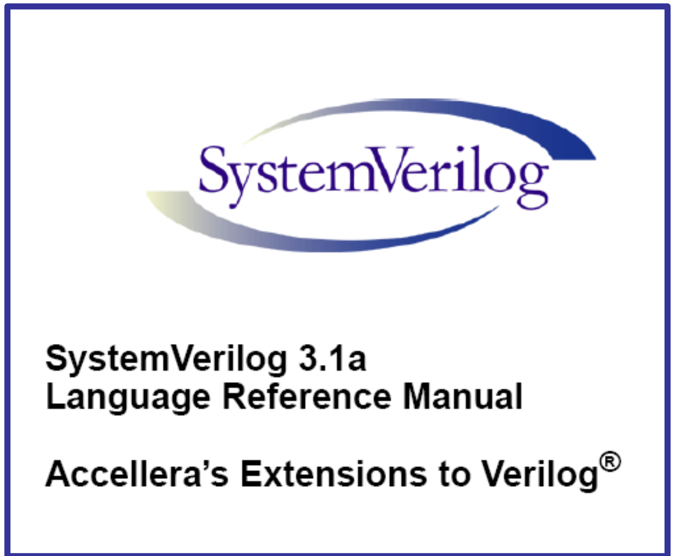
A short introduction to SystemVerilog

- For those who know VHDL
- We aim for synthesis

Verilog & SystemVerilog

- 1984 – Verilog invented, C-like syntax
- First standard – Verilog 95
- Extra features – Verilog 2001
- A super set - SystemVerilog

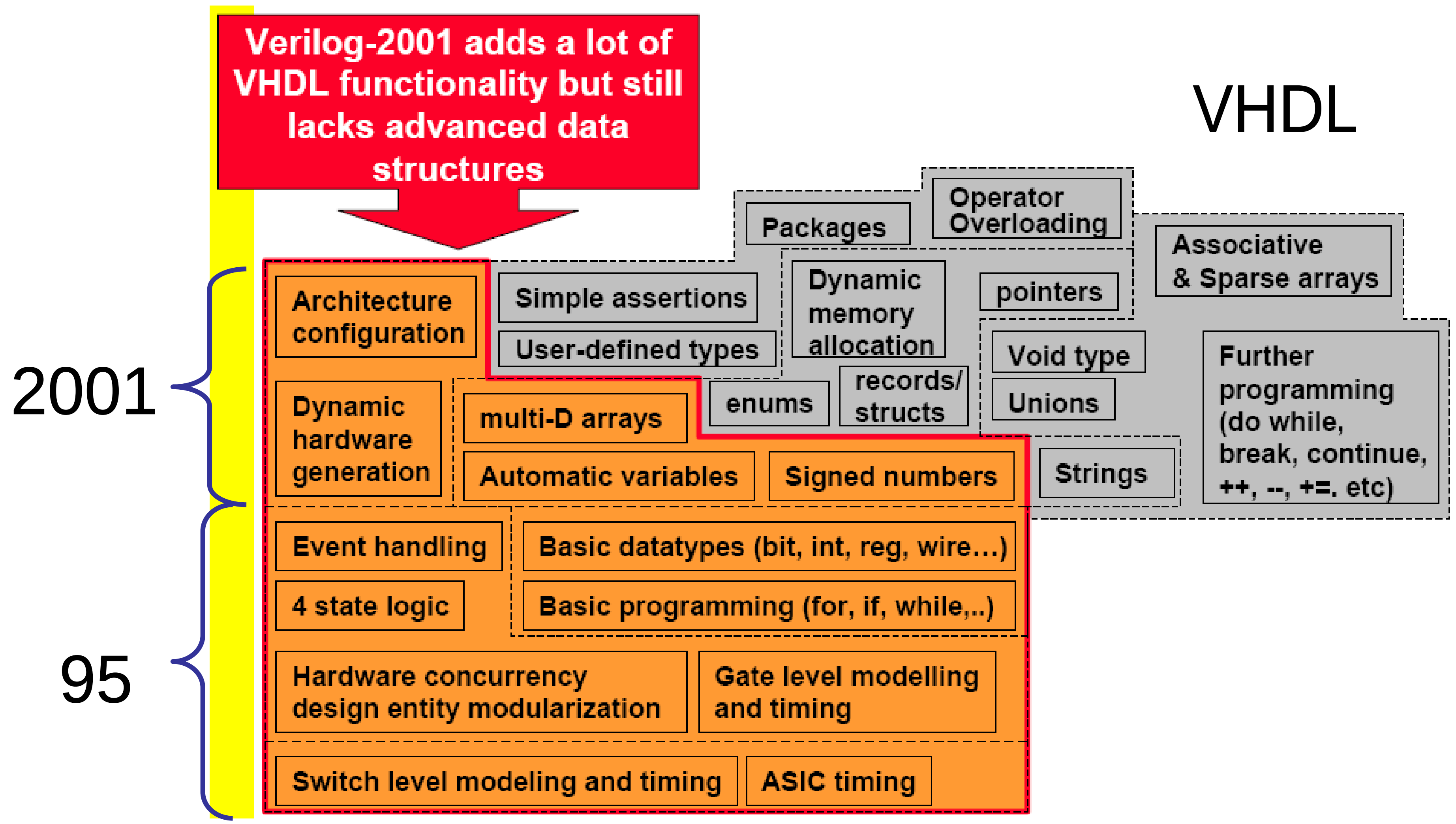
www.vhdl.org/sv/SystemVerilog_3.1a.pdf



SystemVerilog 3.1a
Language Reference Manual

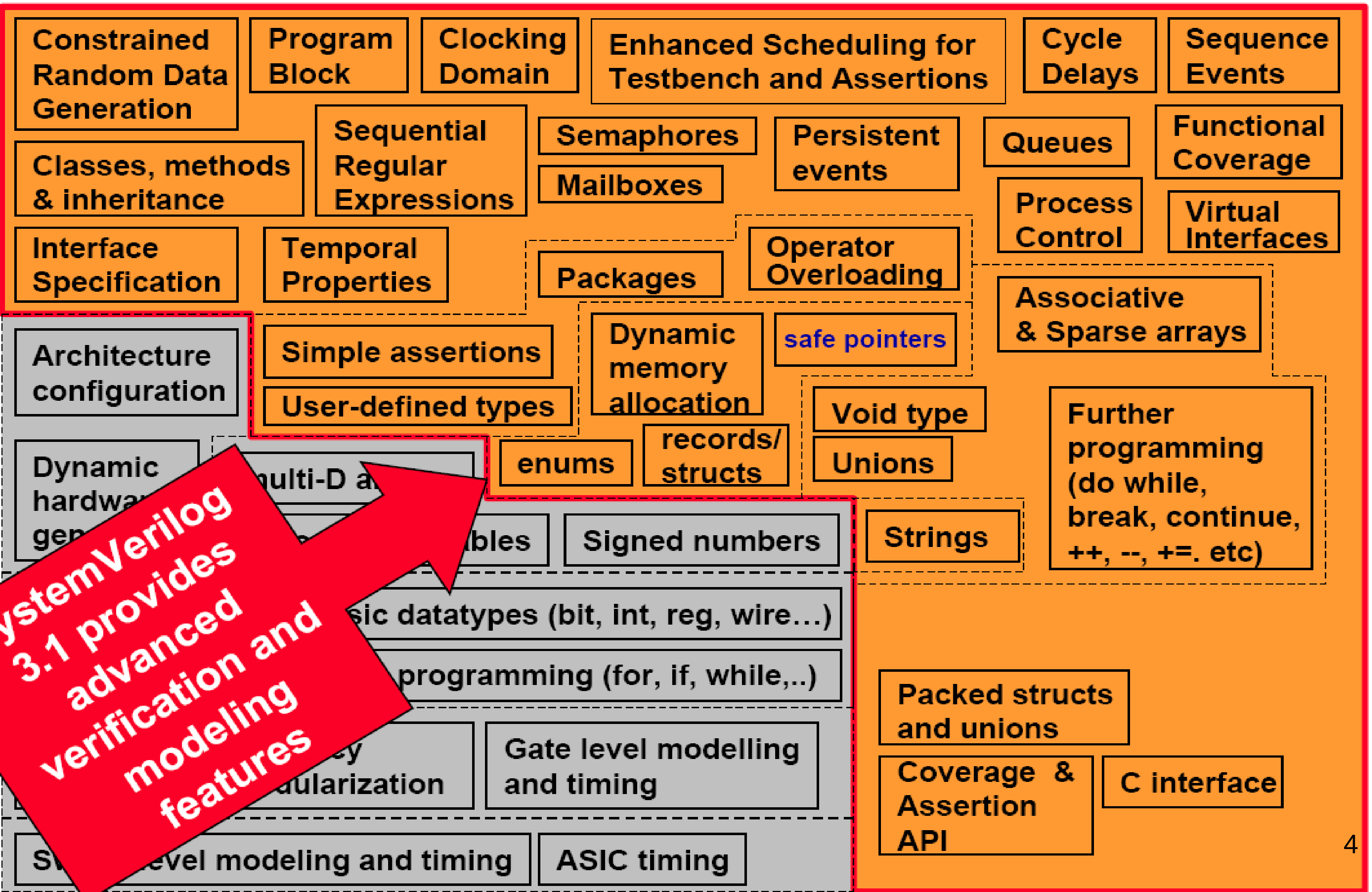
Accellera's Extensions to Verilog®

Verilog vs VHDL



Verilog-2001 adds a lot of VHDL functionality but still lacks advanced data structures
VHDL
2001
95
Architecture configuration
Simple assertions
User-defined types
Dynamic hardware generation
multi-D arrays
Automatic variables
Signed numbers
Packages
Operator Overloading
Dynamic memory allocation
pointers
Associative & Sparse arrays
enums
records/ structs
Void type
Unions
Strings
Further programming (do while, break, continue, ++, --, +=. etc)
Event handling
Basic datatypes (bit, int, reg, wire…)
4 state logic
Basic programming (for, if, while,..)
Hardware concurrency design entity modularization
Gate level modelling and timing
Switch level modeling and timing
ASIC timing

