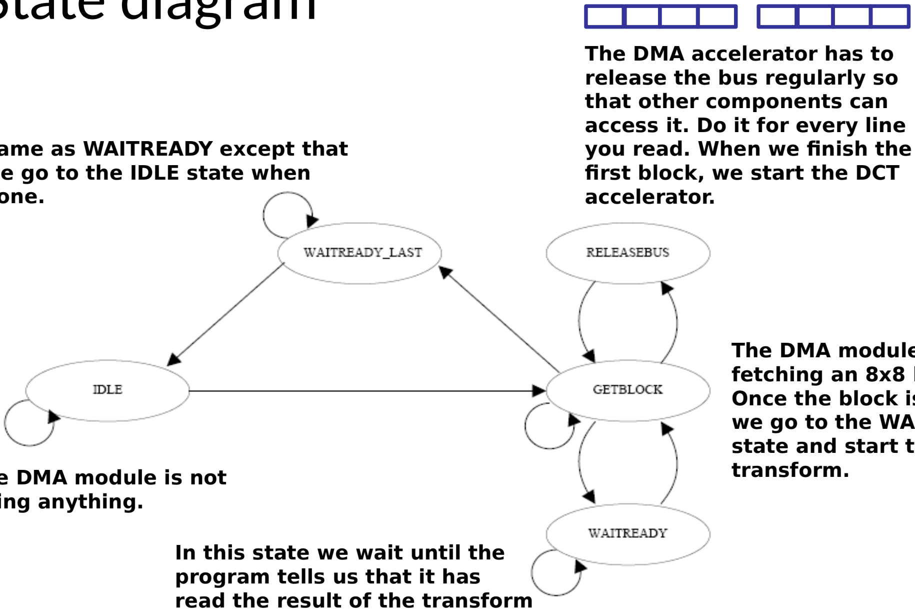


testbild.raw

State diagram

The DMA accelerator has to release the bus regularly so that other components can access it. Do it for every line you read. When we finish the first block, we start the DCT accelerator.

Same as WAITREADY except that we go to the IDLE state when done.



The DMA module is fetching an 8x8 block. Once the block is fetched we go to the WAITREADY state and start the DCT transform.

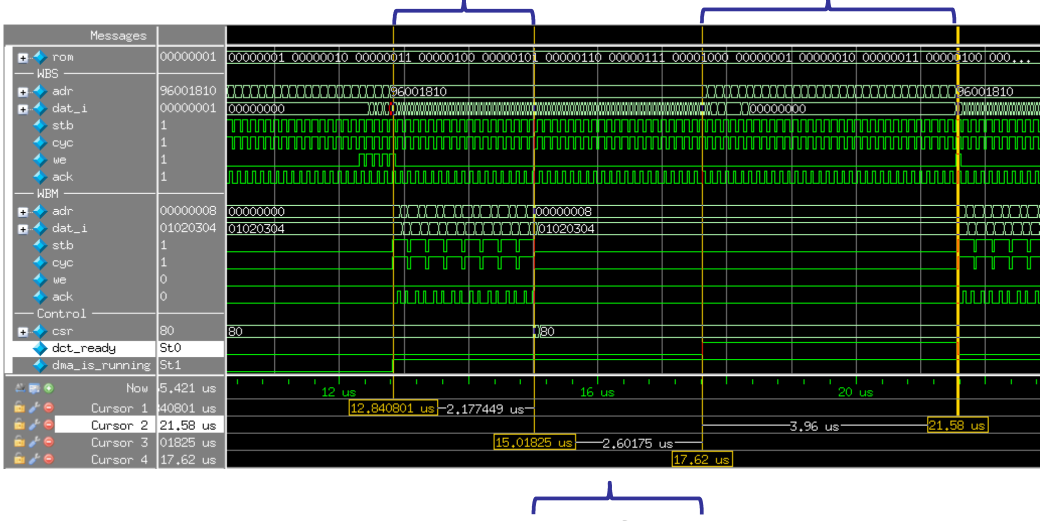
The DMA module is not doing anything.

In this state we wait until the program tells us that it has read the result of the transform by writing to the control register.

