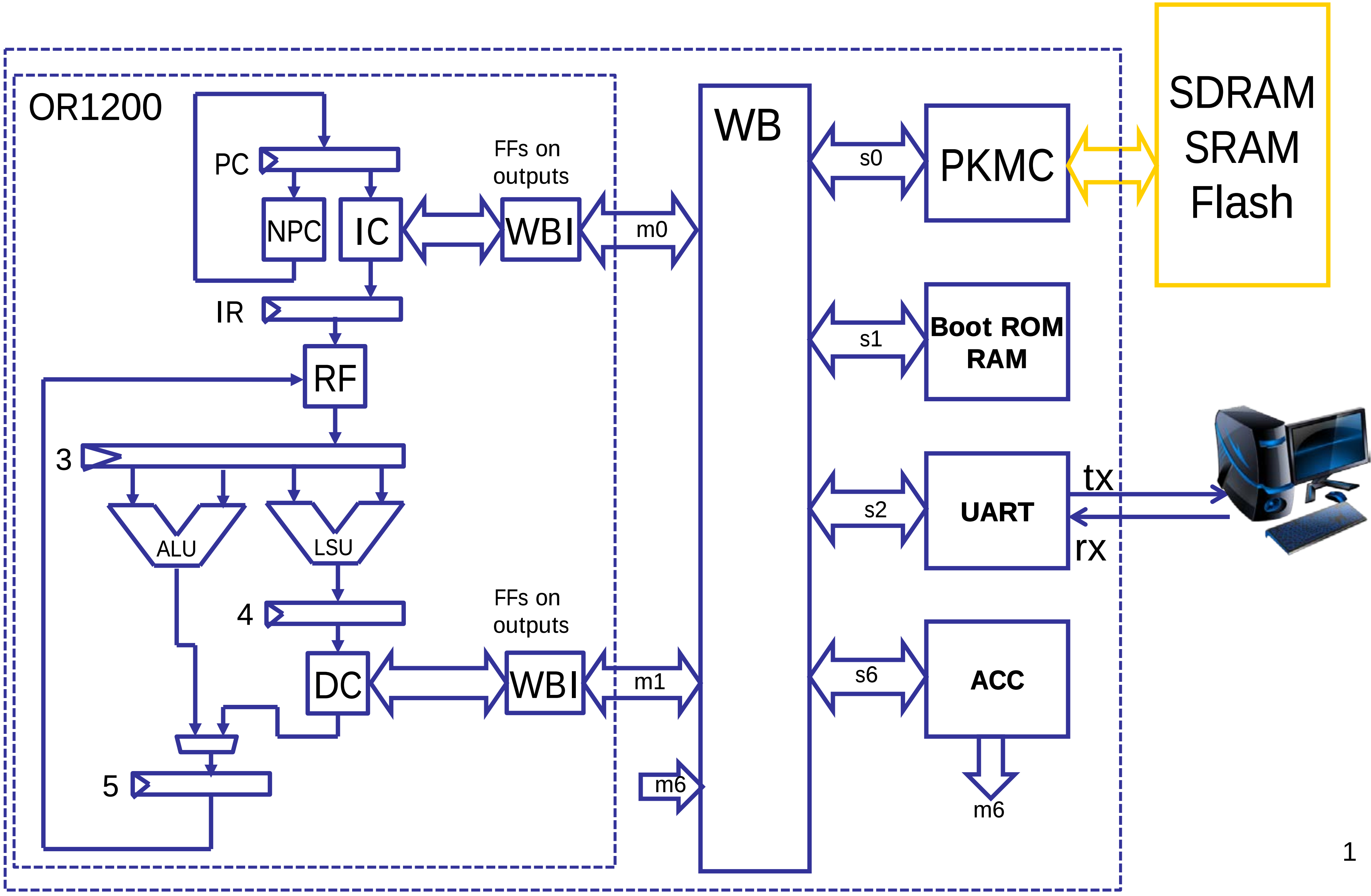
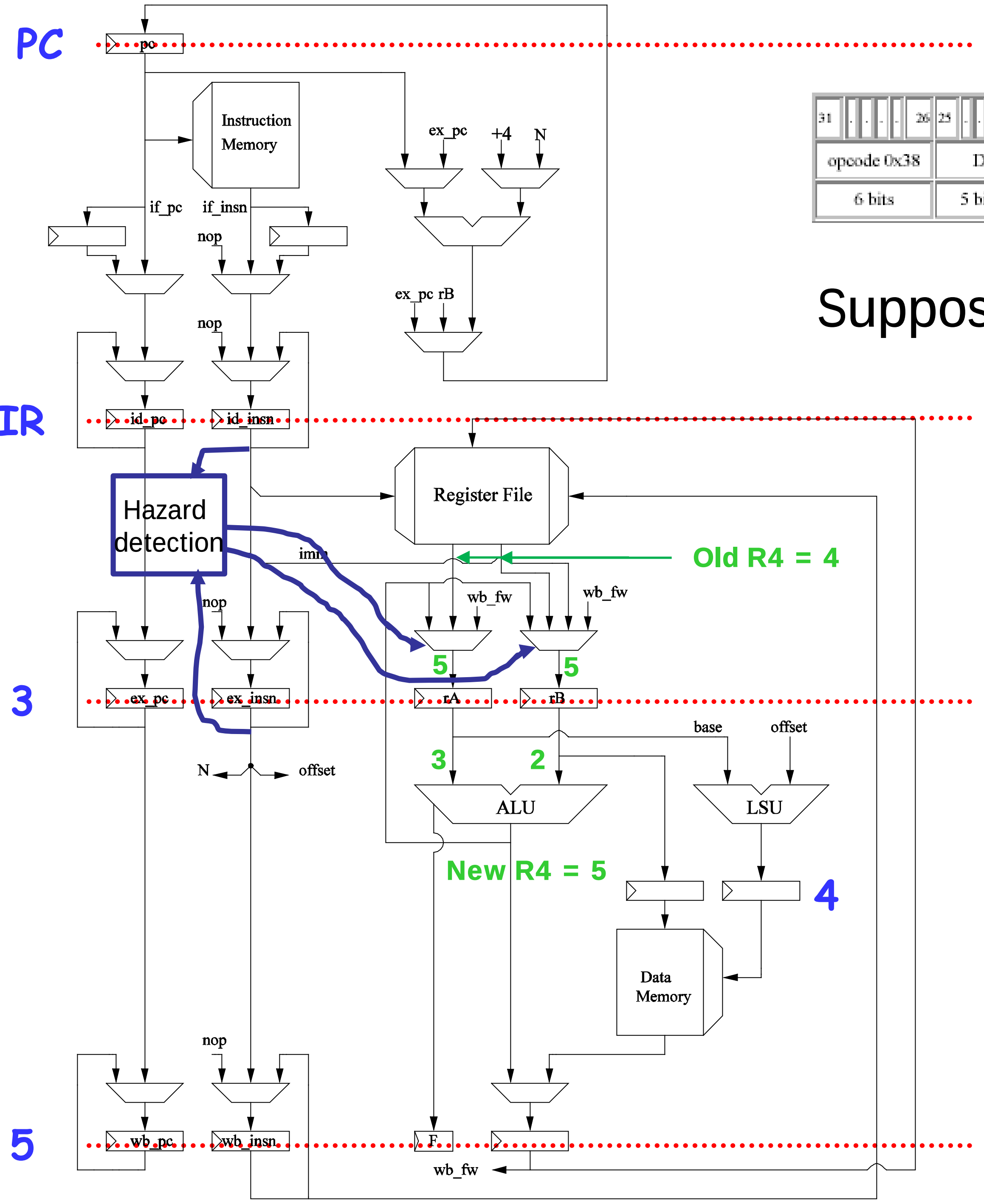
4-Pipeling, Caches,Testbenches