SystemVerilog



Constrained Random Data Generation
Program Block
Clocking Domain
Enhanced Scheduling for Testbench and Assertions
Cycle Delays
Sequence Events
Classes, methods & inheritance
Sequential Regular Expressions
Semaphores
Mailboxes
Persistent events
Queues
Functional Coverage
Process Control
Virtual Interfaces
Interface Specification
Temporal Properties
Packages
Operator Overloading
Associative & Sparse arrays
Architecture configuration
Simple assertions
User-defined types
Dynamic memory allocation
safe pointers
Void type
Unions
Further programming (do while, break, continue, ++, --, +=. etc)
Dynamic hardwa
gen
ulti-D a
enums
records/ structs
bles
Signed numbers
Strings
sic datatypes (bit, int, reg, wire…)
programming (for, if, while,..)
y
ularization
Gate level modelling and timing
Packed structs and unions
Coverage & Assertion API
C interface
S
evel modeling and timing
ASIC timing
ystemVerilog 3.1 provides advanced verification and modeling features

An example: PN check digit

A **PN** consists of 10 digits

$$\boxed{d_1d_2d_3d_4d_5d_6d_7d_8d_9} \longrightarrow d_{10}$$

d_{10} is a check digit computed by the algorithm

$$d_{10} = (10 - (d_1 + \underset{\nwarrow}{[2d_2]} + d_3 + \ldots + d_9)) \bmod 10$$