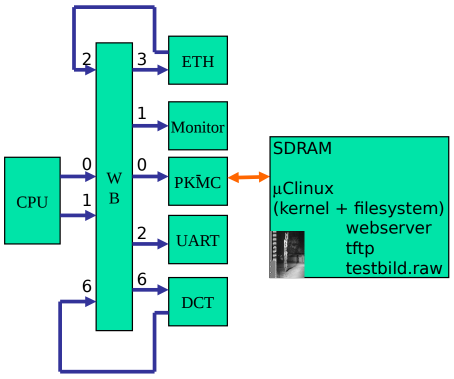
A measurement: make sim_jpeg

Copy 16 words from **SDRAM** to **DCT (DMA)**

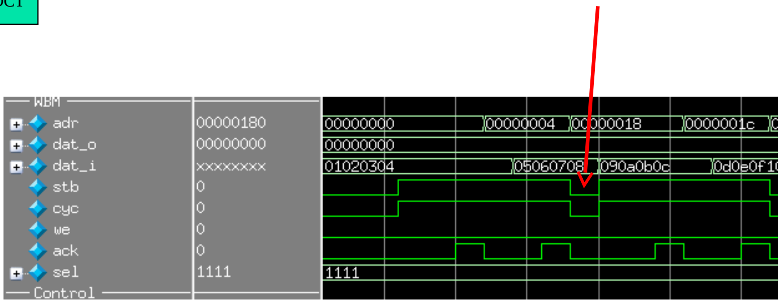
Copy 32 words from **DCT** to **SDRAM**



A closer look at the DMA

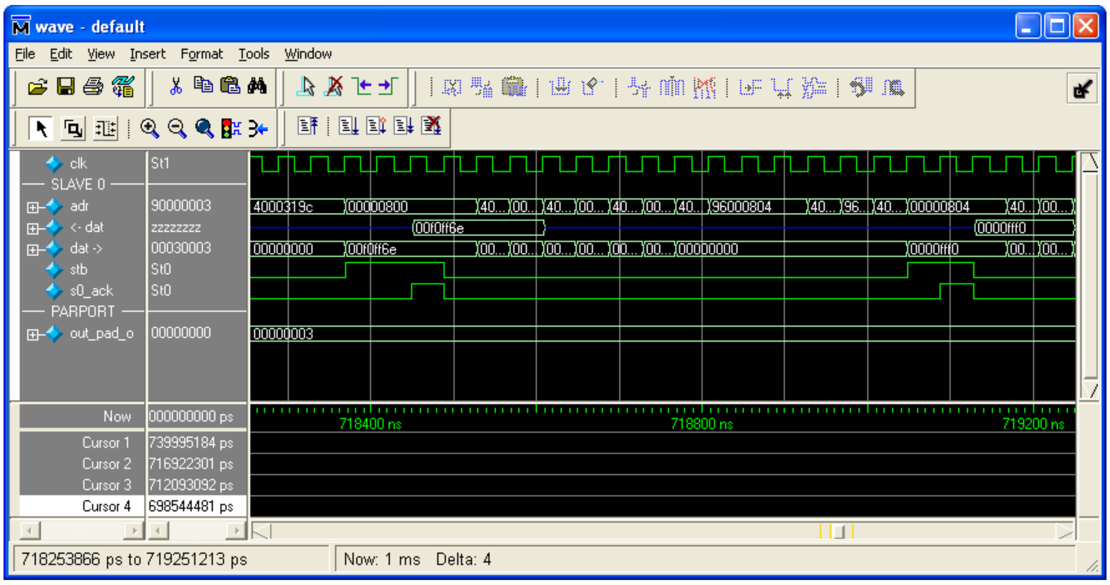


Release bus for
m0, m1, m2
⇒ If CPU is waiting it will
get the bus



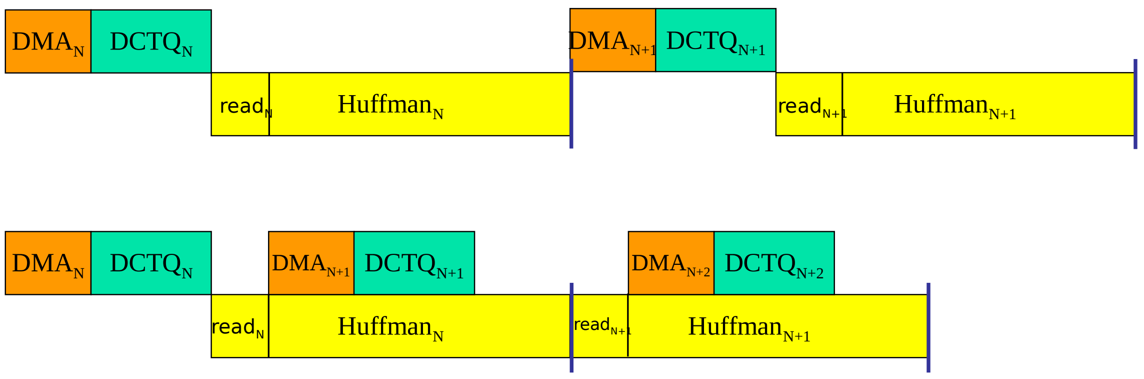
```
WBM
adr      00000180   00000000  00000004  00000018  0000001c
dat_o    00000000   00000000
dat_i    xxxxxxxx   01020304  05060708  090a0b0c  0d0e0f10
stb      0
cyc      0
we       0
ack      0
sel      1111       1111
Control
```

DCT => Memory (Software)



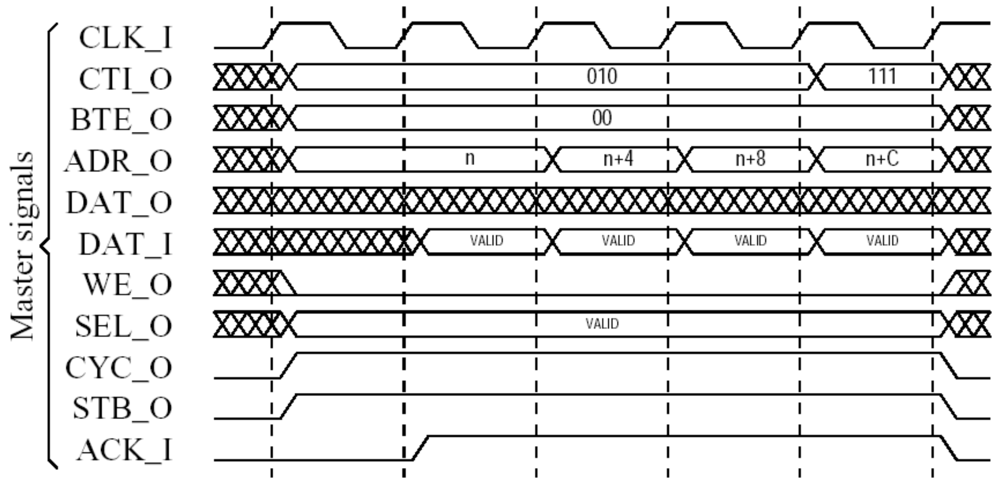
```
clk          St1
SLAVE 0
adr          90000003    4000319c  00000800  40... 00... 40... 00... 40... 00... 40... 96000804  40... 96... 40... 00000804  40... 00...
<- dat       zzzzzzzz    00f0ff6e  0000fff0
dat ->       00030003    00000000  00f0ff6e  00... 00... 00... 00... 00... 00... 00000000  0000fff0  00... 00...
stb          St0
s0_ack       St0
PARPORT
out_pad_o    00000000    00000003

Now          000000000 ps     718400 ns    718800 ns    719200 ns
Cursor 1     739995184 ps
Cursor 2     716922301 ps
Cursor 3     712093092 ps
Cursor 4     698544481 ps

718253866 ps to 719251213 ps      Now: 1 ms   Delta: 4
```

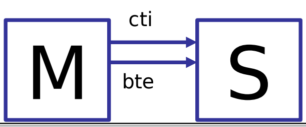
A hint



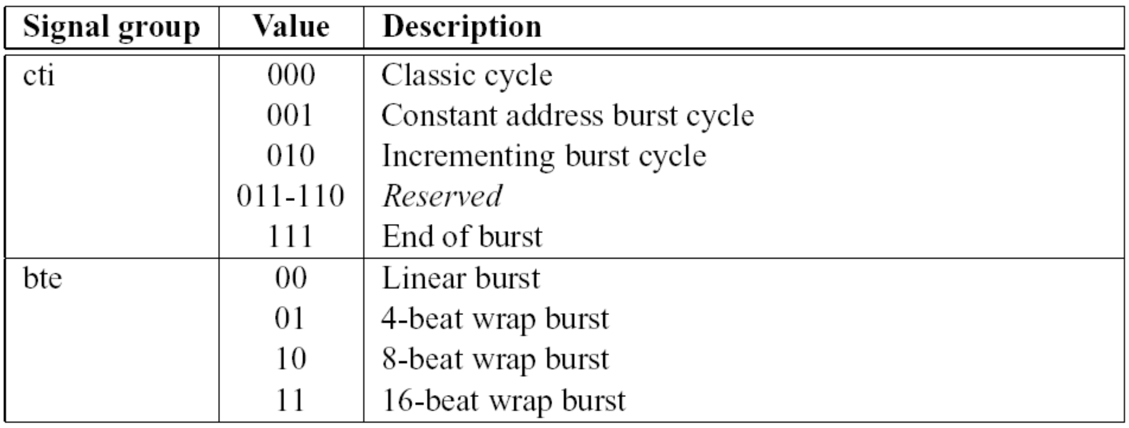
How long time do these blocks take?