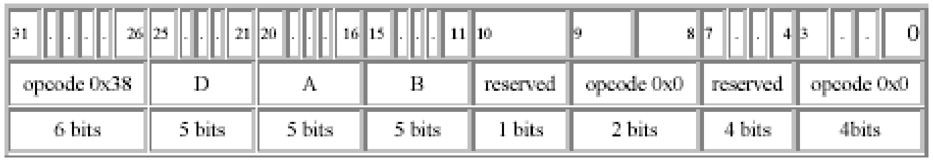


1) Data dependency

RAW = read after write

```
0: add r4,r3,r2

4: add r5,r4,r4
```



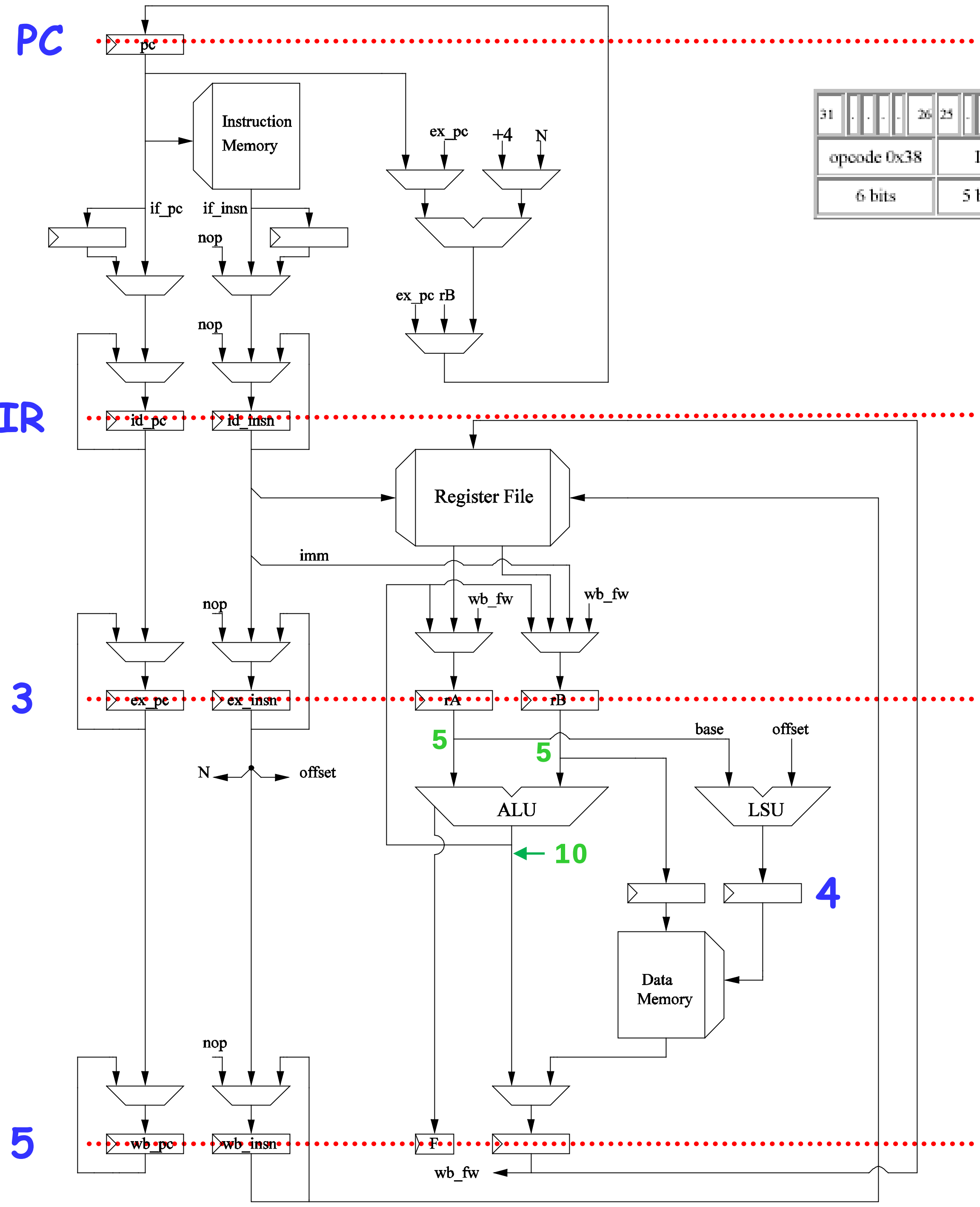


31 26	25 . . . 21	20 . . . 16	15 . . . 11	10	9 8	7 . . 4	3 . . 0
opcode 0x38	D	A	B	reserved	opcode 0x0	reserved	opcode 0x0
6 bits	5 bits	5 bits	5 bits	1 bits	2 bits	4 bits	4bits

Suppose rn contains n

```
add r5,r4,r4
```

```
add r4,r3,r2
```



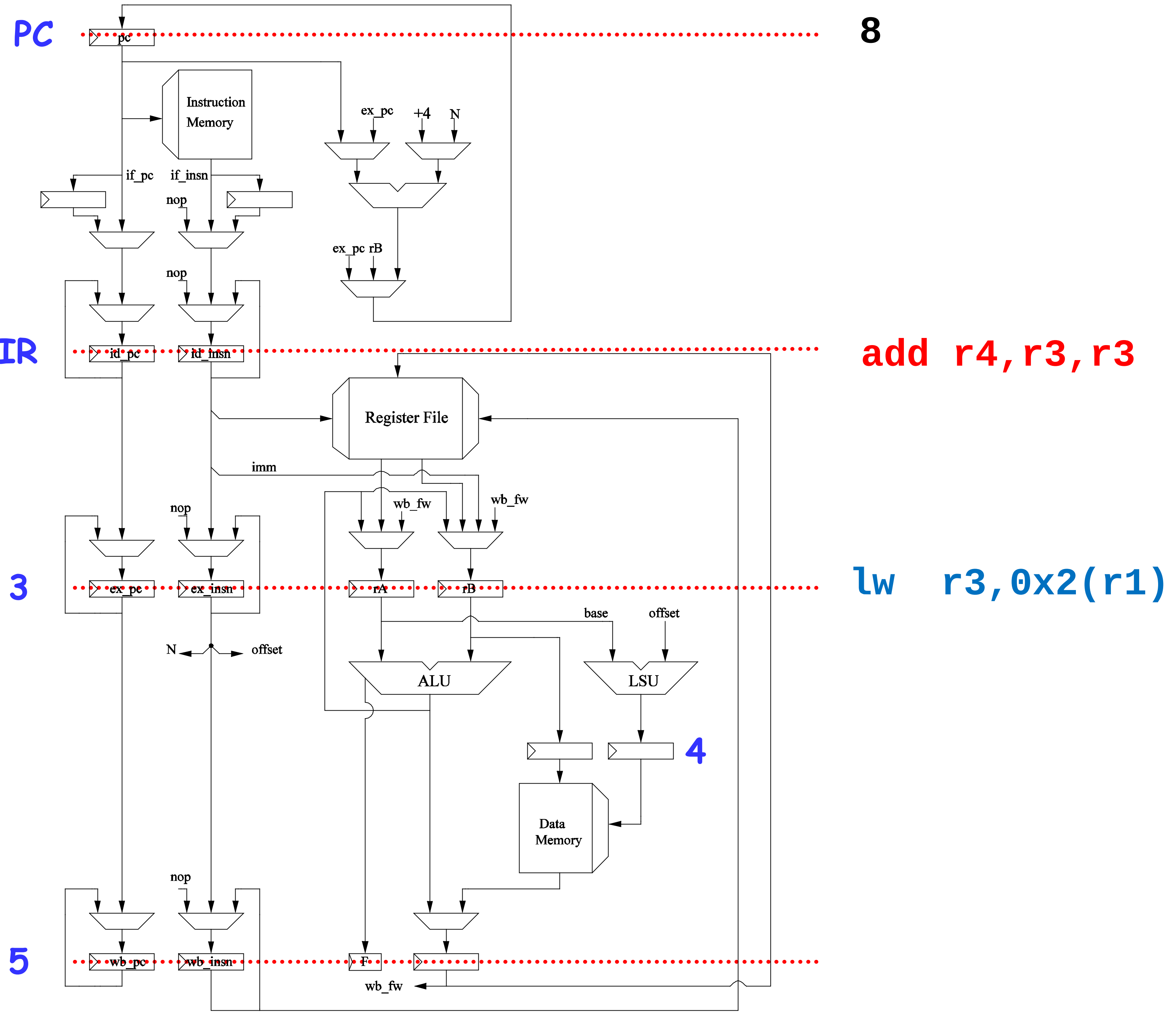
31 … 26	25 … 21	20 … 16	15 … 11	10	9 … 8	7 … 4	3 … 0
opcode 0x38	D	A	B	reserved	opcode 0x0	reserved	opcode 0x0
6 bits	5 bits	5 bits	5 bits	1 bits	2 bits	4 bits	4bits

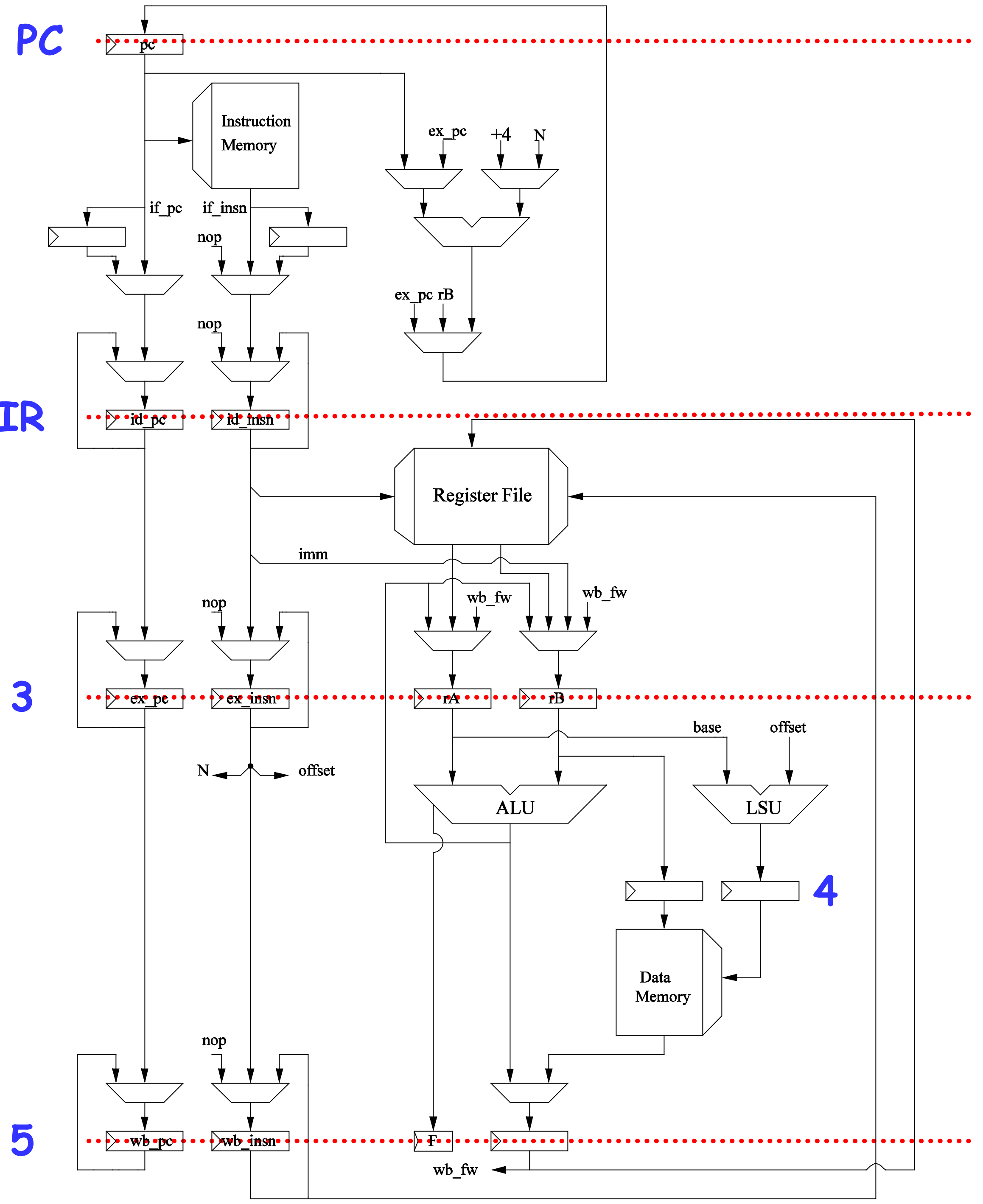
add r5,r4,r4

add r4,r3,r2

2) Loads

```
0: lw  r3,0x2(r1)
4: add r4,r3,r3
```


PC
8
pc
Instruction Memory
ex_pc
+4
N
if_pc
if_insn
nop
ex_pc rB
IR
add r4,r3,r3
id_pc
id_insn
Register File
imm
wb_fw
wb_fw
3
lw r3,0x2(r1)
ex_pc
ex_insn
rA
rB
base
offset
N
offset
ALU
LSU
4
Data Memory
nop
5
wb_pc
wb_insn
F
wb_fw