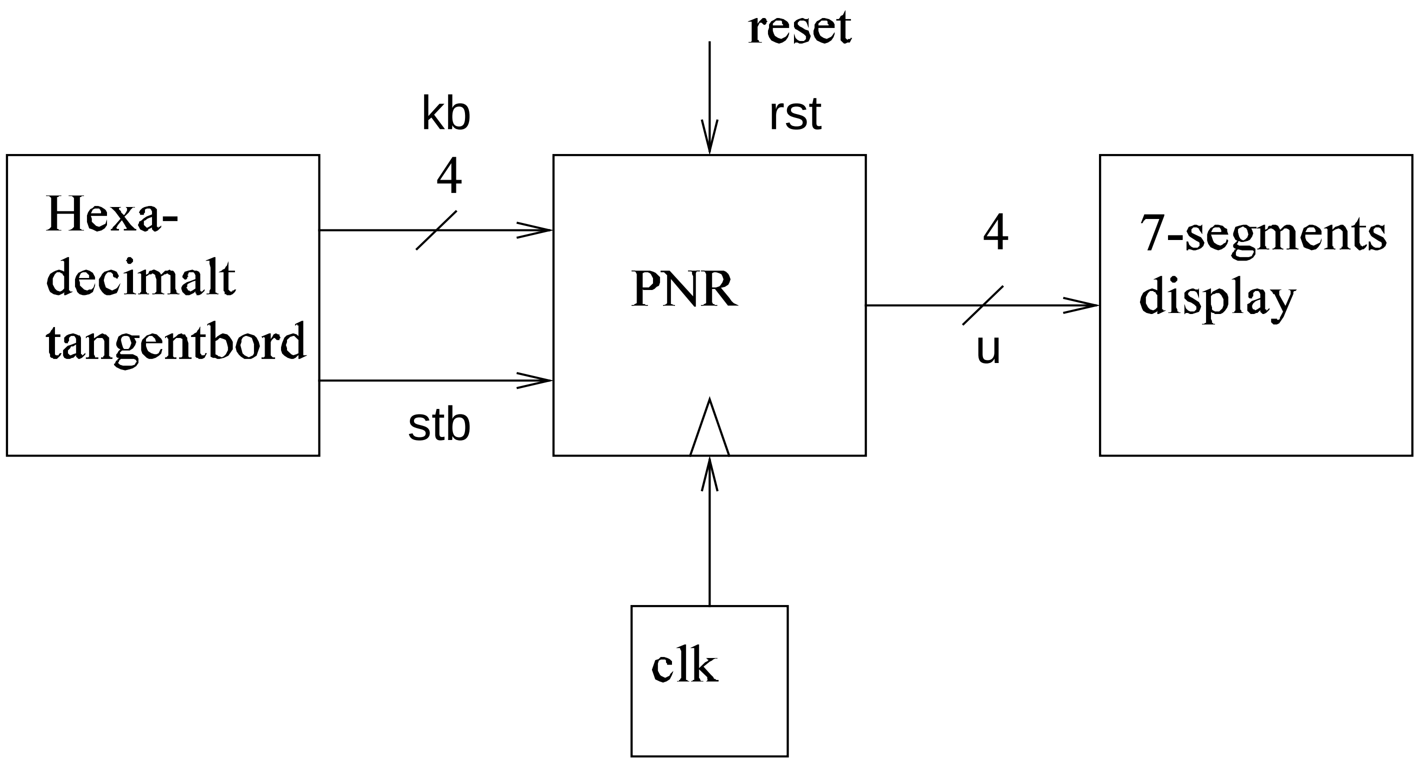
digit sum

$$S^{(0)} = 0$$

$$S^{(k)} = S^{(k-1)} \oplus_{10} X2(d_k) \quad k = 1\ldots9$$

$$d_{10} = 10 - S_9$$

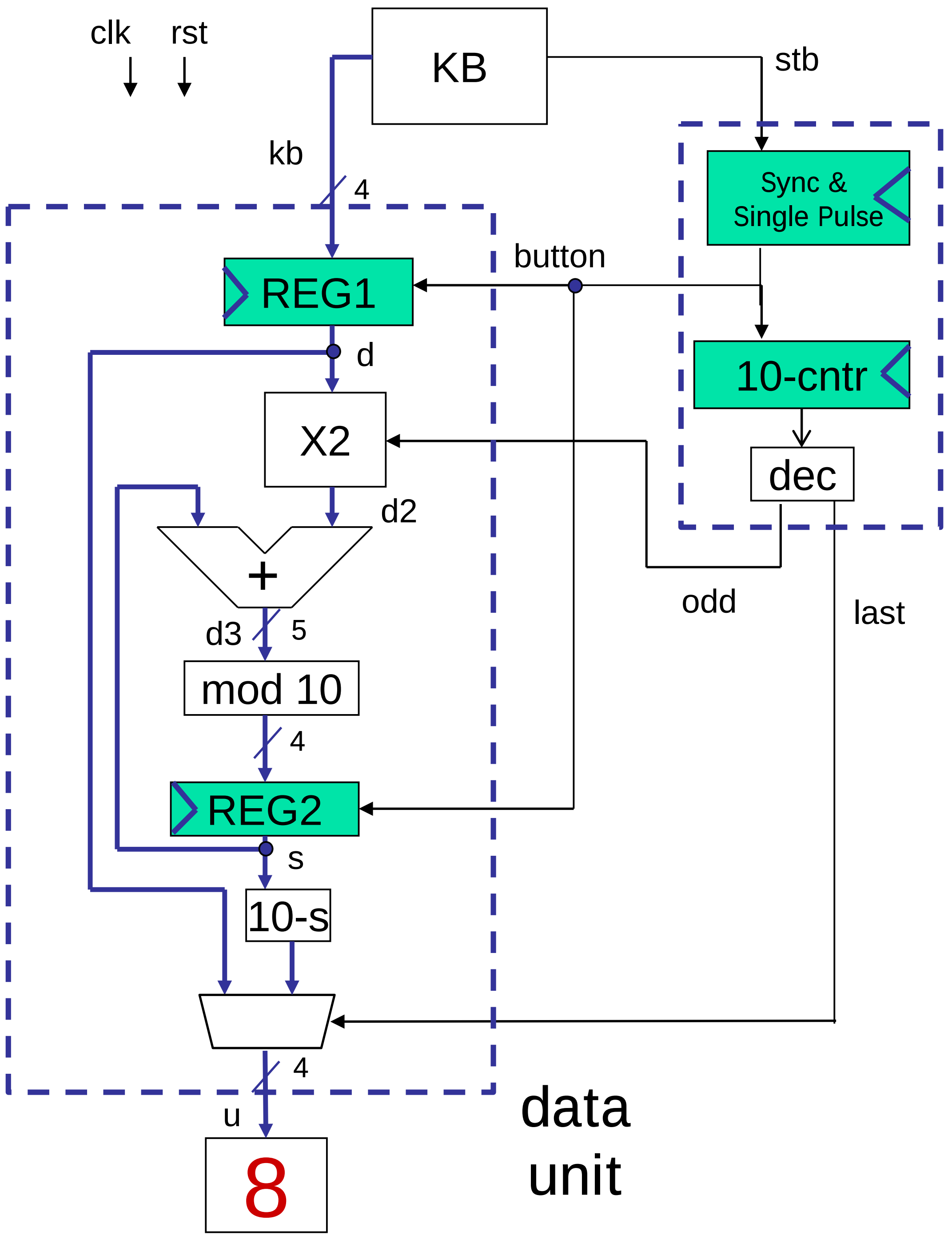
We want to design the block PNR



Top module

```
                          `timescale 1ns / 1ps
                                      time unit  precision
`include "timescale.v“

module pnr(input clk,rst,stb,
           input [3:0] kb,
           output [3:0] u);


// our design



endmodule
```

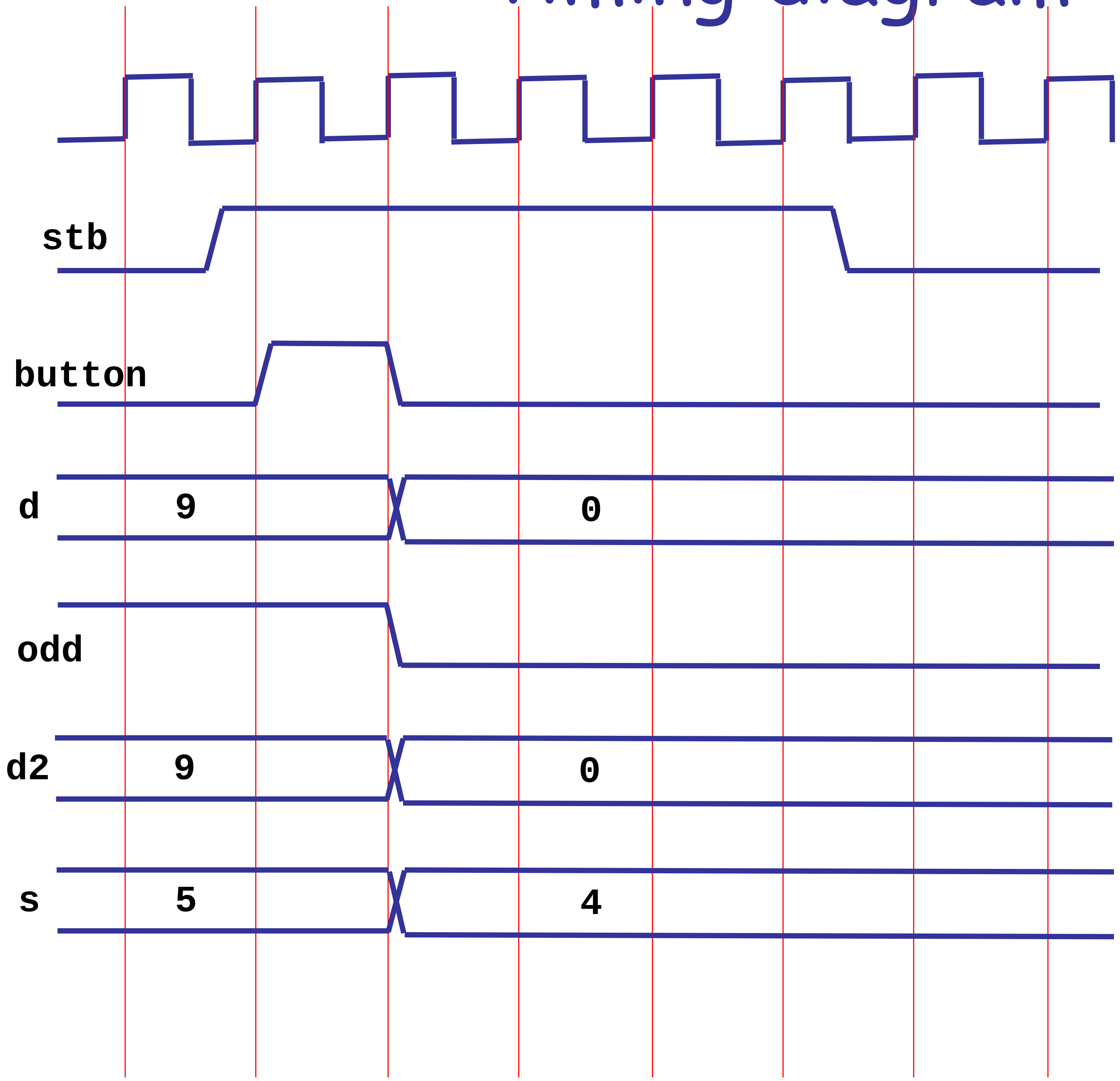
* No entity/architecture distinction => just module



schematics

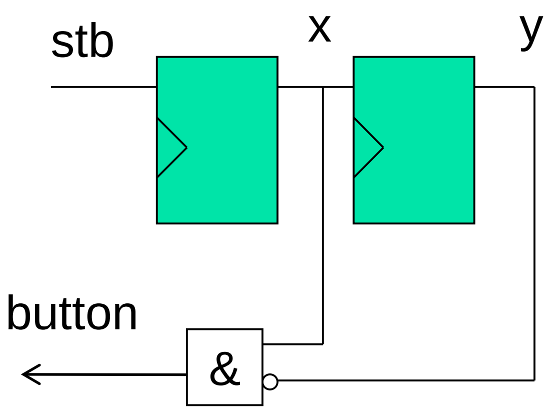
- Green boxes: synch FSM
- White boxes: Comb
- button is singlepulsed

Timing diagram



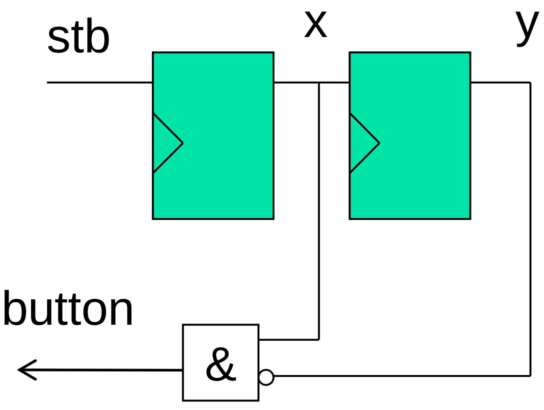
Synch and Single Pulse

```
reg x,y;     // variable type  (0,1,Z,X)
wire button; // net type (0,1,Z,X)

// SSP
   always @(posedge clk)        // procedural block
   begin
       x <= stb;
       y <= x;
   end

  assign button = x & ~y;     //continuous assignment
```



Is this the same thing?

```
reg x,y;      // variable type  (0,1,Z,X)
wire button; // net type (0,1,Z,X)

// SSP
 always @(posedge clk)          // procedural block
   begin
       x <= stb;
   end

 always @(posedge clk)          // procedural block
   begin
       y <= x;
   end

  assign button = x & ~y;
```

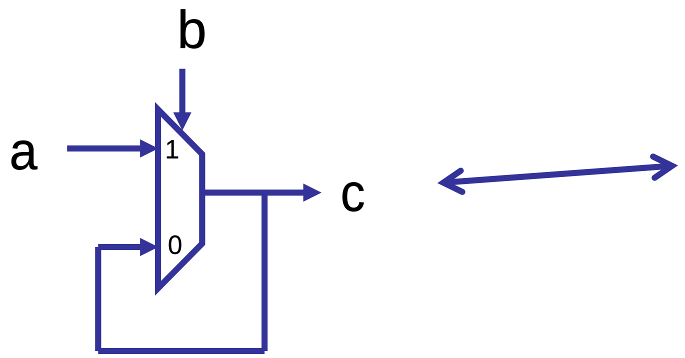


One more thing

```
 // This is OK
 always @(posedge clk)
   begin
       x <= stb;
       if (rst)
         x <= 0;
   end

// same as
always @(posedge clk)
   begin
       if (rst)
         x <= 0;
       else
         x <= stb;
   end
```

```
// This is not OK
// multiple assignment
 always @(posedge clk)
   begin
       x <= stb;
   end

 always @(posedge clk)
   begin
     if (rst)
       x <= 0;
   end
```

