TSEA44: Computer hardware – a system on a chip

Lecture 2: A short introduction to SystemVerilog

LINKÖPING UNIVERSITY

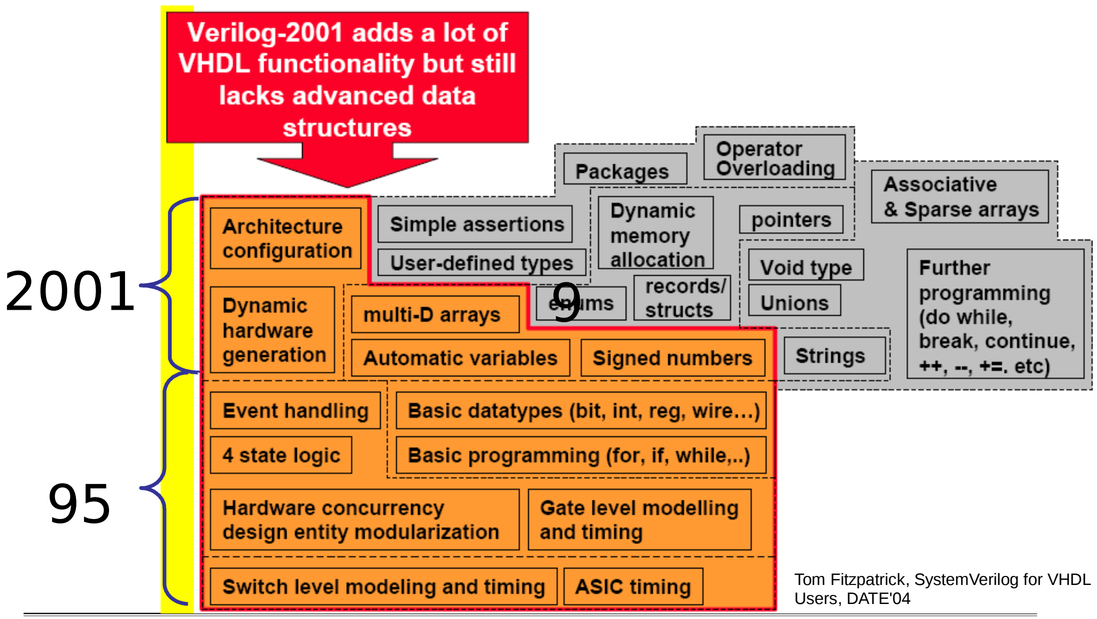
(System)Verilog

- Assume background knowledge of VHDL and logic design
- Focus on coding for synthesis
  - Testbenches will be mentioned
- Verilog was initially used for **modelling** of hardware, but later where software created that allowed the model to be synthesized into hardware
  - Adds restrictions on how the code should be written

History of Verilog and SystemVerilog

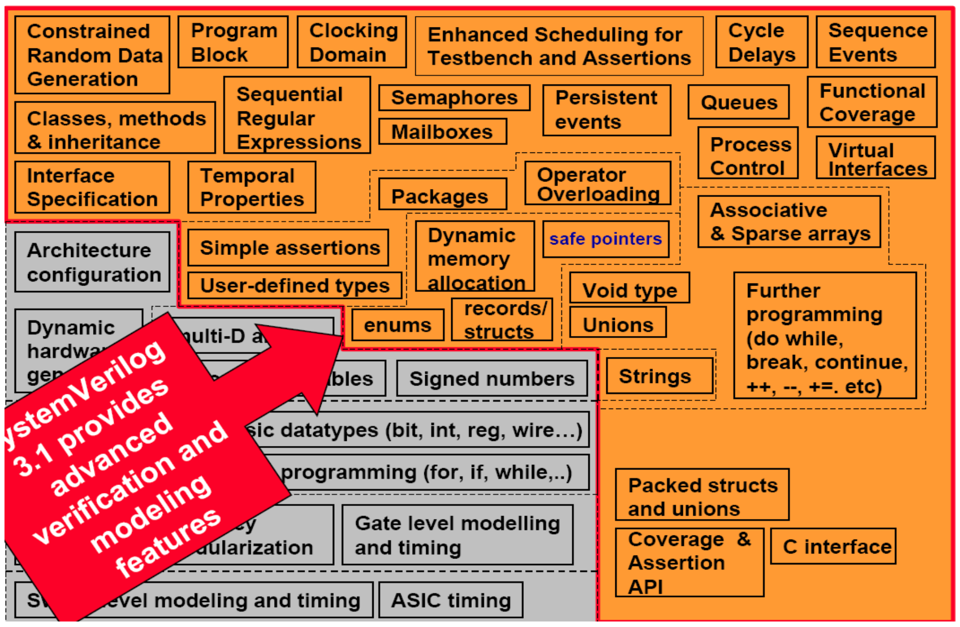
1985	Verilog invented, C-like syntax, initial use: modelling of hardware
	<- VHDL defined in 1987
1995	First standard of Verilog (Verilog-95, IEEE 1364)
2001	Extra features added (Verilog-2001, IEEE 1364-2001)
2005	Minor extension (Verilog-2005)
2005	SystemVerilog standardized (SystemVerilog-2005, IEEE 1800-2005) as an extension to Verilog-2005
2009	Merge of SystemVerilog and Verilog (IEEE 1800-2009)