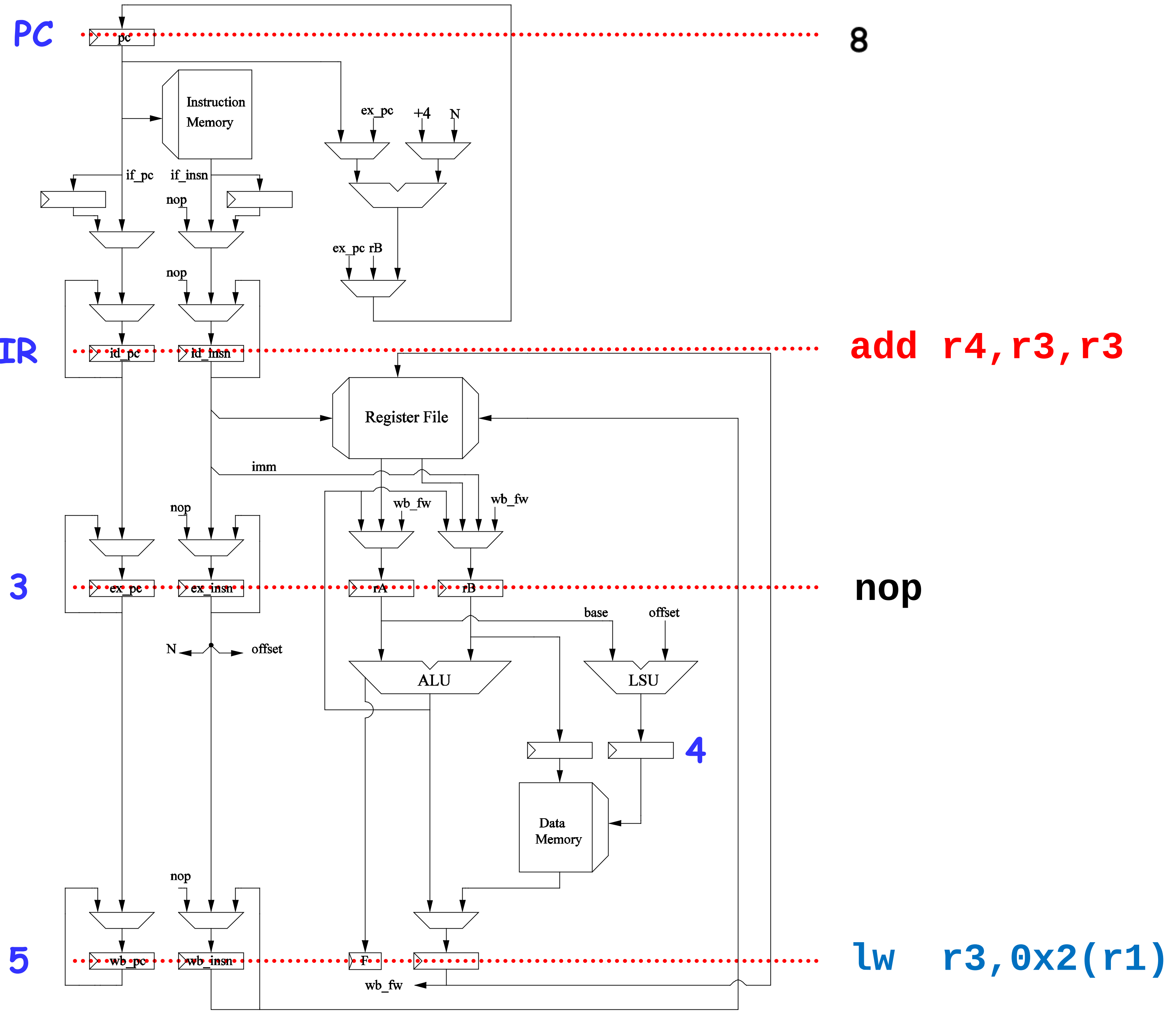


8

add r4,r3,r3

nop

lw r3,0x2(r1)


PC
pc
8
Instruction Memory
ex_pc
+4
N
if_pc
if_insn
nop
ex_pc rB
IR
id_pc
id_insn
add r4,r3,r3
Register File
imm
wb_fw
nop
3
ex_pc
ex_insn
rA
rB
nop
base
offset
N
offset
ALU
LSU
4
Data Memory
nop
5
wb_pc
wb_insn
F
wb_fw
lw r3,0x2(r1)

Caches

- Are essential! Without them we can forget about 1 CPI!
- We want to fetch 1 instruction every clock cycle
  - 24kB Boot RAM 3 CK,
  - 64MB SDRAM 4 CK
- **Size**: depending on the FPGA
  we have 120 x 2KB block RAMs
  => 8kB each IC,DC
  Type: direct mapped (or set associative)
- Cache management special-purpose registers
- Main idea: keep often used data in cache
  - If I visit mem(A) => I will visit mem(A) again
    => I will visit mem(A+1)

Direct mapped cache

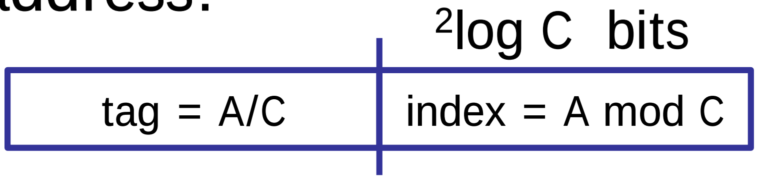
A = address to mem

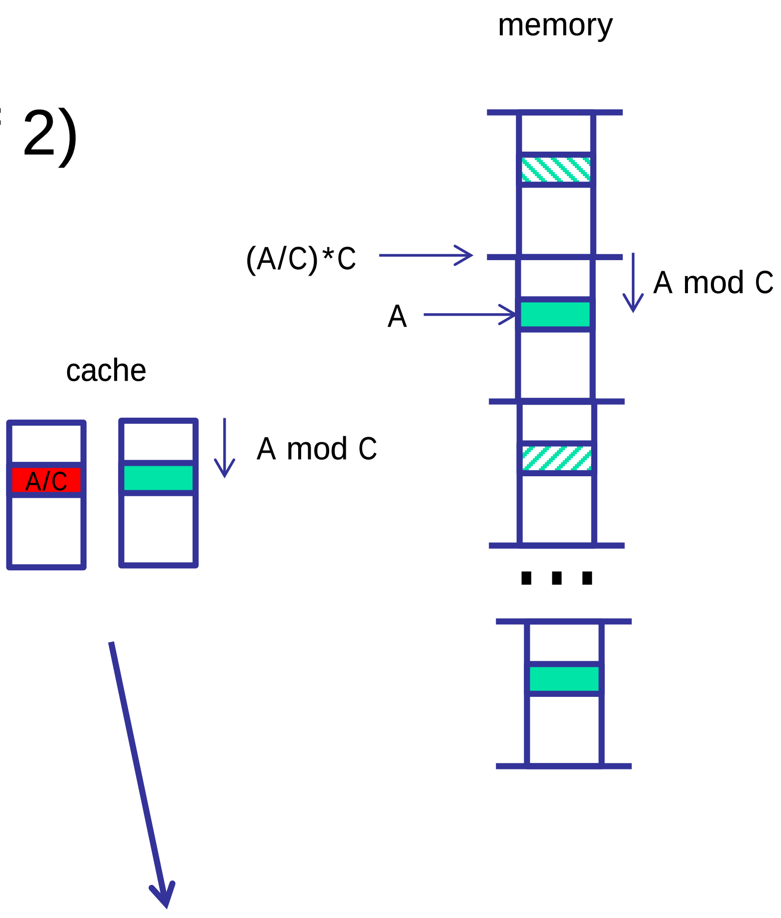
C = cache size (always a power of 2)

The address A can be written

$$A \overset{\text{def}}{=} \left(\frac{A}{C}\right) * C + A \bmod C$$

address:

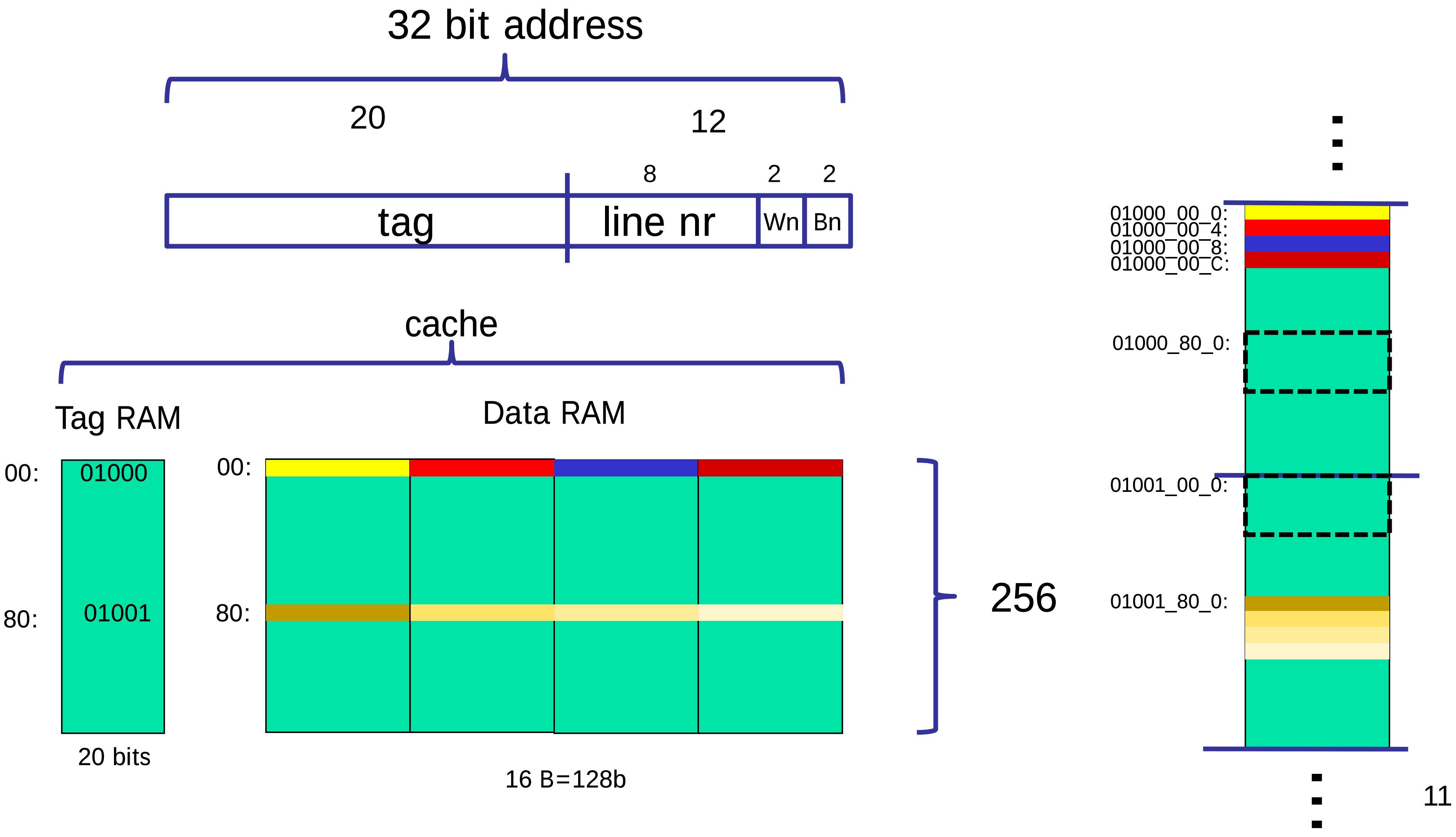




cache(A mod C) = {A/C, mem(A)}

4kB cache example

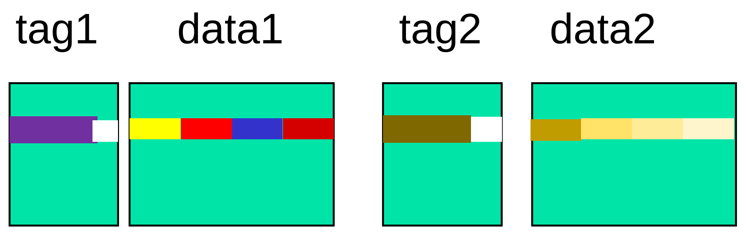
⇒ more than 1 word is fetched on a cache miss
⇒ a cache line is 4 words = 16 bytes
⇒ same size of data RAM, but 256x128

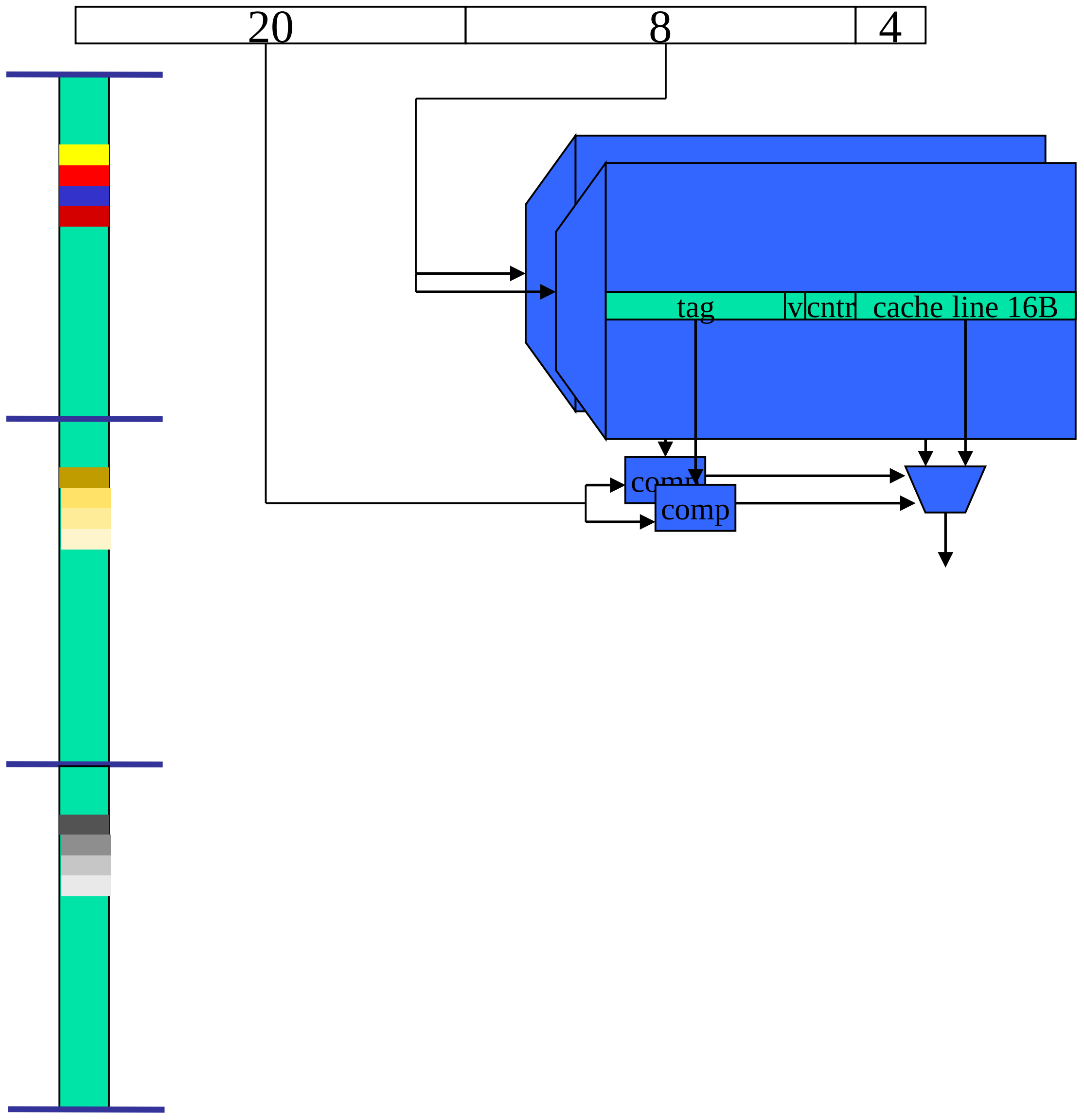


Direct mapped cache - 4kB

2

A 2-way 8kB cache
block size = 4kB



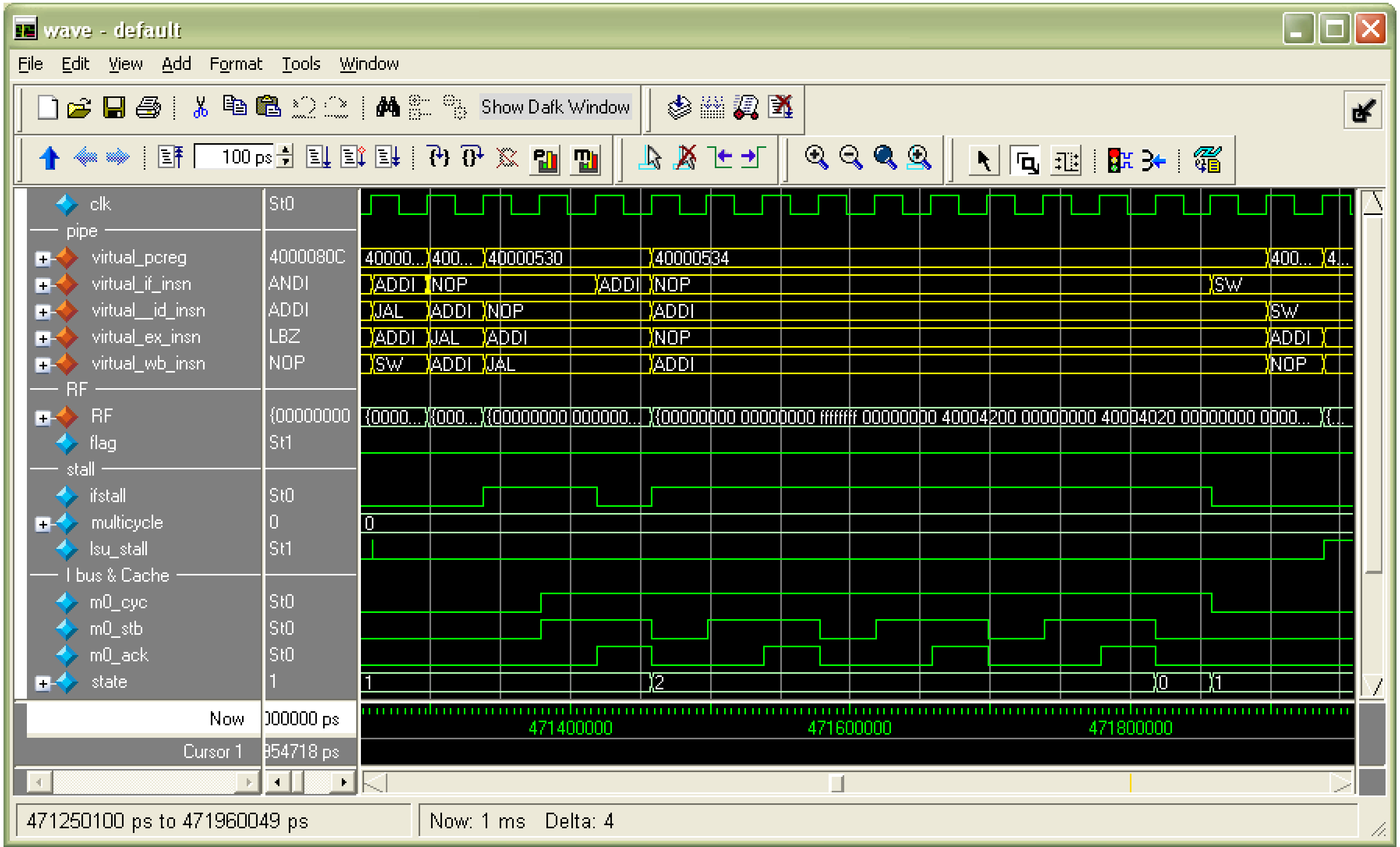


Flags:
valid,
LRU = least recently used

2-3 bit counter per cacheline,
inc when line used

An IC cache miss

```
40000530:    9c 21 ff e4          l.addi r1,r1,0xffffffe4
40000534:    d4 01 48 04          l.sw 0x4(r1),r9
40000538:    d4 01 50 08          l.sw 0x8(r1),r10
```