Burst Read





Burst cycle types



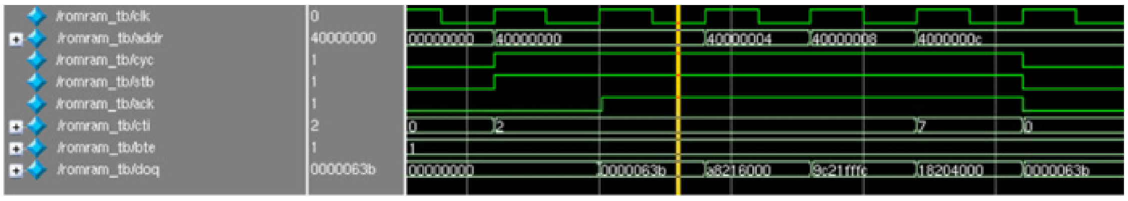
Signal group	Value	Description
cti	000	Classic cycle
	001	Constant address burst cycle
	010	Incrementing burst cycle
	011-110	*Reserved*
	111	End of burst
bte	00	Linear burst
	01	4-beat wrap burst
	10	8-beat wrap burst
	11	16-beat wrap burst

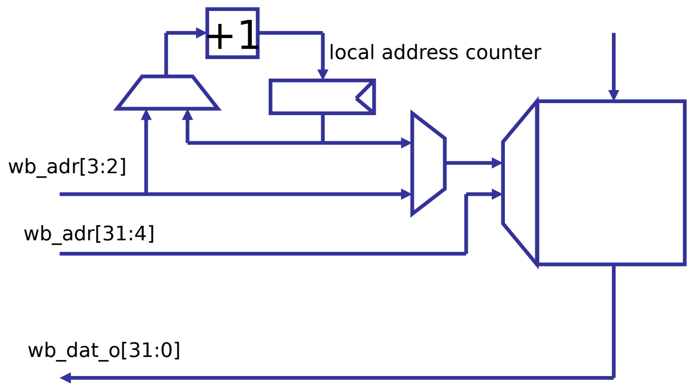
Burst access

- Note: Only the SRAM memory controller i the Leela memory controller has burst support
  - It is a graphics controller not used in our lab setup

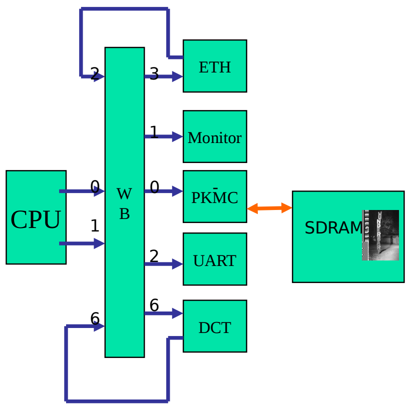
2016-11-24 23:45

Changes in the slave





Why not write DMA? (acc -> memory)



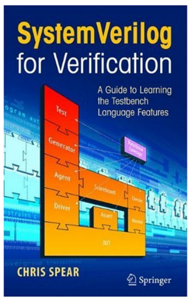
```
// This the main encoding loop
void encode_image(void)
{
   int i;
   int MCU_count = width*height/DCTSIZE2;
   short MCU_block[DCTSIZE2];

   for(i = 0; i < MCU_count; i++)
   {
      forward_DCT(MCU_block);
      encode_mcu_huff(MCU_block);
   }
}
```

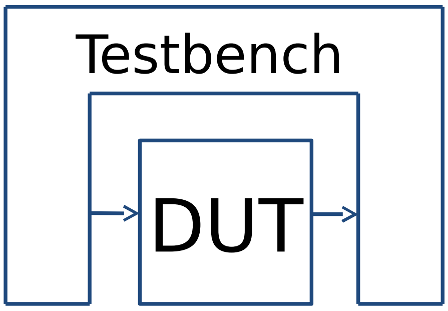
1) I/O is on 0x90, 0x91, …, 0x99
   other addr to PKMC
2) Noncacheable data mem addr >= 0x8000_0000,
   SDRAM 0x0, SRAM 0x2000_0000 or 0xc000_0000
2) **`MCU_block`** must be in noncacheable area
3) Skip **`MCU_block,`** let **`encode_mcu_huff`** read from **`acc`**

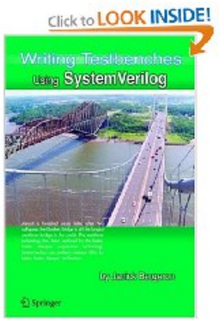
Testbenches

Spear,Chris:
System Verilog for verification.
Springer

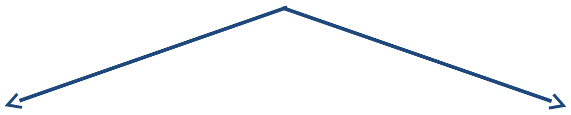
Bergeron,Janick:
Writing testbenches using System Verilog.
Springer


SystemVerilog for Verification
A Guide to Learning the Testbench Language Features
CHRIS SPEAR
Springer





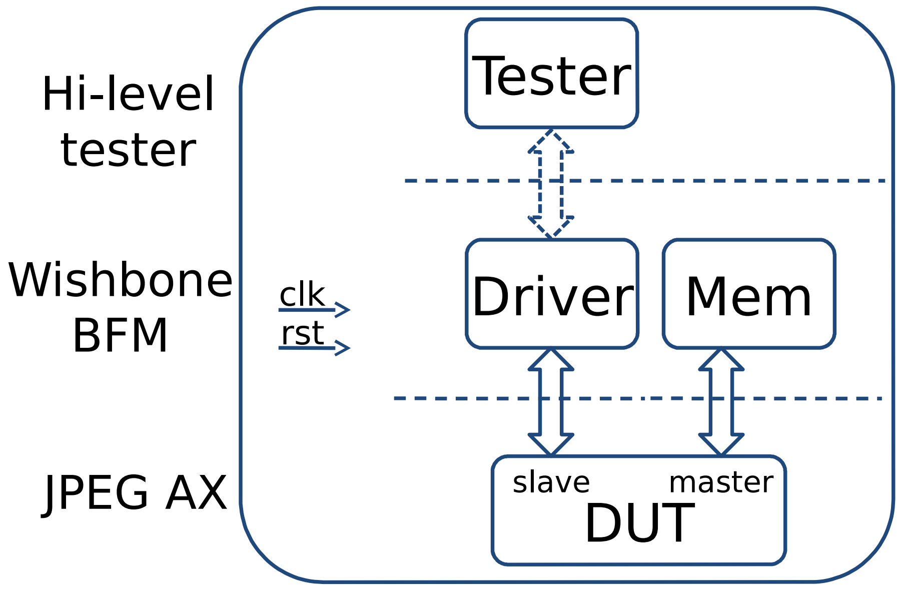
Testbenches



Like an FSM
(same as DUT)
- complicated to design
- hard to test timing
- hard to test flow