Verilog vs VHDL


Verilog-2001 adds a lot of VHDL functionality but still lacks advanced data structures
2001
95
Architecture configuration
Simple assertions
User-defined types
Dynamic hardware generation
multi-D arrays
enums
records/ structs
Automatic variables
Signed numbers
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
Packages
Operator Overloading
Dynamic memory allocation
pointers
Associative & Sparse arrays
Void type
Unions
Strings
Further programming (do while, break, continue, ++, --, +=. etc)
9

Tom Fitzpatrick, SystemVerilog for VHDL Users, DATE'04

SystemVerilog

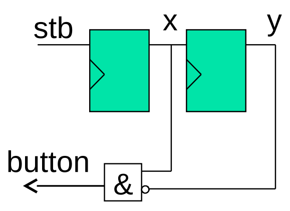

Constrained Random Data Generation
Program Block
Clocking Domain
Enhanced Scheduling for Testbench and Assertions
Cycle Delays
Sequence Events
Classes, methods & inheritance
Sequential Regular Expressions
Semaphores
Mailboxes
Persistent events
Queues
Functional Coverage
Process Control
Virtual Interfaces
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 ge
ulti-D a
enums
records/ structs
bles
Signed numbers
Strings
sic datatypes (bit, int, reg, wire…)
programming (for, if, while,..)
Packed structs and unions
ularization
Gate level modelling and timing
Coverage & Assertion API
C interface
S
evel modeling and timing
ASIC timing
ystemVerilog 3.1 provides advanced verification and modeling features

LINKÖPING UNIVERSITY

Synchronization and single pulse

- Asynchronous inputs must be synchronized to the input clock
- Useful: indicate input signal edge detection

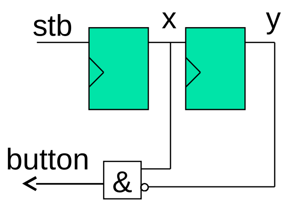
```
reg x,y;     // variable type  (0,1,Z,X)
wire button; // net type (0,1,Z,X)

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

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



LINKÖPING UNIVERSITY

A variation of the same thing

- Multiple always block



```
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;
```

Resetable D-flipflop

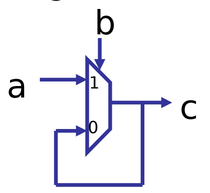
```
  // This is OK
 always @(posedge clk)
   begin
     x <= stb;
     if (rst)
          x <= 0;
   end

// 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
```

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

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

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



- always_ff, always_comb, always_lath are explicit
- Compiler now knows user intent and can flag errors accordingly

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

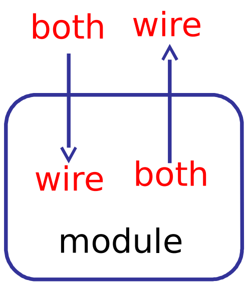
```
// yes
always_comb
   if (b)
      c = a;
   else
      c = d;
```

Reg or wire in Verilog

1)
```
always …
   a <= b & c;
```
reg both

2)
wire both
```
assign a = b & c;
```

3)



SystemVerilog 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