Cache policy

Cacheline = 4 words = 16B

Instruction cache

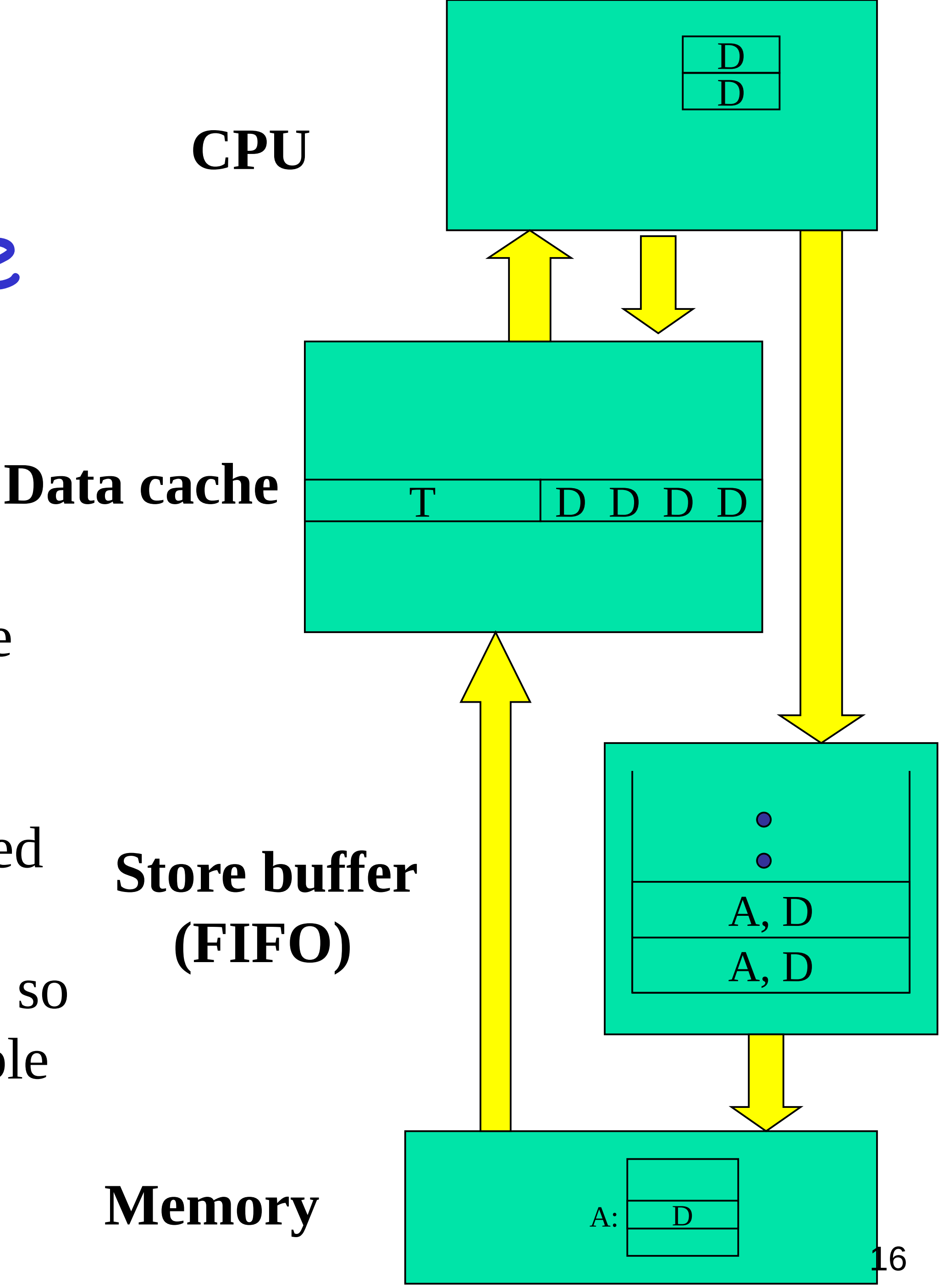
	hit	miss
read	read from cache	fill (replace) cacheline from memory

Data cache

	hit	miss
read	read from cache	fill (replace) cacheline from memory
write	write to cache write thru to memory	write to memory only

or1200 store buffer

- In a write-through data cache every write is equivalent to a cache miss!
- A store (write) buffer is placed between CPU and memory
- Writes are placed in a queue, so that the data cache is available on the next clock cycle



or1200 store buffer

read hit: read from cache

read miss: a potential data hazard! The data might be in the SB

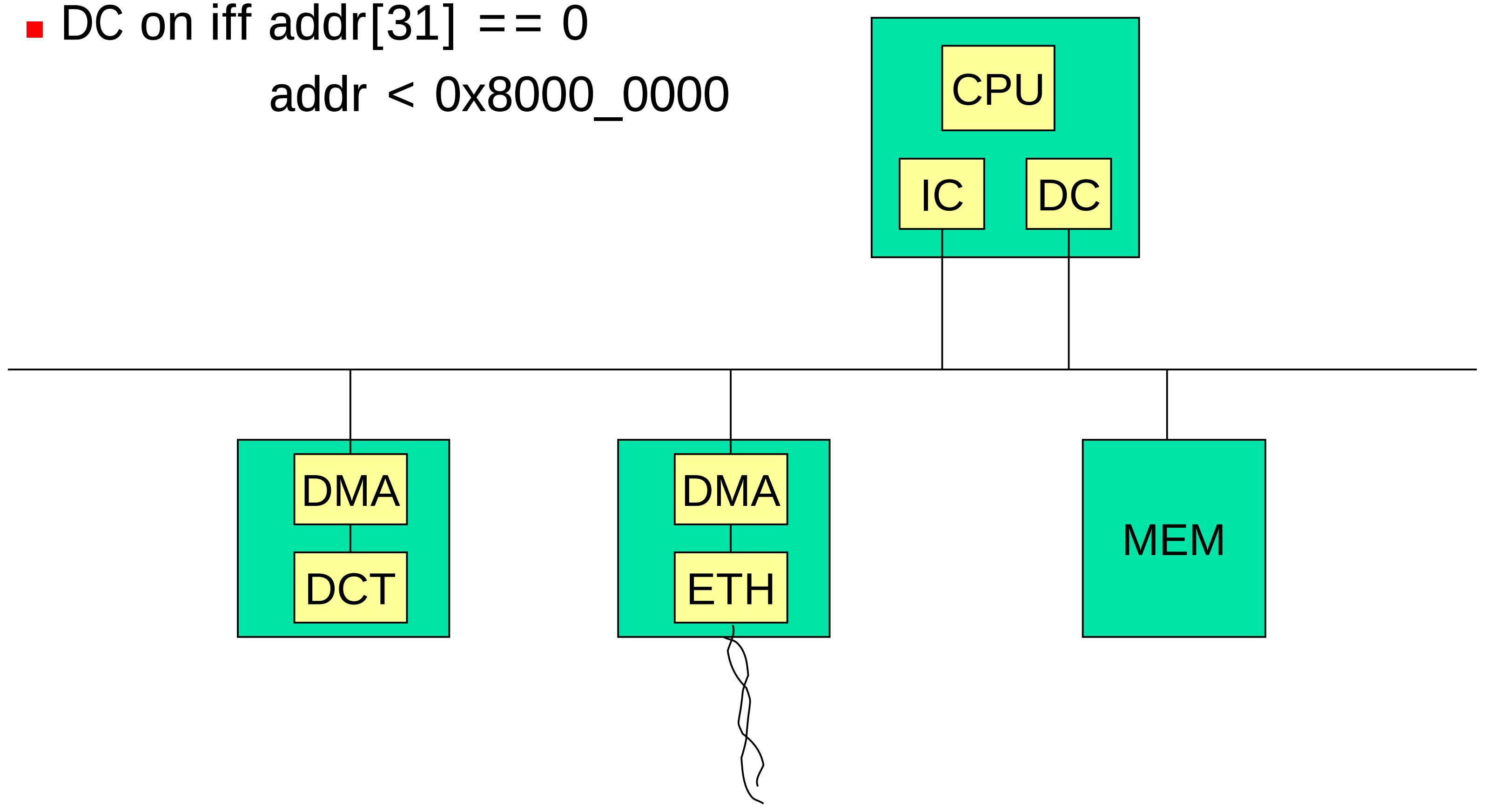
1) wait until SB is empty.
Then do a cache line refill.
No benefit from the SB in this case!

2) update DC from SB. SB must be
designed like a cache (probably associative)

Watch out!