Like High-Level Software
(very different from DUT)
- easy to design
- easy to test timing
- easy to test flow

An example: A TB for your design



LINKÖPING UNIVERSITY

Testbench: top level

```
module jpeg_top_tb();
   logic       clk = 1'b0;
   logic       rst = 1'b1;
   wishbone wb(clk,rst), wbm(clk,rst);

   initial begin
      #75 rst = 1'b0;
   end

   always #20 clk = ~clk;

   // Instantiate the tester
   tester tester0();

   // Instantiate the drivers
   wishbone_tasks wb0(.*);

   // Instantiate the DUT
   jpeg_top dut(.*);
   mem mem0(.*);
endmodule // jpeg_top_tb
```

LINKÖPING UNIVERSITY

Testbench: Hi-level tester

```
program tester();
  int result = 0;
  int d = 32'h01020304;

  initial begin

    for (int i=0; i<16; i++) begin
      jpeg_top_tb.wb0.m_write(32'h96000000 + 4*i, d); // fill inmem
      d += 32'h04040404;
    end

    jpeg_top_tb.wb0.m_write(32'h96001000, 32'h01000000); // start ax

    while (result != 32'h80000000)
      jpeg_top_tb.wb0.m_read(32'h96001000,result);    // wait for ax

      for (int j=0; j<8; j++) begin
        for (int i=0; i<4; i++) begin                  // print outmem
          jpeg_top_tb.wb0.m_read(32'h96000800 + 4*i + j*16,result);
          $fwrite(1,"%5d ", result >>> 16);
          $fwrite(1,"%5d ", (result << 16) >>>16);
        end
        $fwrite(1,"\n");
      end
  end
endprogram // tester
```

Testbench: mem

```
module mem(wishbone.slave wbm);
   logic [7:0] rom[0:2047];
   logic [1:0] state;
   logic [8:0] adr;
   integer     blockx, blocky, x, y, i;

   initial begin
   // A test image, same as dma_dct_hw.c
   for (blocky=0; blocky<`HEIGHT; blocky++)
      for (blockx=0; blockx<`WIDTH; blockx++)
         for (i=1, y=0; y<8; y++)
            for (x=0; x<8; x++)
               rom[blockx*8+x+(blocky*8+y)*`PITCH] = i++;    // these are not wishbone cycles
   end

   assign wbm.err = 1'b0;
   assign wbm.rty = 1'b0;

   always_ff @(posedge wbm.clk)
      if (wbm.rst)
         state <= 2'h0;
      else
         case (state)
            2'h0: if (wbm.stb) state <= 2'h1;
            2'h1: state <= 2'h2;
            2'h2: state <= 2'h0;
         endcase
```

```
   assign wbm.ack = state[1];

   always_ff @(posedge wbm.clk)
      adr <= wbm.adr[8:0];

   assign wbm.dat_i = {rom[adr], rom[adr+1],
                       rom[adr+2], rom[adr+3]};
endmodule // mem
```

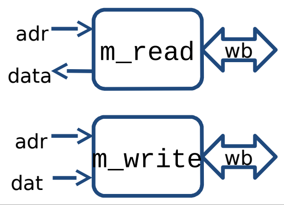
DMA? Easy!

```
…
// Init DMA-engine
      jpeg_top_tb.wb0.m_write(32'h96001800, 32'h0);
      jpeg_top_tb.wb0.m_write(32'h96001804, ?);
      jpeg_top_tb.wb0.m_write(32'h96001808, ?);
      jpeg_top_tb.wb0.m_write(32'h9600180c, ?);
      jpeg_top_tb.wb0.m_write(32'h96001810, ?);           // start DMA engine

      for (int blocky=0; blocky<`HEIGHT; blocky++) begin
         for (int blockx=0; blockx<`WIDTH; blockx++) begin
            // Wait for DCTDMA to fill the DCT accelerator
            result = 0;
            while (?)                              // wait for block to finish
               jpeg_top_tb.wb0.m_read(32'h96001810, result);
            $display("blocky=%5d blockx=%5d", blocky, blockx);
            for (int j=0; j<8; j++) begin
               for (int i=0; i<4; i++) begin
                  jpeg_top_tb.wb0.m_read(32'h96000800 + 4*i + j*16, result);
                  $fwrite(1,"%5d ", result >>> 16);
                  $fwrite(1,"%5d ", (result << 16) >>>16);
               end
               $fwrite(1,"\n");
            end
            jpeg_top_tb.wb0.m_write(?);              // start next block
         end
      end
      …
```

wishbone_tasks.sv

- May/may not consume time
- May/may not be synthable
- Do not contain **always/initial**
- Do not return values. Pass via output

```
module wishbone_tasks(wishbone.master wb);
   int result = 0;

   reg oldack;
   reg [31:0] olddat;

   always_ff  @(posedge wb.clk) begin
      oldack <= wb.ack;
      olddat <= wb.dat_i;
   end
```

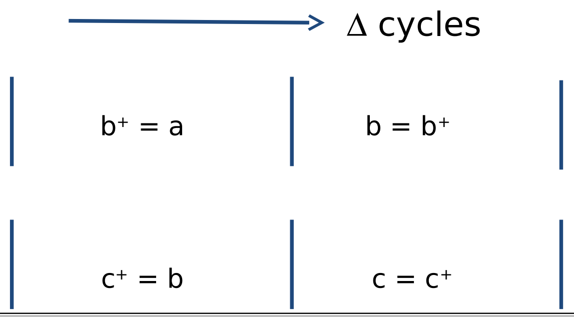


```
// *******************************
task m_read(input [31:0] adr,
            output logic [31:0] data);
   begin
   @(posedge wb.clk);
   wb.adr <= adr;
   wb.stb <= 1'b1;
   wb.we  <= 1'b0;
   wb.cyc <= 1'b1;
   wb.sel <= 4'hf;

   @(posedge wb.clk);
   #1;

   while (!oldack) begin
     @(posedge wb.clk);
     #1;
   end

   wb.stb <= 1'b0;
   wb.we  <= 1'b0;
   wb.cyc <= 1'b0;
   wb.sel <= 4'h0;

   data = olddat;
   end
endtask // m_read

// *******************************
task m_write(input [31:0] adr,
             input [31:0] dat);
  // similar to m_read
endtask // m_write

endmodule // wishbone_tasks
```