```
// s and d2 4 bits long, d3 5 bit long
assign d3 = s + d2;
```

s and d3 gets zero-extended

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

assign d3 = s + d2;
```

s and d3 get sign-extended

Blocking vs non-blocking, sequential

- Blocking assignment (=)
  - Assignments are blocked when executing
  - The statements will be executed in sequence, one after the other
- Non-blocking assignment (<=)
  - Assignments are not blocked
  - The statements will be executed concurrently

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

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

Use <= for sequential logic

Blocking vs non-block, combinatorial

```
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

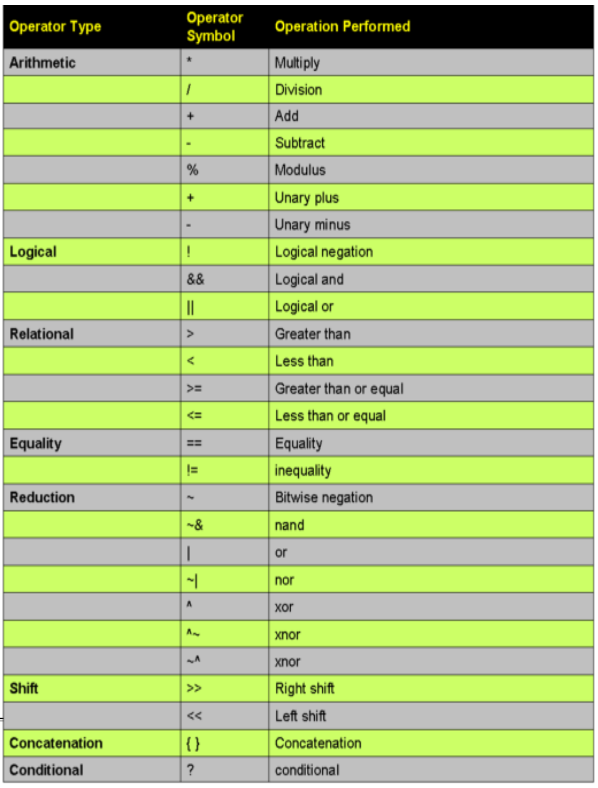
Verilog constructs for synthesis

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

- Reduction:
  return scalar value from vector by pairwise calculation

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



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

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));
```

LINKÖPING UNIVERSITY

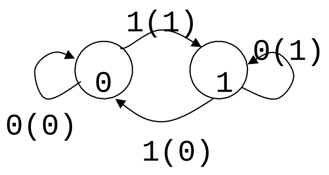
Constants (really substitution macros)

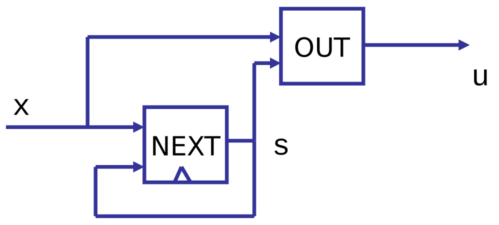
```
`include ”myconstants.v”

`define PKMC

`define S0 1’b0

`define S1 1’b1


`ifdef PKMC

...

`else

...

`endif
```

Important: It is `, not '
(back tick instead of
apostrophe)

LINKÖPING UNIVERSITY

Example of an FSM (nextstate + out)

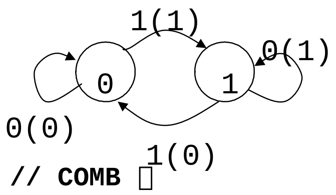


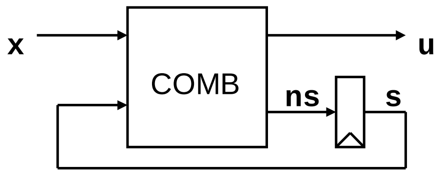


```
//NEXT
always_ff @(posedge clk) begin
 if (rst)
      s <= `S0;
 else
   case (s)
      `S0:
        if (x)
          s <= `S1;
      default:
        if (x)
          s <= `S0;
end
```

```
//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
```