SV: always_{comb, ff, ~~latch~~}



```
// forgot else branch
// a synthesis warning
always @(a or b)
   if (b) c = a;
```

```
// compilation error
always_comb
   if (b)
      c = a;
```

```
// yes
always_comb
   if (b)
      c = a;
   else
      c = d;
```

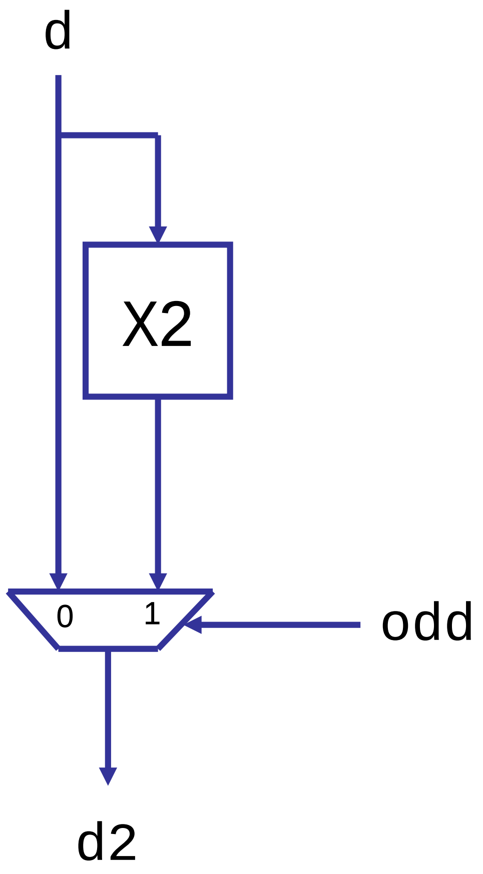
- always blocks do not guarantee capture of intent
- If not edge-sensitive then only a warning if latch inferred

- always_comb, always_latch and always_ff are explicit
- Compiler Now Knows User Intent and can flag errors accordingly

decade counter

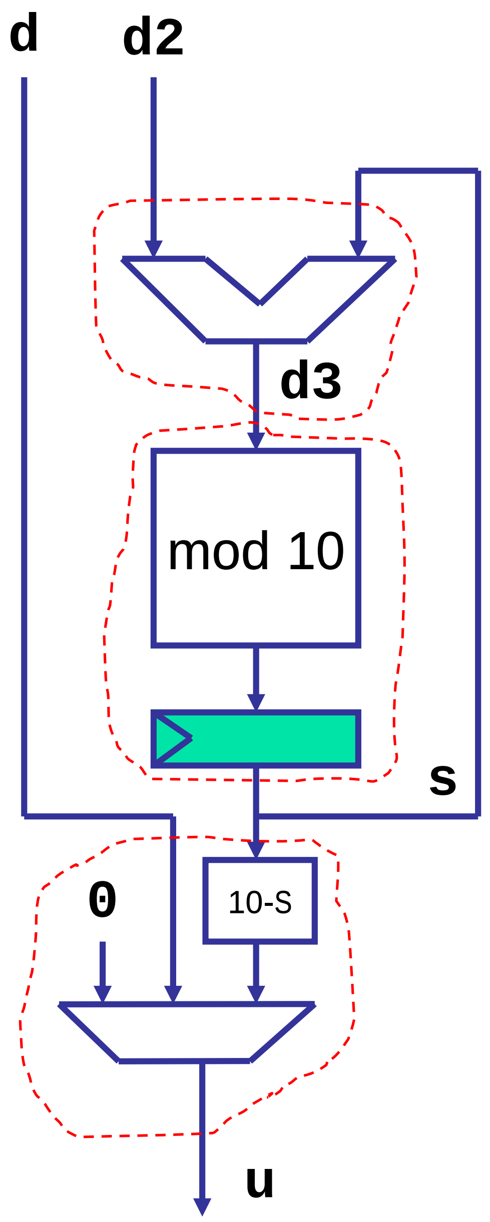
```
reg [3:0]      p;
wire           odd,last;

// 10 counter
  always_ff @(posedge clk) begin
    if (rst)
      p <= 4'd0;
    else begin
      if (button)
        if (p<9)
          p <= p+1;
        else
          p <= 4'd0;
    end
  end

  assign odd = ~p[0];
  assign last = (p==4'h9) ? 1'b1 : 1'b0;
```

X2



```
always_comb begin
   if (odd)
      case (d)
        4'h0:
          d2 = 4'h0;
        4'h1:
          d2 = 4'h2;
        4'h3:
          d2 = 4'h6;
        4'h4:
          d2 = 4'h8;
        4'h5:
          d2 = 4'h1;
        4'h6:
          d2 = 4'h3;
        4'h7:
          d2 = 4'h5;
        4'h8:
          d2 = 4'h7;
        4'h9:
          d2 = 4'h9;
        default:
          d2 = 4'h0;
      endcase
   else
     d2 = d;
end
```

ADD REG2,MOD10 K



```
// ADD
assign d3 = {1'b0,s} + {1'b0,d2};

// REG2 and MOD10
always_ff @(posedge clk) begin
  if (rst)
    s <= 4'h0;
  else if  (button)
    if (d3 < 10)
      s <= d3[3:0];
    else
      s <= d3[3:0] + 4'd6;
end

// K
assign u = (last == 1'b0) ? d :
           (s == 4'd0) ? 4'd0 :
           4'd10 - s;
```

reg or wire in Verilog

1)

```
always …
   a <= b & c;
```

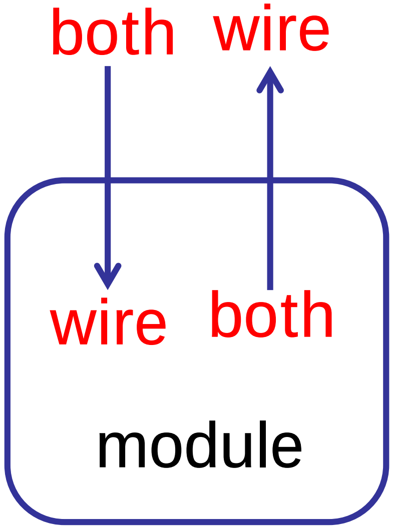
reg both

2)

wire both

```
assign a = b & c;
```

3)



SV relaxes variable use

A variable can receive a value from one of these :

- any number of `always`/`initial`-blocks
- one `always_ff`/`always_comb`-block
- one continuous assignment
- one module instance

We can skip `wire`/`reg`, use `logic` instead

Signed/unsigned

Numbers in verilog (95) are unsigned. If you write

```
  assign d3 = s + d2;
```

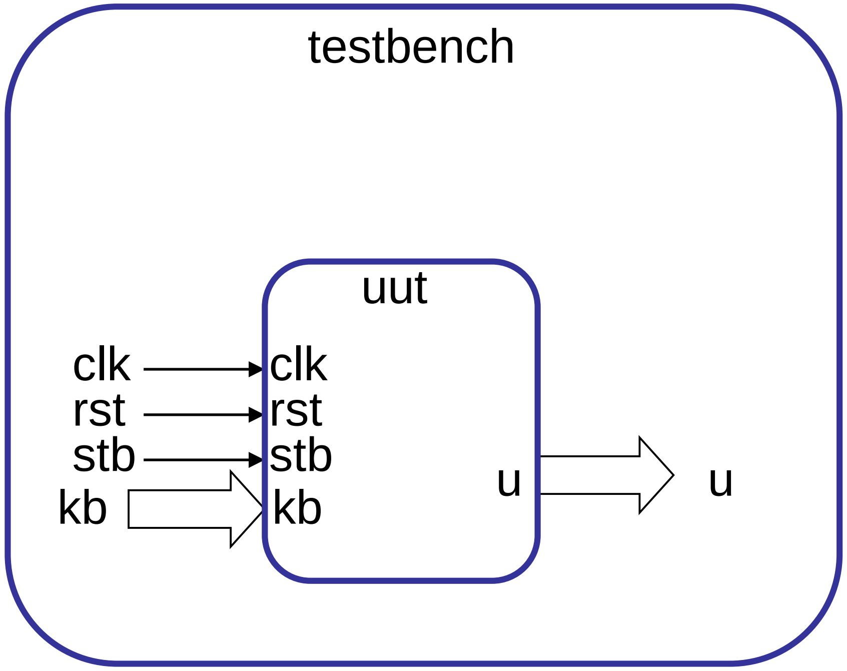
s and **d2** get zero-extended

```
wire signed [4:0] d3;
reg signed [3:0] s;
wire signed [3:0] d2;

assign d3 = s + d2;
```

s and **d2** get sign-extended

Test bench part 1

```
`include "timescale.v"

module testbench();
// Inputs
    reg clk;
    reg rst;
    reg [3:0] kb;
    reg stb;

// Outputs
    wire [3:0] u;

// Instantiate the UUT
    pnr uut (
        .clk(clk),
        .rst(rst),
        .kb(kb),
        .stb(stb),
        .u(u));
```



```
SV: pnr uut(.*);
```