Race conditions

Threads executing
in parallel
in
no particular order

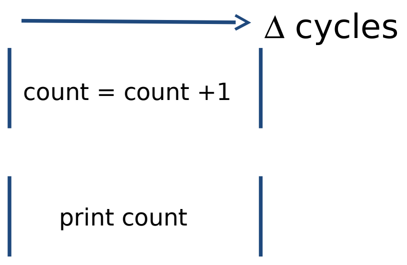
```
always_ff @(posedge clk) begin
   b <= a;
end

always_ff @(posedge clk) begin
   c <= b;
end
```

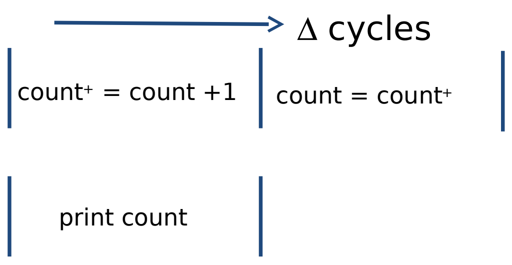


Race conditions

```
always_ff @(posedge clk) begin
   count = count + 1;
end

always_ff @(posedge clk) begin
   $write(”count=%d\n”, count);
end
```

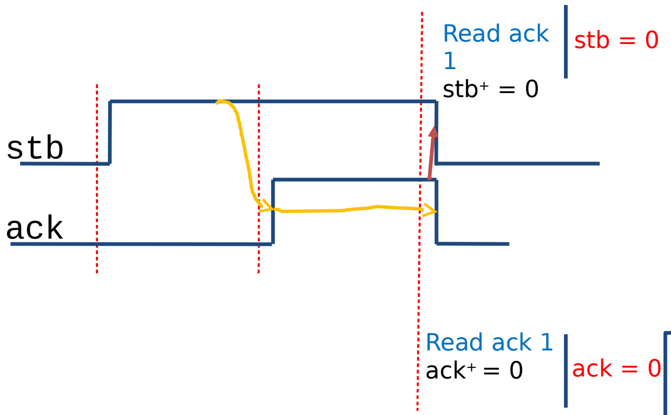
```
always_ff @(posedge clk) begin
   count <= count + 1;
end

always_ff @(posedge clk) begin
   $write(”count=%d\n”, count);
end
```





2016-11-24 23:45

Hm…



TB
```
initial begin
 @(posedge clk);
 stb <= 1;

 @(posedge clk);

 while (ack == 0)
   @(posedge clk);

 stb <= 0;
end
```

ack stb

DUT
```
always_ff @(posedge clk)
  case (ack)
     0: if (stb)
          ack <= 1;
     1: ack <= 0;
  endcase;
```

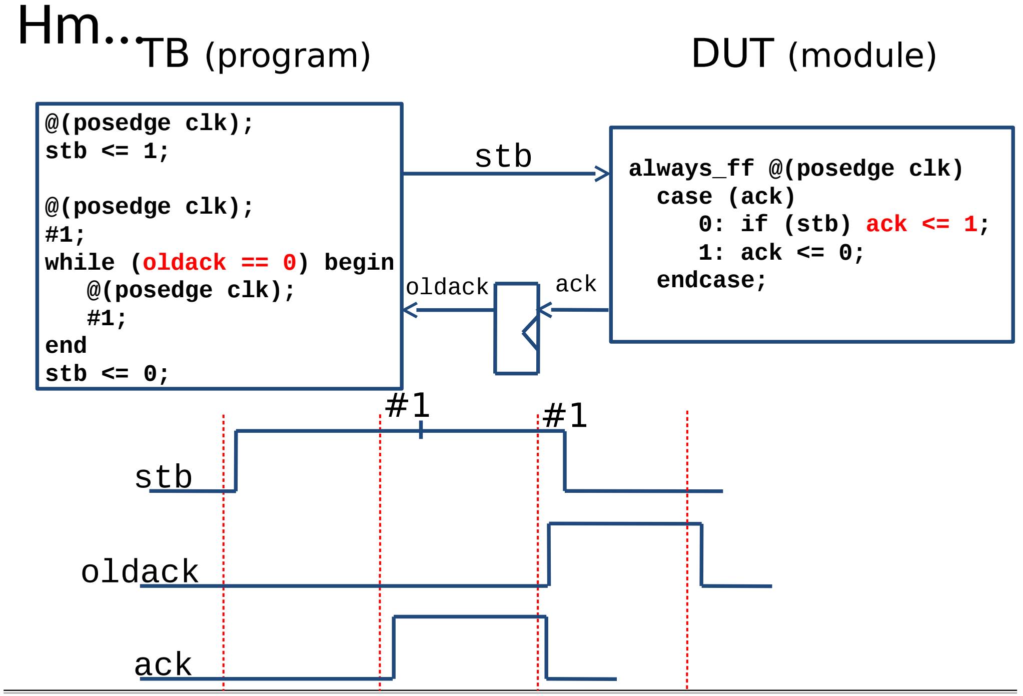
Nonblocking assignment (<=)
=> no race condition
Blocking assignment (=)
=> race condition

program block

- Purpose: Identifies verification code
- A program is different from a module
  - Only initial blocks allowed
  - Executes last
  - (module -> clocking/assertions -> program)
  - No race situation in previous example!

The Program block functions pretty much like a C program
Testbenches are more like software than hardware

Hm..



TB (program)

```
@(posedge clk);
stb <= 1;

@(posedge clk);
#1;
while (oldack == 0) begin
   @(posedge clk);
   #1;
end
stb <= 0;
```

DUT (module)

```
always_ff @(posedge clk)
  case (ack)
     0: if (stb) ack <= 1;
     1: ack <= 0;
  endcase;
```

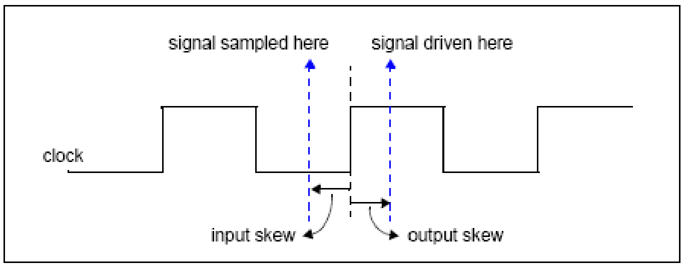
Clocking block

SystemVerilog adds the clocking block that identifies clock signals, and capture the timing and synchronization requirements of the blocks being modeled.