Alternative FSM (separate register)





```
// 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
```

This description stops us from adding unintentional extra states and flipflops

Adding more datatypes, including struct

```
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;
```

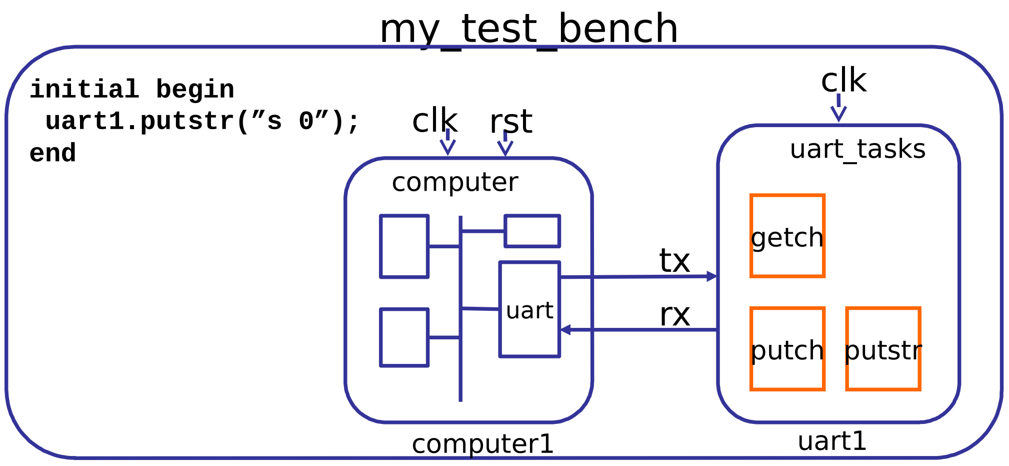
System tasks

- 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
```

Expand testbench functions using tasks

- Tasks are subroutines (like procedures in VHDL)
- Initial statement only in testbenches



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

Tasks example

```
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
```

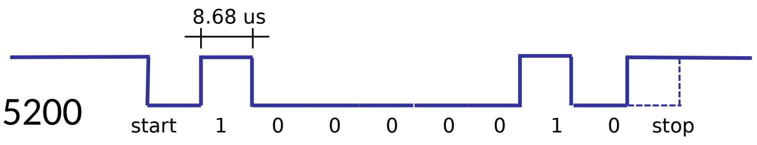
```
task putch(input  byte char);
      begin
     uart_rx = 1'b0;
     for (int i=0; i<8; i++)
       #8680 uart_rx = char[i];
     #8680 uart_rx = 1'b1;
     #8680;
      end
   endtask // putch

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
```

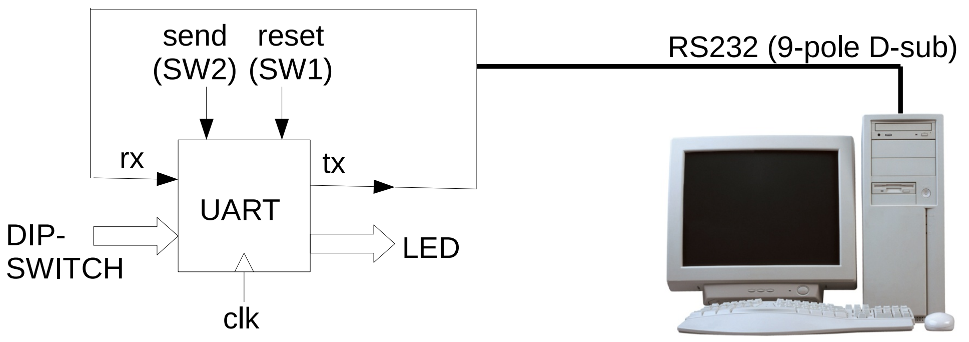
In the testbench

```
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 an UART in Verilog

- Clk = 40 Mhz
- Baud rate = 115200
- Full duplex (support concurrent send and receive)





(Voltage level shifter etc not shown)