Test bench part 2

```
// Initialize Inputs
     initial begin
        clk = 1'b0;
        rst = 1'b1;
        kb = 4'd0;
        stb = 1'b0;
        #70 rst = 1'b0;
        //
        #30 kb = 4'd8;
        #40 stb = 1'b1;
        #30 stb = 1'b0;
        //
        #30 kb = 4'd0;
        #40 stb = 1'b1;
        #30 stb = 1'b0;
     end

    always #12.5 clk = ~clk;  // 40 MHz
 endmodule
```

Blocking vs Non-Blocking

- Blocking assignment (**=**)
  – Assignments are blocked when executing
  – The statements will be executed in sequence, one after one

```
always_ff @(posedge clk) begin
   B = A;
   C = B;
end
```

- Non-blocking assignment (**<=**)
  – Assignments are not blocked
  – The statements will be executed concurrently

```
always_ff @(posedge clk) begin
   B <= A;
   C <= B;
end
```

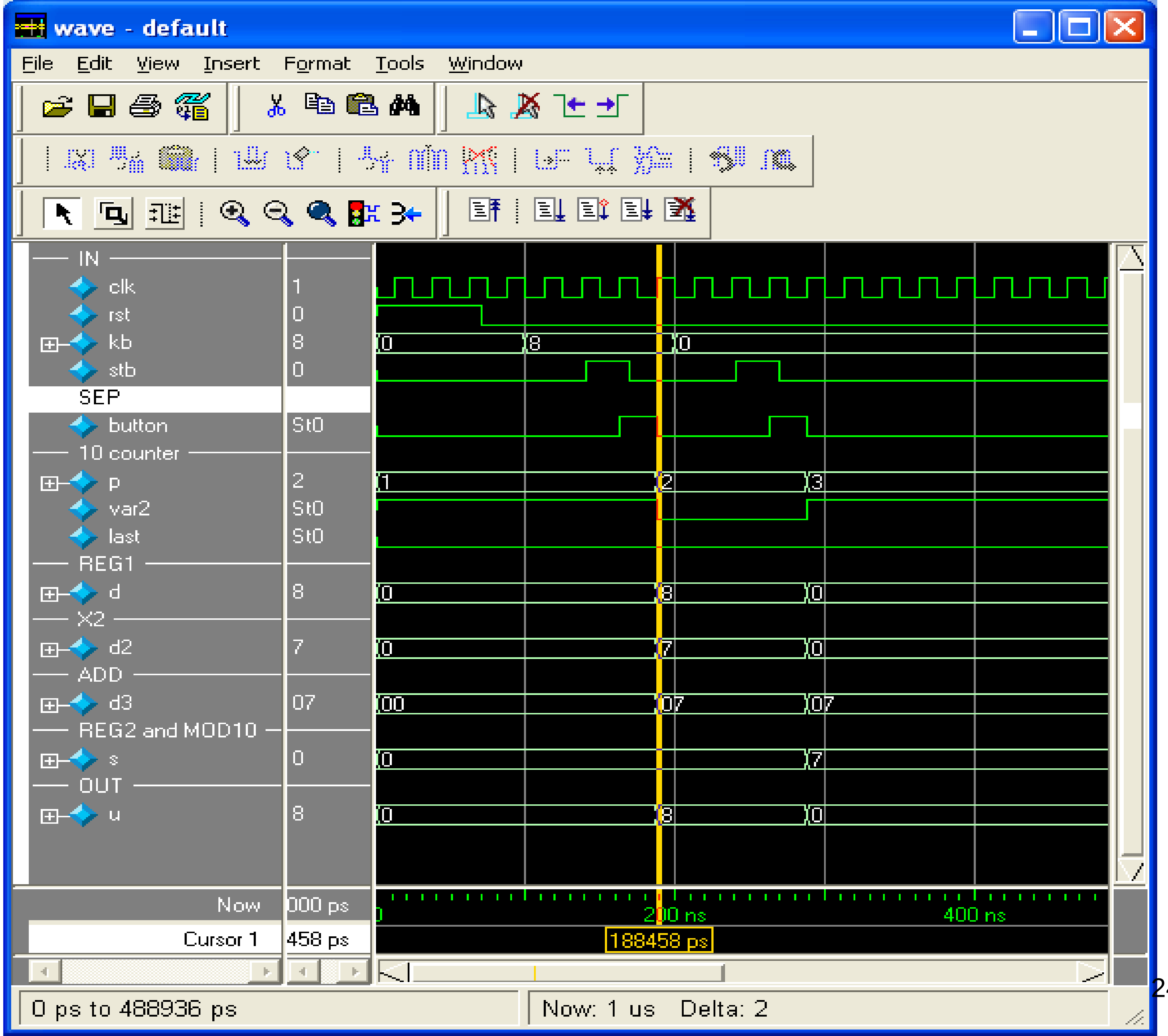
Use <= for sequential logic

Blocking vs Non-Blocking

```
always_comb begin
  C = A & B;
  E = C | D;
end

always_comb begin
  C <= A & B;
  E <= C | D;
end
```

Same result

Use = for combinatorial logic



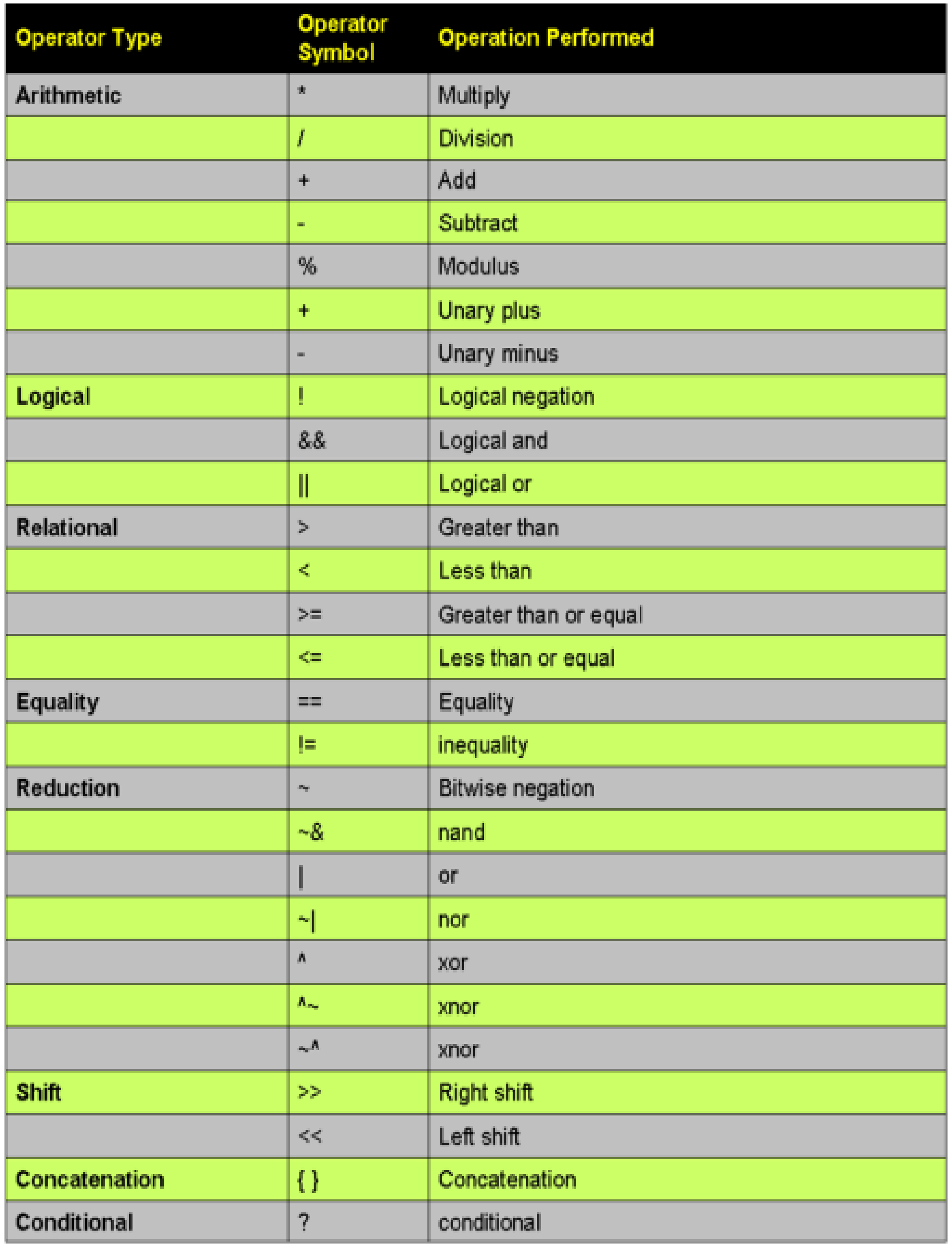
IN	
clk	1
rst	0
kb	8
stb	0
SEP	
button	St0
10 counter	
p	2
var2	St0
last	St0
REG1	
d	8
X2	
d2	7
ADD	
d3	07
REG2 and MOD10	
s	0
OUT	
u	8
Now	000 ps
Cursor 1	458 ps