A clocking block assembles signals that are synchronous to a particular clock, and makes their timing explicit.

The clocking block is a key element in cycle-based methodology, which enables users to write testbenches at a higher level of abstraction. Rather than focusing on signals and transitions in time, the test can be defined in terms of cycles and transactions.

Possible to simulate setup and hold time



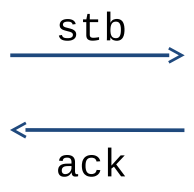
Clocking block

```
interface wishbone(input clk,rst);
   wire stb,ack;

   clocking cb @(posedge clk);
      input ack;
      output stb;
   endclocking // cb

   modport  tb (clocking cb,
                input clk,rst);

endinterface // wishbone
```

```
module tb();
   logic       clk = 1'b0;
   logic       rst = 1'b1;

   // instantiate a WB
   wishbone wb(clk,rst);

   initial begin
      #75 rst = 1'b0;
   end

   always #20 clk = ~clk;

   // Instantiate the DUT
   jpeg_top dut(.*);

   // Instantiate the tester
   tester tester0(.*);
   mem mem0(.*);
endmodule // jpeg_top_tb
```

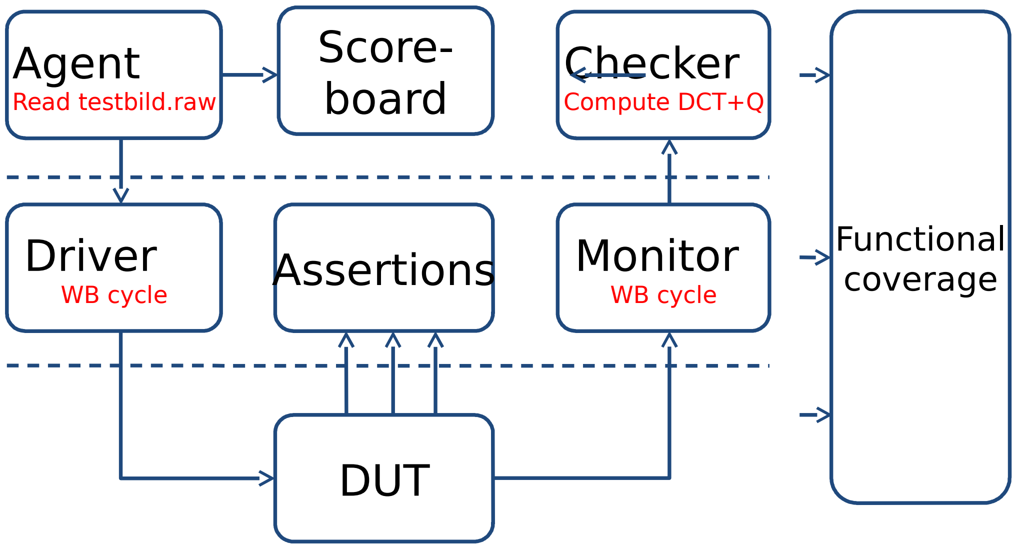
Clocking block

```
program tester(wishbone.tb wb);

   …

   initial begin
      for (int i=0; i<3; i++) begin
         wb.cb.stb <= 0;
         ##1;
         wb.cb.stb <= 1;
         while (wb.cb.ack==0)
            ##1;
      end
   end
endprogram // tester
```



```
module jpeg_top(wishbone wb);
   reg state;

   assign wb.ack = state;

   always_ff @(posedge wb.clk)
     if (wb.rst)
       state <= 1'b0;
     else if(state)
       state <= 1'b0;
     else if (wb.stb)
       state <= 1'b1;
endmodule // jpeg_top
```

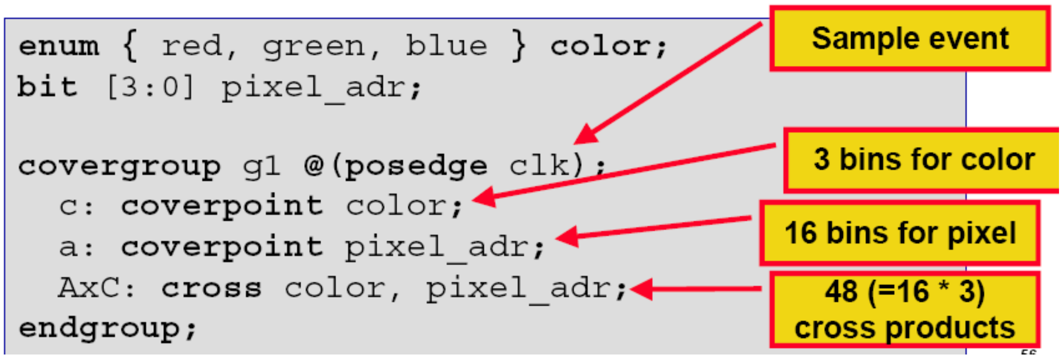
A complex testbench (from Spear: SV for verification)



Object Oriented Programming

- SV includes OOP
- Classes can be defined
  - Inside a program
  - Inside a module
  - Stand alone

Cross coverage



```
enum { red, green, blue } color;
bit [3:0] pixel_adr;

covergroup g1 @(posedge clk);
  c: coverpoint color;
  a: coverpoint pixel_adr;
  AxC: cross color, pixel_adr;
endgroup;
```

Tom Fitzpatrick, SystemVerilog for VHDL Users, DATE'04

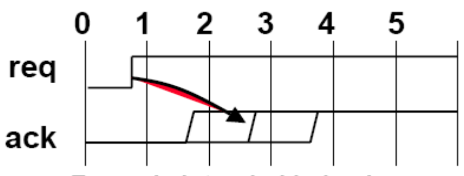
OOP

```
program class_t;

  class packet;
     // members in class
     integer size;
     integer payload [];
     integer i;
     // Constructor
     function new (integer size);
       begin
         this.size = size;
         payload = new[size];
         for (i=0; i < this.size; i ++)
           payload[i] = $random();
       end
     endfunction
     // Task in class (object method)
     task print ();
       begin
         $write("Payload : ");
         for (i=0; i < size; i ++)
           $write("%x ",payload[i]);
         $write("\n");
       end
     endtask
```

```
// Function in class (object method)
     function integer get_size();
       begin
         get_size = this.size;
       end
     endfunction
   endclass

   packet pkt;

   initial begin
     pkt = new(5);
     pkt.print();
     $display ("Size of packet %0d",
               pkt.get_size());
   end

endprogram
```

What is an assertion?