- Caches can be incoherent when using DMA.
- Parts of memory should be non-cacheable
  - IC on for all addresses
  - DC on iff addr[31] == 0

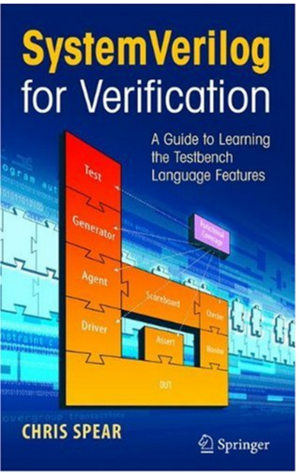
    addr < 0x8000_0000

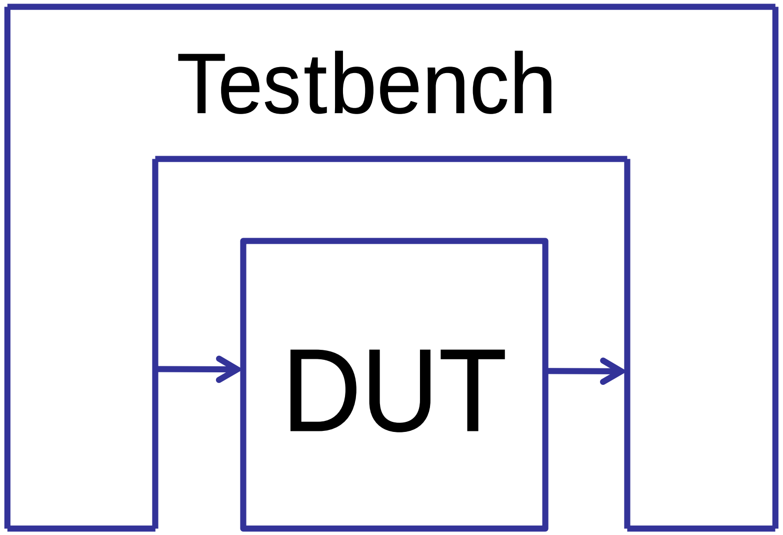


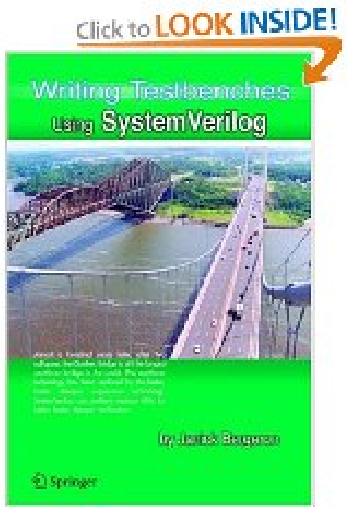
Testbenches

Spear,Chris:
System Verilog for verification.
Springer

Bergeron,Janick:
Writing testbenches using System Verilog.
Springer







Testbenches

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

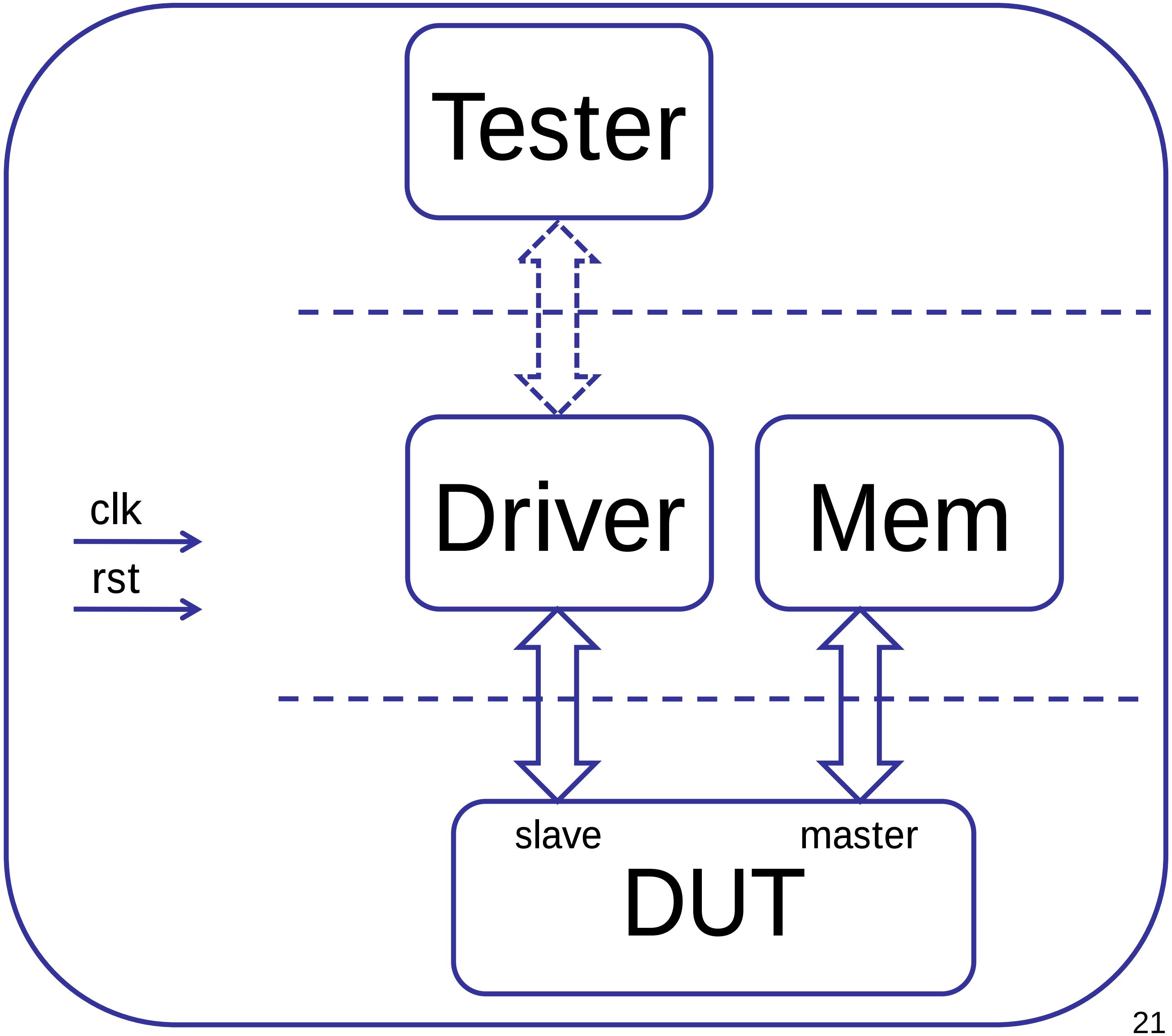
Like High-Level Software
(very different from DUT)
- easy to design
- easy to test timing
- easy to test flow

An example : a TB for your design

Hi-level tester

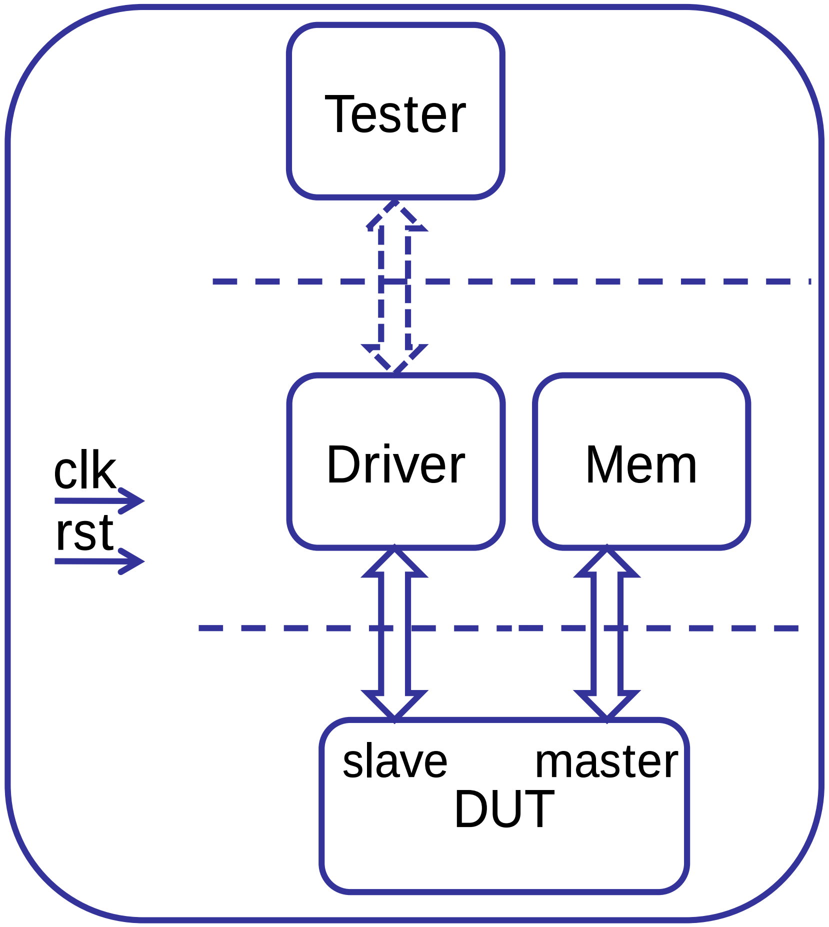
Wishbone BFM

JPEG AX



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
```



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
```

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
```

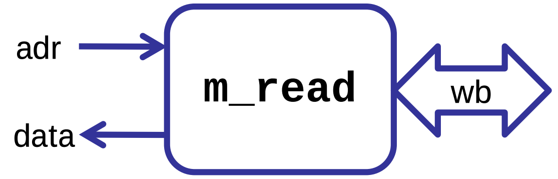
DMA? Easy!

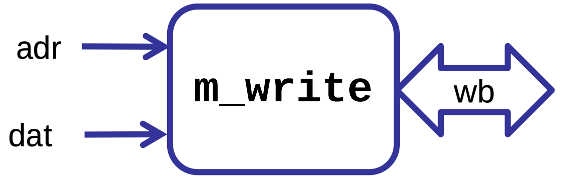
```
…
// Init DMA-engine
      jpeg_top_tb.wb0.m_write(32'h96001800, 32'h0);
      jpeg_top_tb.wb0.m_write(32'h96001804, ?);
      jpeg_top_tb.wb0.m_write(32'h96001808, ?);
      jpeg_top_tb.wb0.m_write(32'h9600180c, ?);
      jpeg_top_tb.wb0.m_write(32'h96001810, ?);                   // start DMA
engine