0 ps to 488936 ps
Now: 1 us Delta: 2

Verilog constructs for synthesis

Construct type	Keyword	Notes
ports	input, inout, output	
parameters	parameter	
module definition	module	
signals and variables	wire, reg	Vectors are allowed
instantiation	module instances	E.g., mymux ml(out, iO, il, s);
Functions and tasks	function, task	Timing constructs ignored
procedural	always, if, then, else, case	initial is not supported
data flow	assign	Delay information is ignored
loops	for, while, forever	
procedural blocks	begin, end, named blocks, disable	Disabling of named blocks allowed

Operators



Operator Type	Operator Symbol	Operation Performed
Arithmetic	*	Multiply
	/	Division
	+	Add
	-	Subtract
	%	Modulus
	+	Unary plus
	-	Unary minus
Logical	!	Logical negation
	&&	Logical and
	||	Logical or
Relational	>	Greater than
	<	Less than
	>=	Greater than or equal
	<=	Less than or equal
Equality	==	Equality
	!=	inequality
Reduction	~	Bitwise negation
	~&	nand
	|	or
	~|	nor
	^	xor
	^~	xnor
	~^	xnor
Shift	>>	Right shift
	<<	Left shift
Concatenation	{ }	Concatenation
Conditional	?	conditional

Replication
{3{a}}
same as
{a,a,a}

Parameters

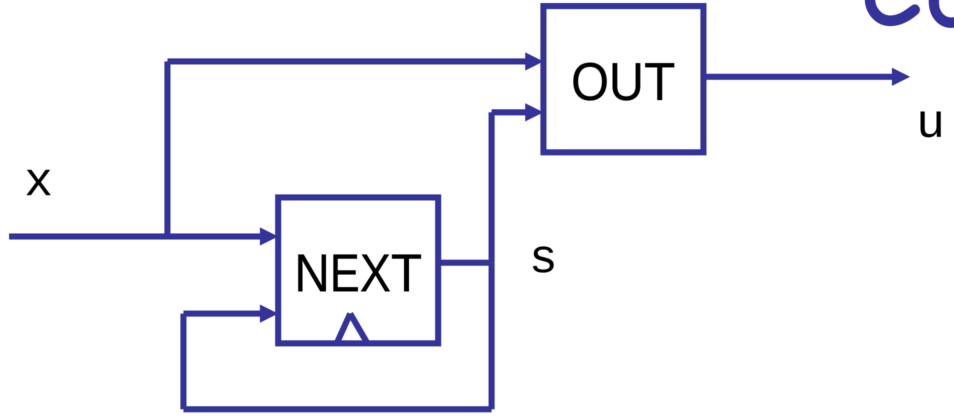
```
module w(x,y);

input x;
output y;

parameter z=16;
localparam s=3’h1;

...



endmodule
```

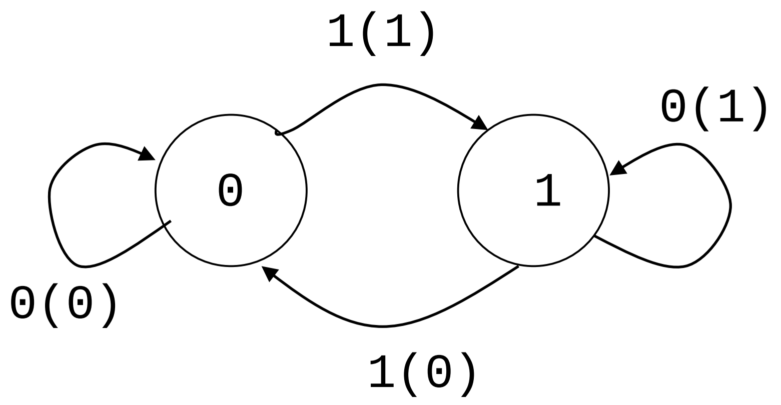
```
w w0(a,b);


w #(8) w1(a,b);


w #(.z(32)) w2(.x(a),.y(b));
```

Constants …

```
`include ”myconstants.v”
`define PKMC
`define S0 1’b0
`define S1 1’b1

`ifdef PKMC
...
`else
...
`endif
```

Comment: FSM1

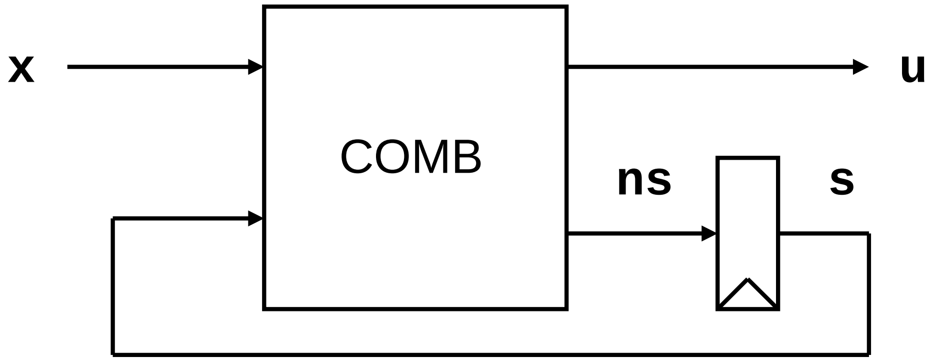


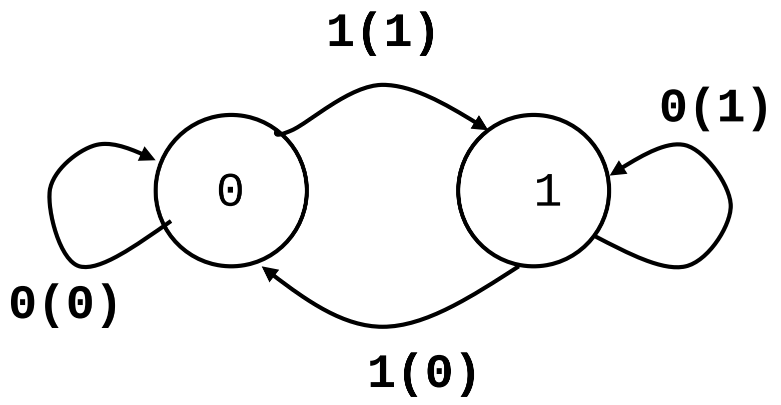


```
//NEXT
always_ff @(posedge clk) begin
 if (rst)
      s <= `S0;
 else
   case (s)
      `S0:
        if (x)
          s <= `S1;
      default:
        if (x)
          s <= `S0;
end
```

```
//OUT
always_comb begin
   case (s)
      `S0: if (x)
               u = 1’b1;
            else
               u = 1’b0;
      default: if (x)
                  u = 1’b0;
               else
                  u = 1’b1;
end
```

Comment: FSM2





```
// COMB ☺
always_comb begin
   ns = `S0;   // defaults
   u = 1'b0;
   case (s)
      `S0: if (x) begin
               ns = `S1;
               u = 1'b1;
           end
      default:
           if (~x) begin
               u = 1'b1;
               ns = `S1;
           end
end
```

```
// state register
always_ff @(posedge clk) begin
   if (rst)
      s <= `S0;
   else
      s <= ns;
end
```

Good to have

```
typedef logic [3:0] nibble;
nibble  nibbleA, nibbleB;

typedef enum {WAIT, LOAD, STORE} state_t;
state_t state, next_state;

typedef  struct {
   logic [4:0] alu_ctrl;
   logic stb,ack;
   state_t state } control_t;
control_t control;

assign control.ack = 1’b0;
```