- A concise description of [un]desired behavior

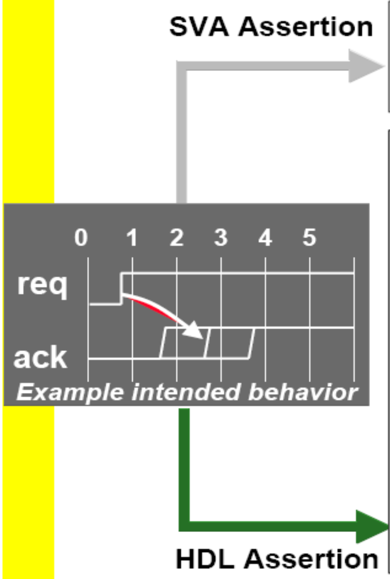


Example intended behavior

“After the request signal is asserted, the acknowledge signal must come 1 to 3 cycles later”

Tom Fitzpatrick, SystemVerilog for VHDL Users, DATE'04

Assertions



```
property req_ack;
 @(posedge clk) req ##[1:3] $rose(ack);
endproperty
as_req_ack: assert property (req_ack);
```

VHDL

```
sample_inputs : process (clk)
begin
  if rising_edge(clk) then
    STROBE_REQ <= REQ;
    STROBE_ACK <= ACK;
  end if;
end process;
protocol: process
  variable CYCLE_CNT : Natural;
begin
  loop
    wait until rising_edge(CLK);
    exit when (STROBE_REQ = '0') and (REQ = '1');
  end loop;
  CYCLE_CNT := 0;
  loop
    wait until rising_edge(CLK);
    CYCLE_CNT := CYCLE_CNT + 1;
    exit when ((STROBE_ACK = '0') and (ACK = '1')) or (CYCLE_CNT = 3);
  end loop;
  if ((STROBE_ACK = '0') and (ACK = '1')) then
    report "Assertion success" severity Note;
  else
    report "Assertion failure" severity Error;
  end if;
end process protocol;
```

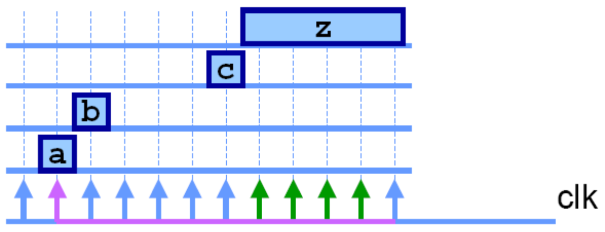
Tom Fitzpatrick, SystemVerilog for VHDL Users, DATE'04

Assertions

- Assertions are built of
  1. Boolean expressions
  2. Sequences
  3. Properties
  4. Assertion directives

Sequential regular expressions

- Describing a sequence of events
- Sequences of Boolean expressions can be described with a specified time step in-between
- ##N delay operator
- [*N] repetition operator



```
sequence s1;
  @(posedge clk) a ##1 b ##4 c ##[1:5] z;
endsequence
```

Properties

- Declare property by name
- Formal parameters to enable property reuse
- Top level operators
  - not desired/undesired
  - disable iff reset
  - |->, |=> implication

```
property p1;
disable iff (rst)
   x |-> s1;
endproperty
```

Assertion Directives

- assert – checks that the property is never violated
- cover – tracks all occurrences of property

  a1: assert p1 else $display(”grr”);

```
property s2a;
     @(posedge clk) disable iff (rst)
             $rose(stb) |-> ##[0:16] $rose(ack);
endproperty

a_s2a:assert property (s2a) else
      $display("   (%0t)(%m) Delayed ack on addr %h",
                $time, adr);
```

Coverage

- *Code coverage (code profiling)*
  - reflects how thorough the HDL code was exercised
- *Functional Coverage (histogram binning)*
  - perceives the design from a user's or a system point of view
  - Have you covered all of your typical scenarios?
  - Error cases? Corner cases? Protocols?
- Functional coverage also allows relationships,
  - ”OK, I've covered every state in my state machine, but did I ever have an interrupt at the same time? When the input buffer was full, did I have all types of packets injected? Did I ever inject two errorneous packets in a row?”

Coverage

```
//   DUT With Coverage
module simple_coverage();

logic [7:0]  addr;
logic [7:0]  data;
logic        par;
logic        rw;
logic        en;

// Coverage Group
covergroup memory @ (posedge en);
  address : coverpoint addr {
    bins low    = {0,50};
    bins med    = {51,150};
    bins high   = {151,255};
  }
  parity : coverpoint  par {
    bins even  = {0};
    bins odd   = {1};
  }
  read_write : coverpoint rw {
    bins  read  = {0};
    bins  write = {1};
  }
endgroup
```

```
memory mem = new();

// Task to drive values
task drive (input [7:0] a, input [7:0] d,
            input r);
  #5 en <= 1;
  addr  <= a;
  rw    <= r;
  data  <= d;
  par   <= ^d;
  $display ("@%2tns Address :%d data %x,
            rw %x, parity %x",
            $time,a,d,r, ^d);
  #5    en <= 0;
  rw    <= 0;
  data  <= 0;
  par   <= 0;
  addr  <= 0;
  rw    <= 0;
endtask

// Testvector generation
initial begin
  en = 0;
  repeat (10) begin
    drive ($random,$random,$random);
  end
  #10 $finish;
end

endmodule
```

Report

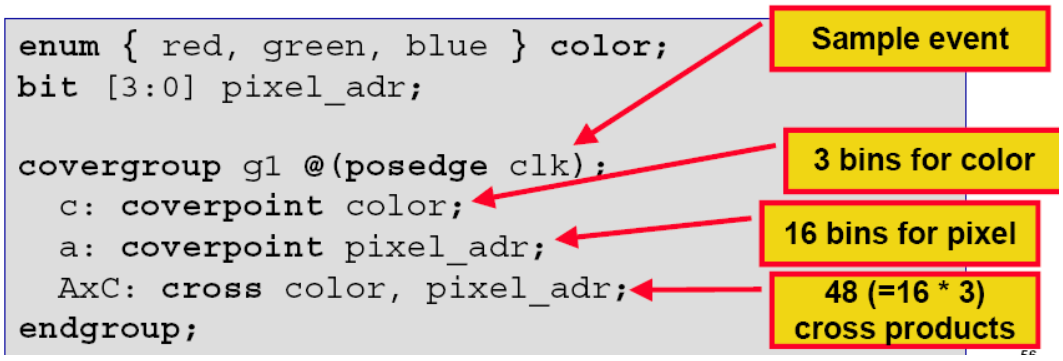
```
# @ 5ns Address :  36 data 81, rw 1, parity 0
# @15ns Address : 99 data 0d, rw 1, parity 1
# @25ns Address :101 data 12, rw 1, parity 0
# @35ns Address : 13 data 76, rw 1, parity 1
# @45ns Address :237 data 8c, rw 1, parity 1
# @55ns Address :198 data c5, rw 0, parity 0
# @65ns Address :229 data 77, rw 0, parity 0
# @75ns Address :143 data f2, rw 0, parity 1
```