      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
```

`program` block

- Purpose: Identifies verification code
- A **`program`** is different from a **`module`**
  - only initial blocks allowed
  - executes last
    (module -> clocking/assertions -> program)

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

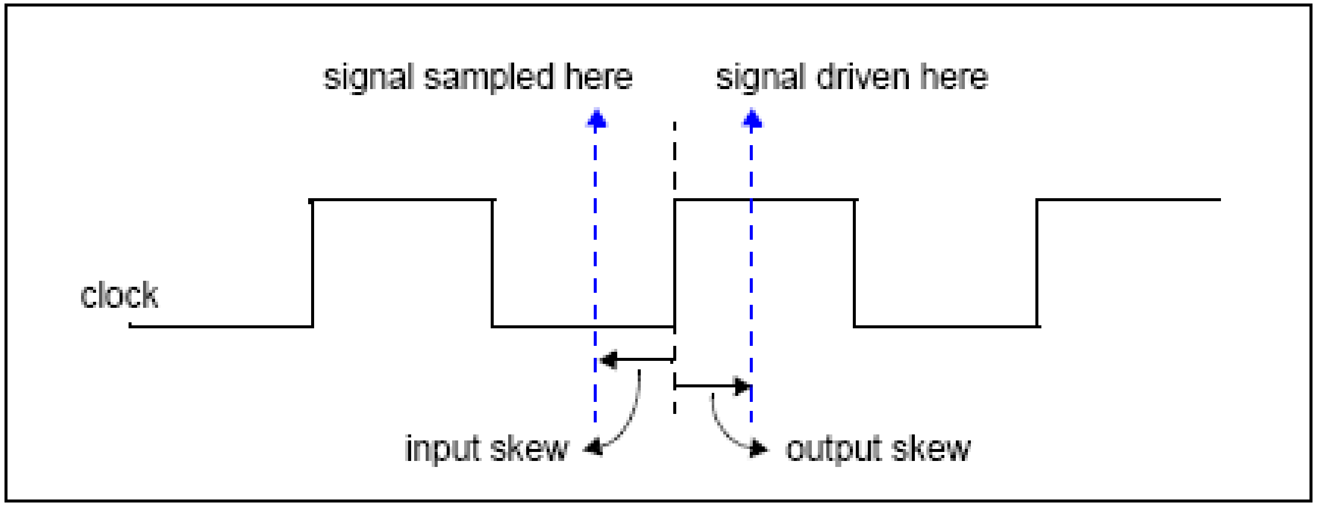
clocking block

SystemVerilog adds the **clocking block that identifies clock signals, and captures 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 a 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



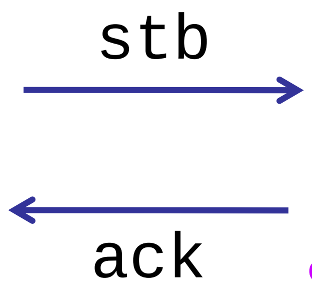
clocking block

```
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

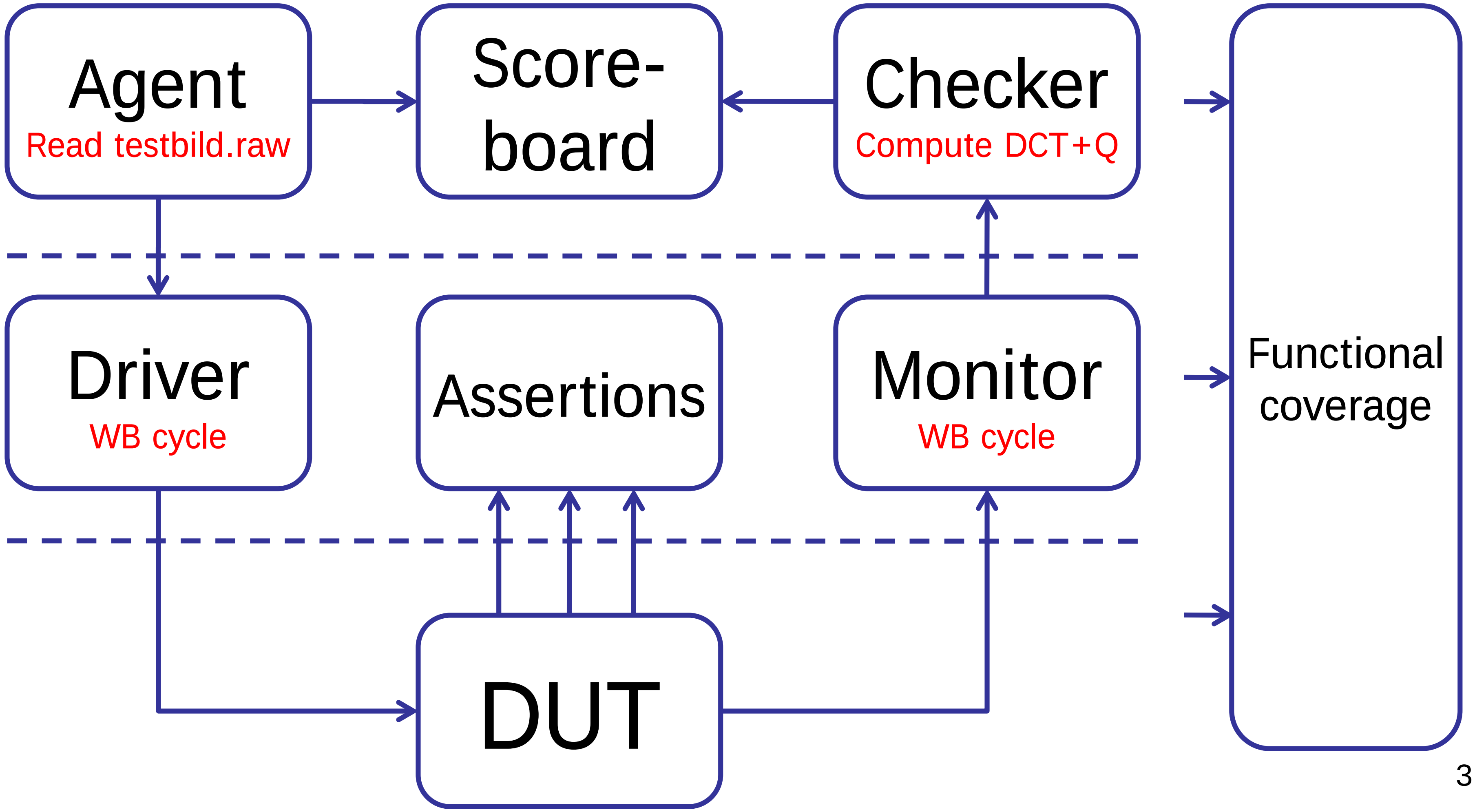
```
program tester(wishbone.tb wb);

   …

   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

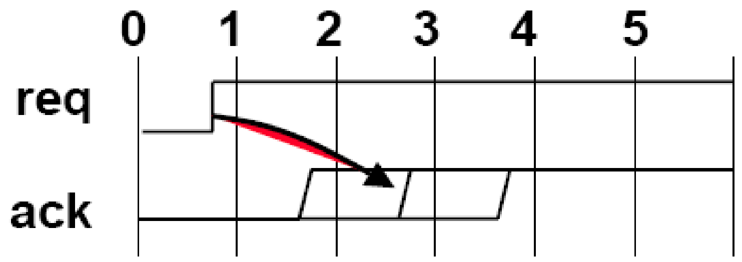
OOP