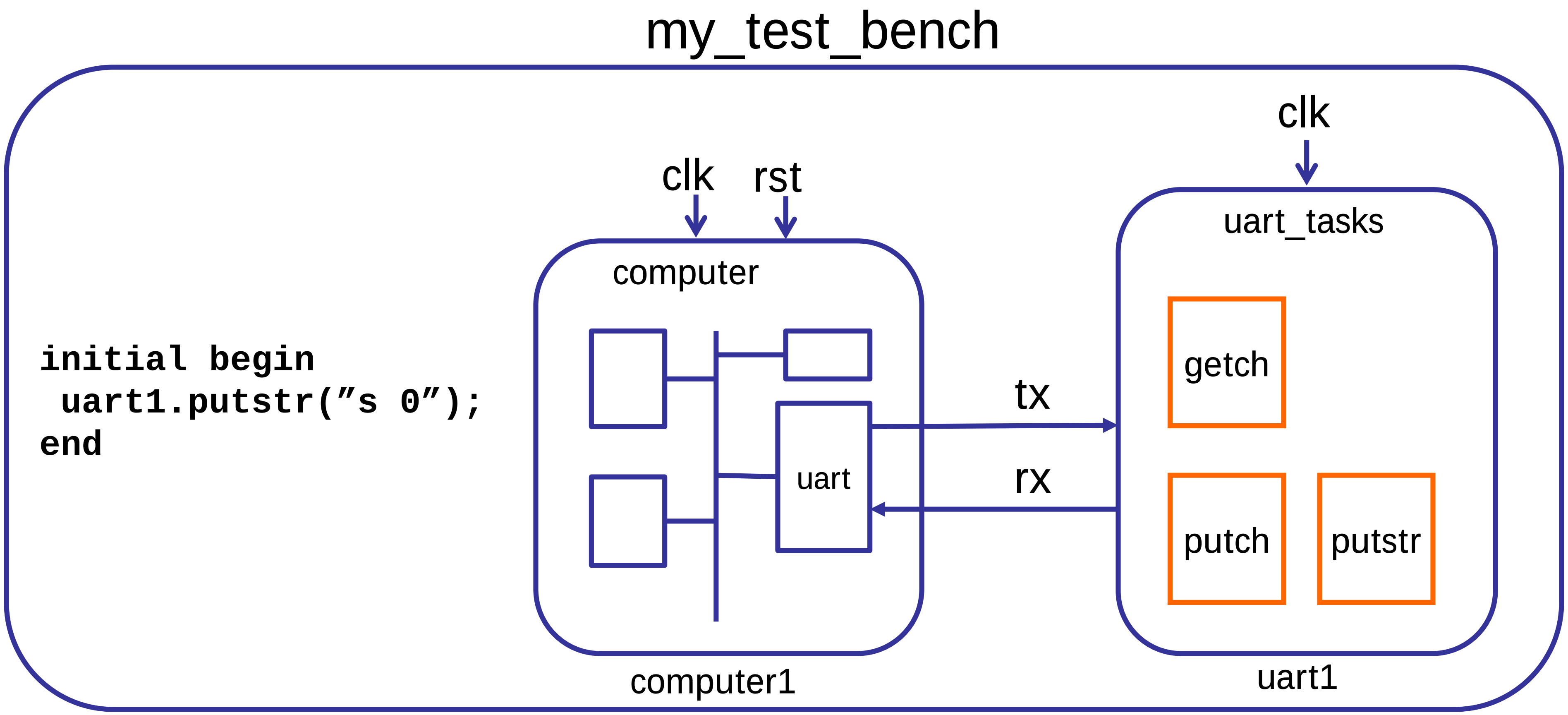
System tasks

Initialize memory from file

```
module test;
reg [31:0] mem[0:511]; // 512x32 memory
integer i;

initial begin
  $readmemh(”init.dat”, mem);
  for (i=0; i<512; i=i+1)
    $display(”mem %0d: %h”, i, mem[i]); // with CR
end
...
endmodule
```

Do you want a nicer test bench?

=> Try a task or two!



```
initial begin
 uart1.putstr(”s 0”);
end
```

Tasks

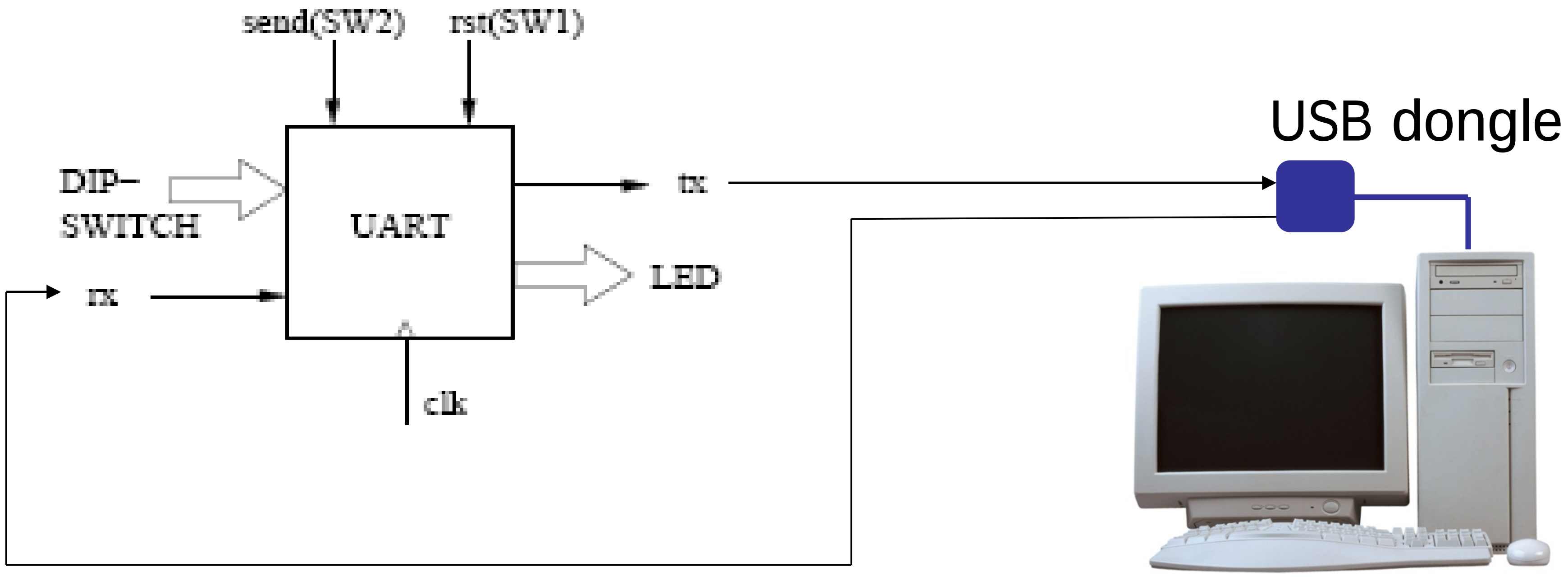
```
module uart_tasks(input clk, uart_tx,
                  output logic uart_rx);

   initial begin
      uart_rx = 1'b1;
   end

   task getch();
      reg [7:0] char;

      begin
         @(negedge uart_tx);
         #4340;
         #8680;
         for (int i=0; i<8; i++) begin
            char[i] = uart_tx;
            #8680;
         end
         $fwrite(32'h1,"%c", char);
      end
   endtask // getch
```

```
   task putch(input  byte char);
      begin
          uart_rx = 1'b0;
          for (int i=0; i<8; i++)
            #8680 uart_rx = char[i];
          #8680 uart_rx = 1'b1;
          #8680;
      end
   endtask // putch

   task putstr(input  string str);
      byte     ch;
      begin
         for (int i=0; i<str.len; i++)
            begin
               ch = str[i];
               if (ch)
                 putch(ch);
            end
         putch(8'h0d);
      end
   endtask // putstr

endmodule // uart_tb
```

In the testbench

```
wire tx,rx;
...
// send a command
initial begin
  #100000    // wait 100 us
  uart1.putstr(”s 0”);
end

// instantiate the test UART
uart_tasks uart1(.*);

// instantiate the computer
computer computer1(.*);
```

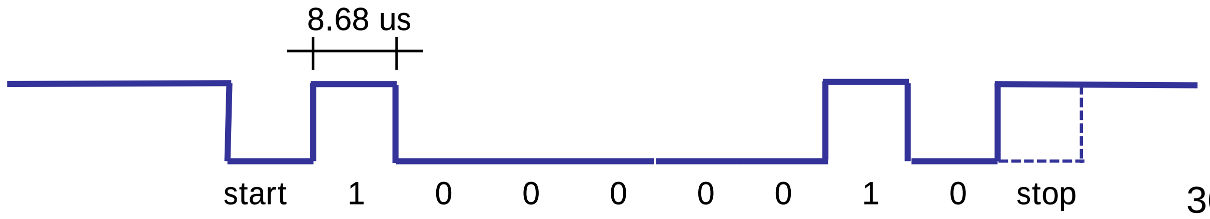
Lab 0 : Build a UART in Verilog



clk = 40 MHz

baud rate = 115200

full duplex



UCF = User constraint file

```
CONFIG PART = XC2V4000-FF1152-4 ;

NET "clk" LOC = "AK19";

# SWx buttons
NET   "stb"                LOC = "B3"   ;
NET   "rst"                 LOC = "C2"   ;

# LEDs
NET "u<0>" LOC = "N9"; //leftmost
NET "u<1>" LOC = "P8";
NET "u<2>" LOC = "N8";
NET "u<3>" LOC = "N7";
…
# DIP switches
NET "kb<0>" LOC = "AL3"; //leftmost
NET "kb<1>" LOC = "AK3";
NET "kb<2>" LOC = "AJ5";
NET "kb<3>" LOC = "AH6";
…
```