ModelSim says:

```
COVERGROUP COVERAGE:
-----------------------------------------------------------------------------------------
Covergroup                                    Metric      Goal/ Status
                                               At Least
-----------------------------------------------------------------------------------------
 TYPE /simple_coverage/memory                  44.4%       100 Uncovered
    Coverpoint memory::address                 33.3%       100 Uncovered
        covered/total bins:                       1          3
        bin low                                   9          1 Covered
        bin med                                   0          1 ZERO
        bin high                                  0          1 ZERO
    Coverpoint memory::parity                  50.0%       100 Uncovered
        covered/total bins:                       1          2
        bin even                                  9          1 Covered
        bin odd                                   0          1 ZERO
    Coverpoint memory::read_write              50.0%       100 Uncovered
        covered/total bins:                       1          2
        bin read                                  9          1 Covered
        bin write                                 0          1 ZERO

TOTAL COVERGROUP COVERAGE: 44.4%  COVERGROUP TYPES: 1
```

Report generator:

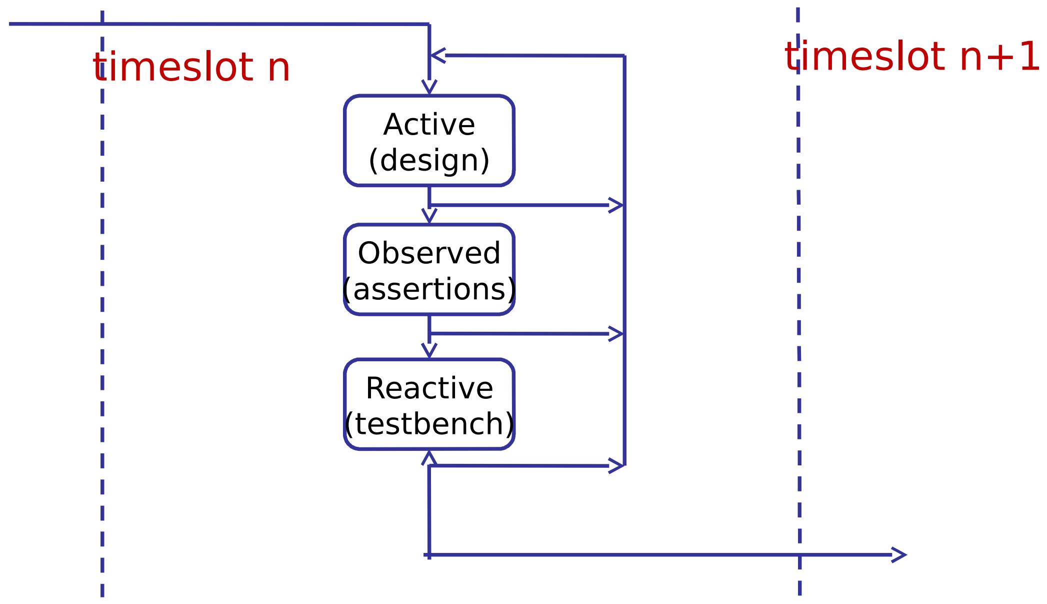
Cross coverage



```
enum { red, green, blue } color;
bit [3:0] pixel_adr;

covergroup g1 @(posedge clk);
  c: coverpoint color;
  a: coverpoint pixel_adr;
  AxC: cross color, pixel_adr;
endgroup;
```

Tom Fitzpatrick, SystemVerilog for VHDL Users, DATE'04

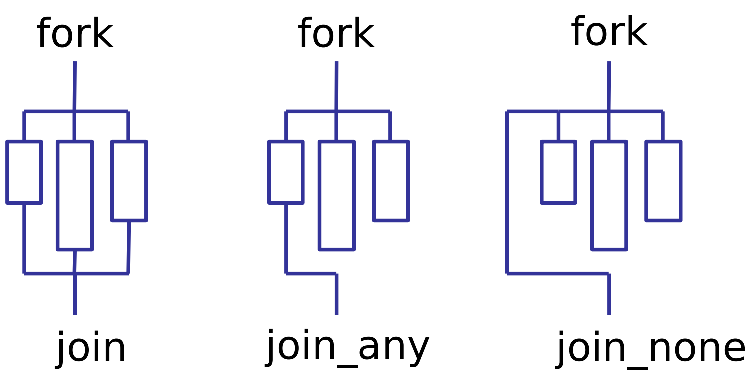
SV enhanced scheduling



Constrained randomization

```
program rc;

class Bus;
   rand bit[31:0] addr;
   rand bit[31:0] data;
   constraint word_align {addr[1:0] == 2'b0;
                          addr[31:24] == 8'h99;}
endclass // Bus

   initial begin
      Bus bus = new;
      repeat (50) begin
     if ( bus.randomize() == 1 )
       $display ("addr = 0x%h   data = 0x%h\n",
                      bus.addr, bus.data);
     else
       $display ("Randomization failed.\n");
      end
   end
endprogram // rc
```

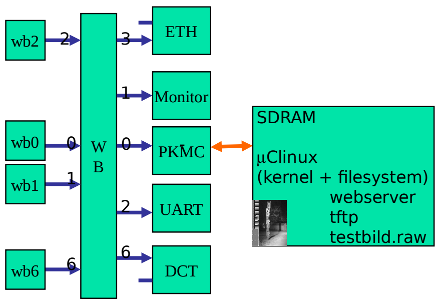
Parallel threads



An example-sketch

- WB arbitration test
  - Instantiate 4 wishbone_tasks

```
program tester2();
  …
  initial begin
     …
     fork
       begin  // 2
         for (int i; i<100; i++)
            jpeg_top_tb.wb2.m_write(32'h100, 32'h0);
       end
       …
       begin  // 6
         for (int i; i<100; i++)
           jpeg_top_tb.wb6.m_write(32'h20000000, result);
       end
       …

     join
     …
  end
endprogram
```



www.liu.se

LINKÖPING
UNIVERSITY