```
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



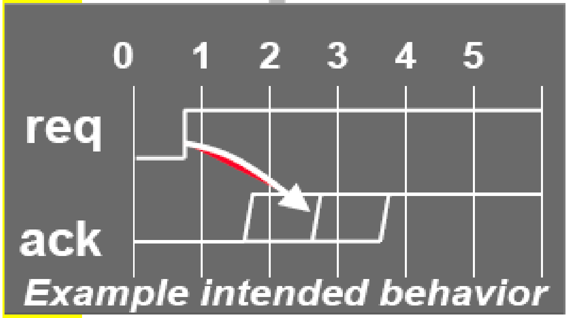
Example intended behavior

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

Assertions

SVA Assertion

```
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;
```

HDL Assertion

Assertions

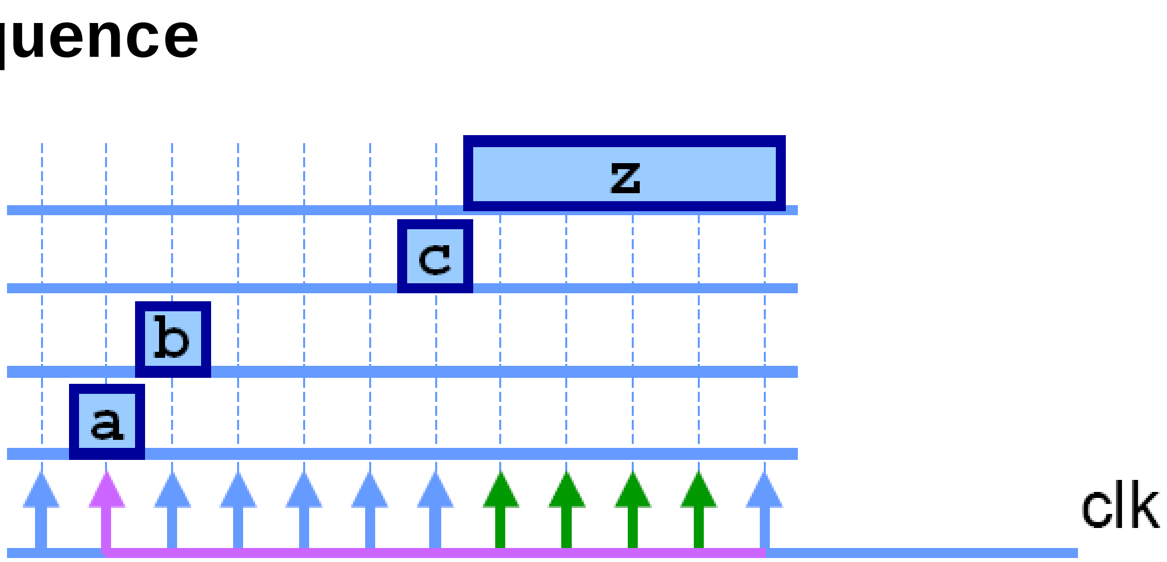
Assertions are built of

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
```



- signal
- expression
- sequence

Properties

- Declares 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 erroneous 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

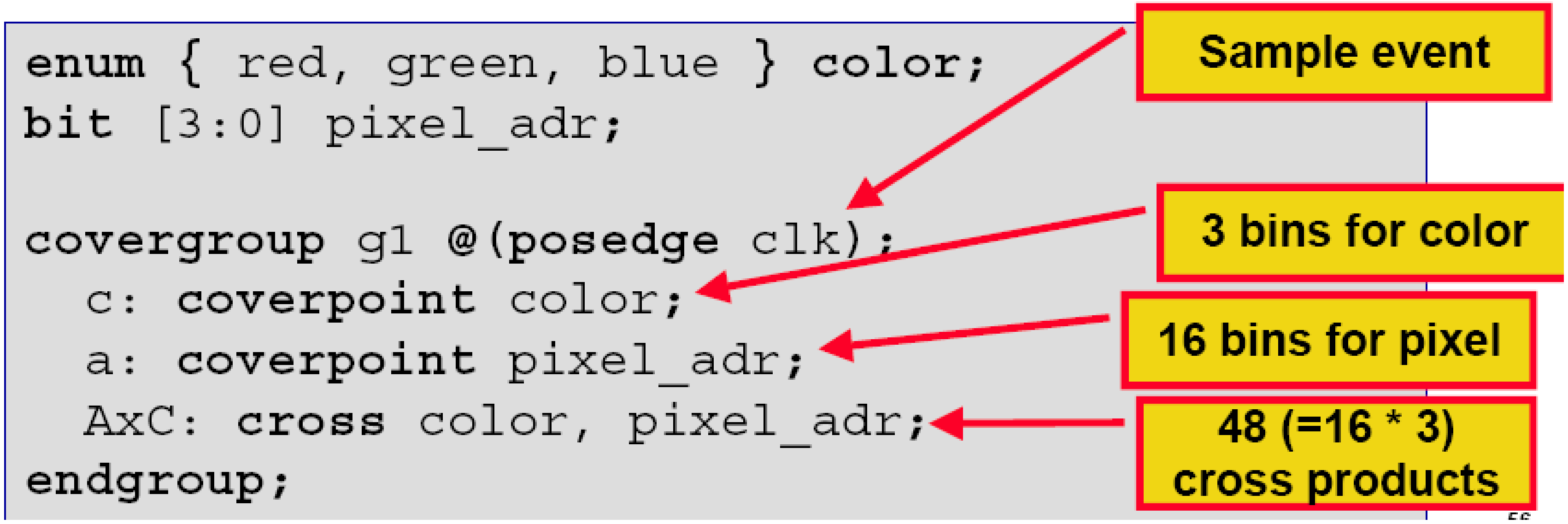
ModelSim says:

```
# @ 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
# @85ns Address :232 data c5, rw 0, parity 0
# @95ns Address :189 data 2d, rw 1, parity 0
```

Report generator:

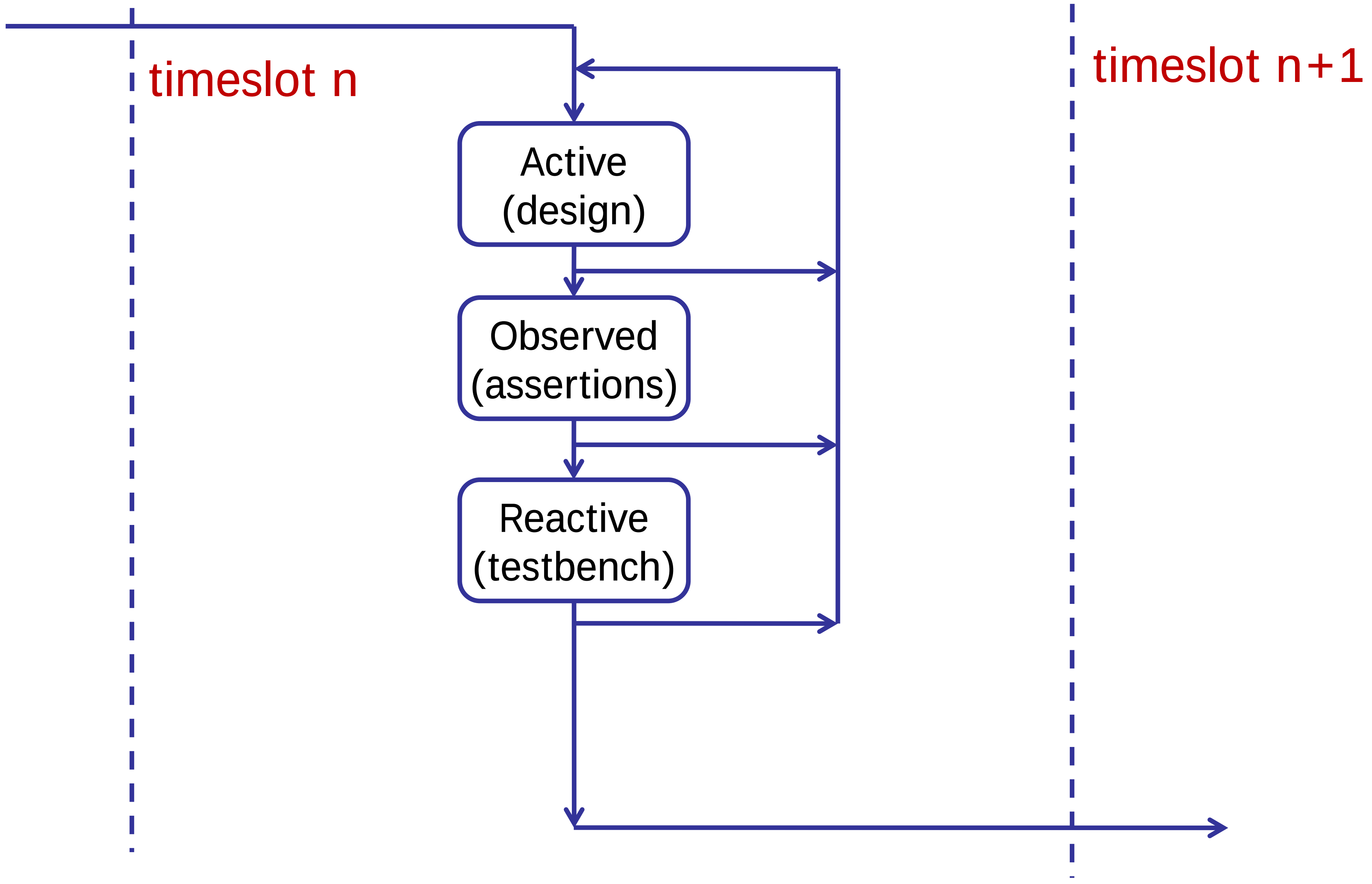
```
COVERGROUP COVERAGE:
------------------------------------------------------------------------------------
Covergroup                                  Metric     Goal/  Status
                                                      At Least
------------------------------------------------------------------------------------
 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
```

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;
```

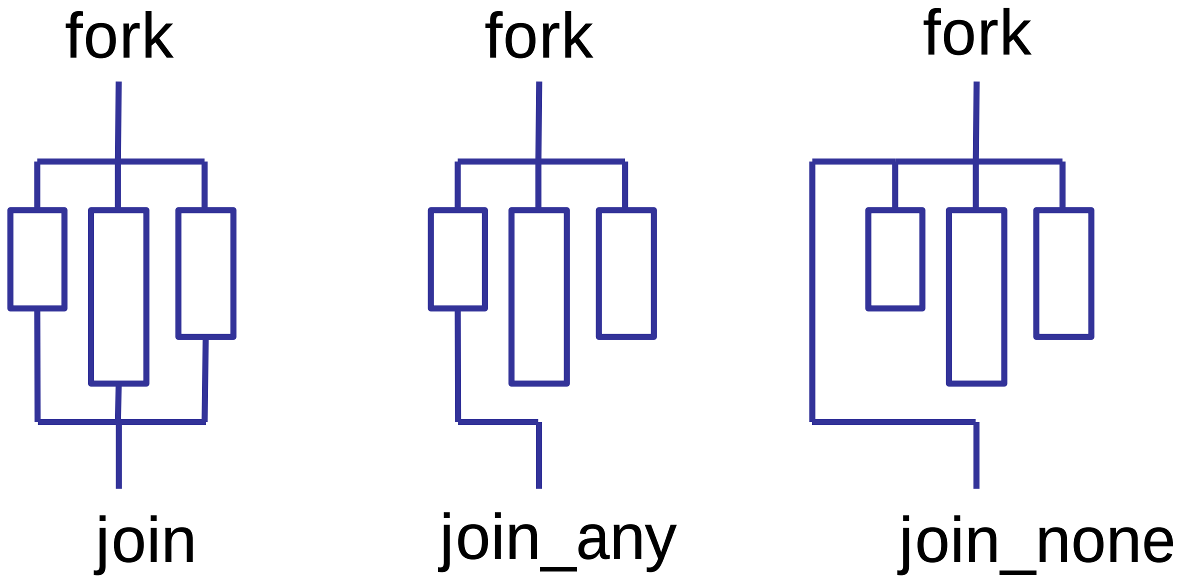
SV enhanced scheduling



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
```

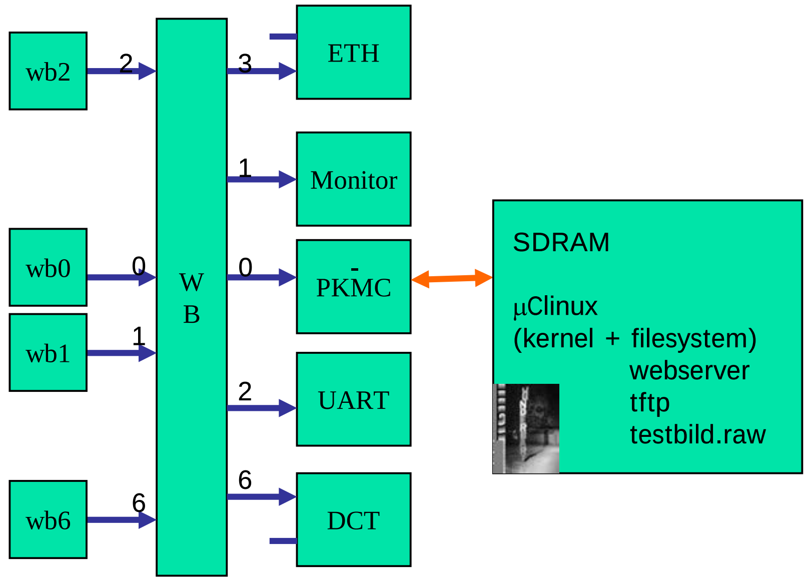
Parallel threads



An example- sketch

WB arbitration test

instantiate 4 wishbone_tasks
(must be declared `automatic`)



```
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
```