Lab0: Testbench

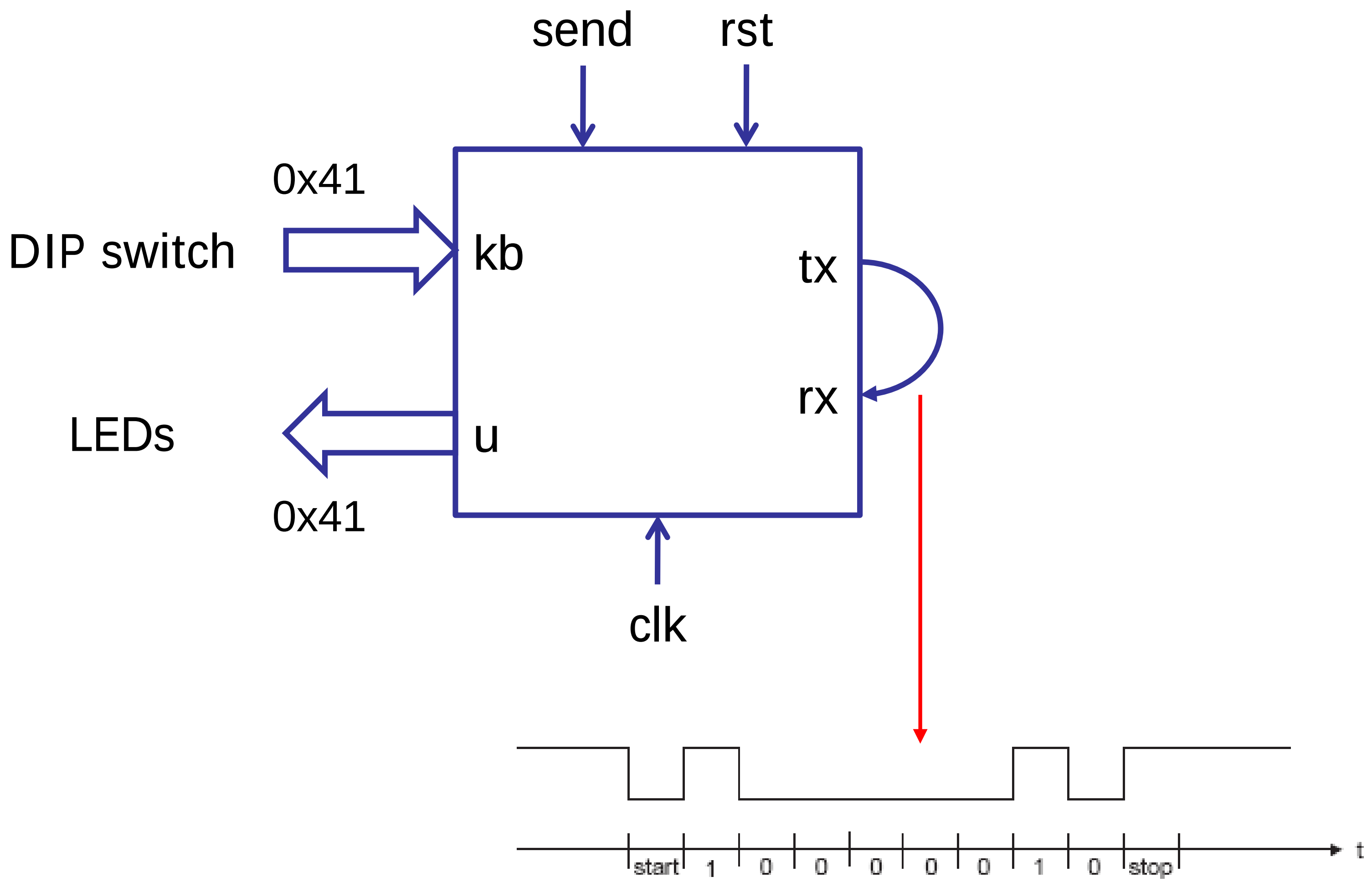
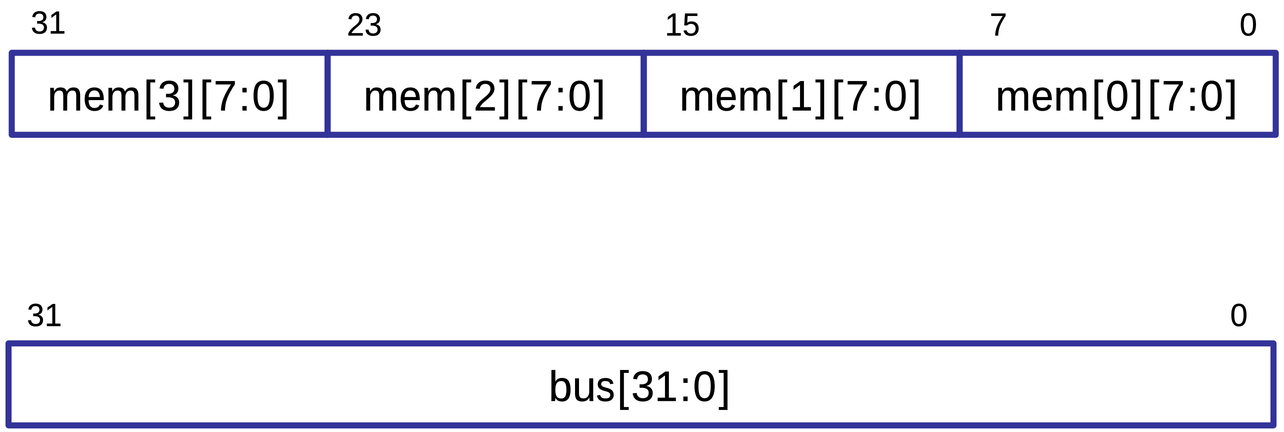


Figure 2.1: The letter A (0x41). Time per bit is 8.68 μs.

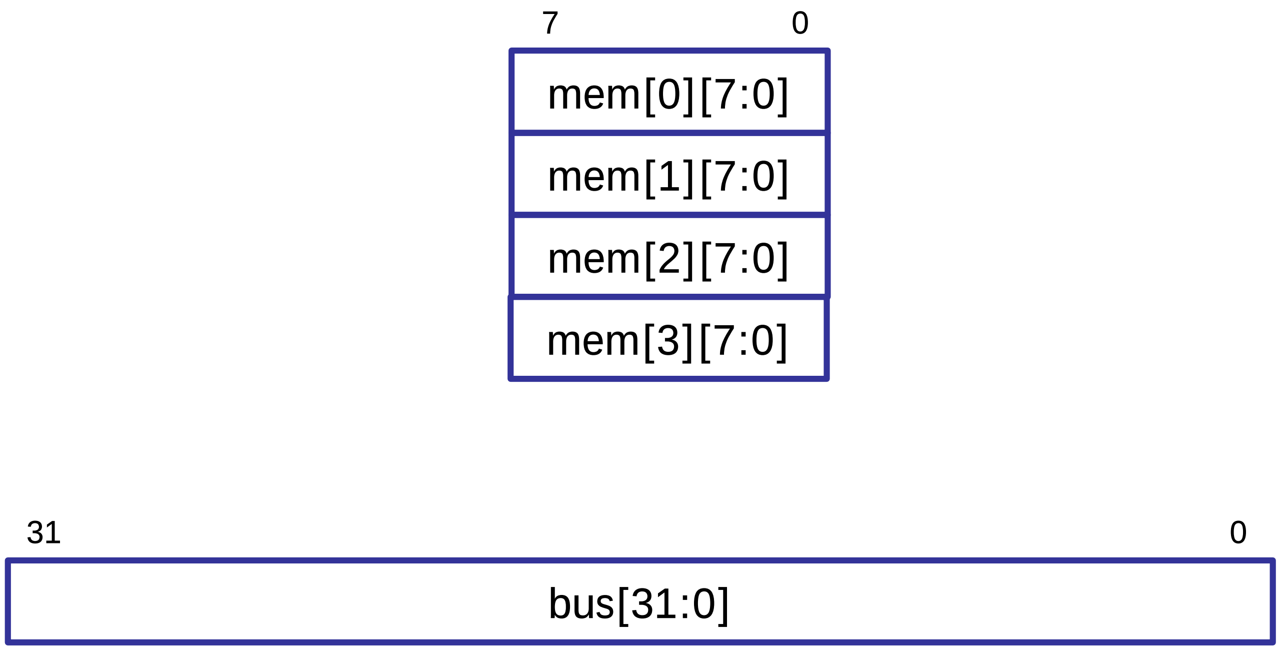
Arrays packed

```
logic [31:0] bus;               // a packed array
logic [3:0][7:0] mem;           // so is this
                                // both are contiguous

assign bus = mem;
assign bus[31:16] = mem[3:2];
```



Arrays unpacked

```
wire [31:0] bus;
reg [7:0] mem [0:3];  // 4 bytes
…
assign bus[31:24] = mem[3];
```


7
0
mem[0][7:0]
mem[1][7:0]
mem[2][7:0]
mem[3][7:0]
31
0
bus[31:0]