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";
…
```

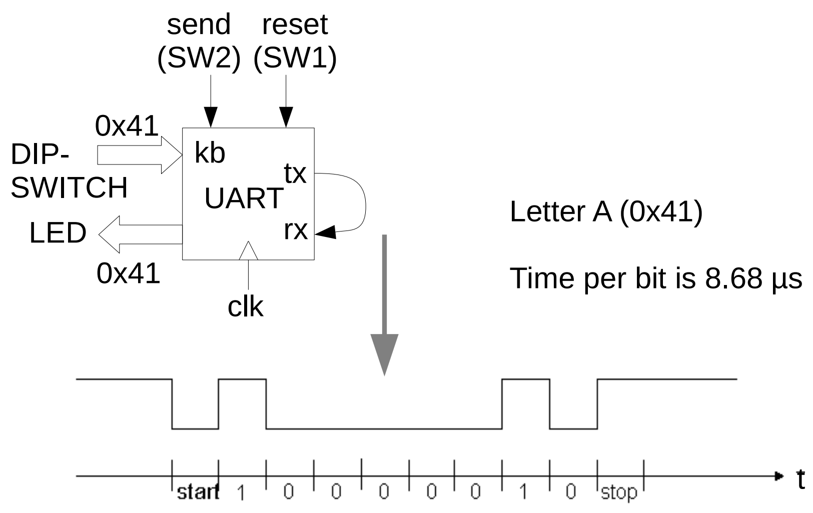
Net names must match names used in the module description

Additional requirement 2016!

- New requirement this year on lab 0 result
  - The reset state on the LED should indicate the value of your student id:s last two digits.
  - Example: Linus123 should have the value 23 on the LED when reset is applied (and kept there until a value is received on rx).
    - 23 = 16+4+2+1 = 00010111
- Remember lab 0 is individual

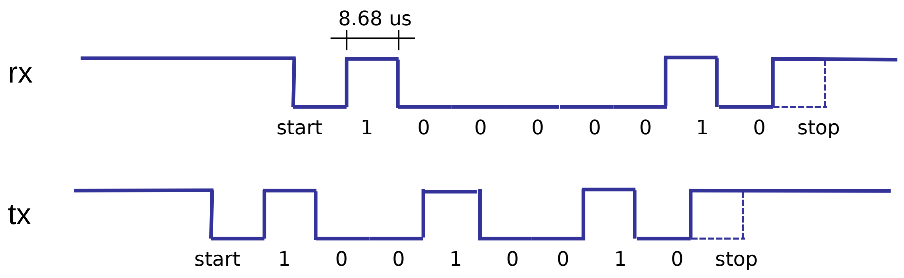
2016-11-02 23:44

Lab 0: Testbench



Things to look out for

- Testbench have automatically synchronization of rx and tx
  - In real life, a rx bit may start when half o a tx bit have been sent



- Do not forget to wait for the stop bit in rx!
  - May otherwise detect last data bit as new start bit!

Bigger example: Personal number check

- Swedish social security number (personnummer) consists of 10 digits, where the last is a checksum digit.

  $d_1\ d_2\ d_3\ d_4\ d_5\ d_6\ d_7\ d_8\ d_9\ d_{10}$

  d_{10} = (10 – ([2d_1]+d_2+[2d_3]+d_4+…+[2d_9])) mod 10

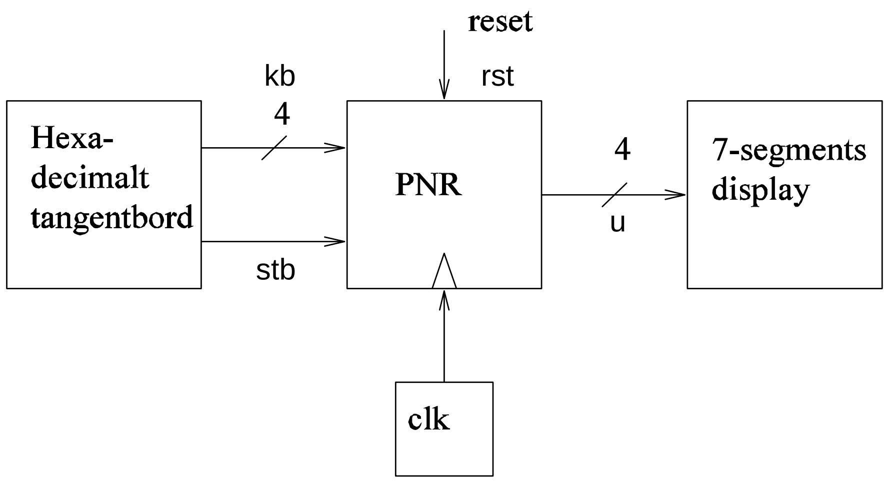
  where [2d_x] is digit sum of 2*d_x (example d_x=6 => 2*6=12 => [2d_x]=3)

- Iterative solution

$$S_0 = 0$$
$$S_k = S_{k-1}\, mod_{10}\, X\, 2(d_k) \quad k = 1..9$$
$$d_{10} = 10 - S_9$$

Overall system

- PNR is calculating the checksum, presenting it on the display



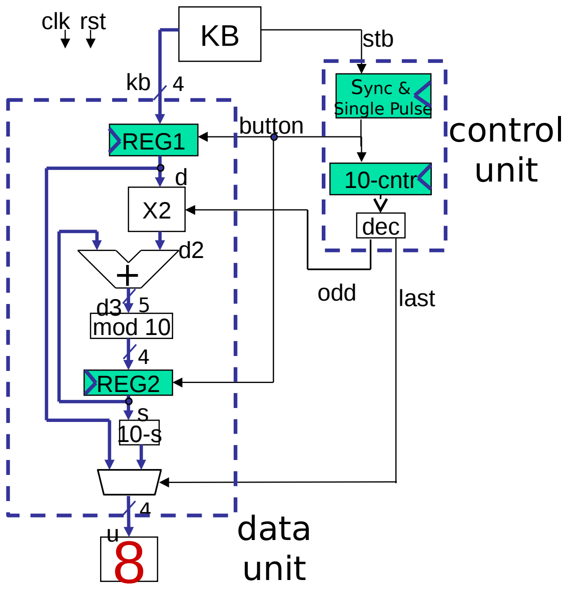
Top module (pnr module)

```
                          `timescale 1ns / 1ps
`include "timescale.v“

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


// our design



endmodule
```

* No entity/architecture distinction => just module

Block schematic (thinking hardware!)

- Split into datapath and control
- Green boxes synch FSM
- White boxes comb
- Button is singlepulsed

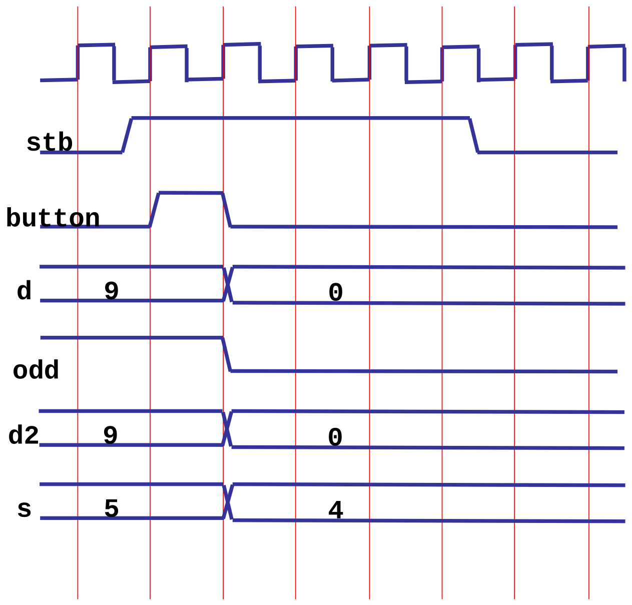


Timing diagram

- When button is pressed:

  d2 = digitsum(2*9) = digitsum(18) = 9

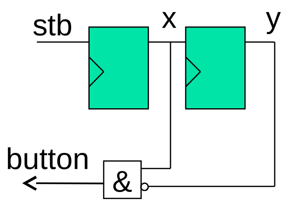
  New s = (d2 + s) mod 10 = (9 + 5) mod 10 = 14 mod 10 = 14



Synch and single pulse shown before

- Always synchronize external inputs!!

```
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;
```

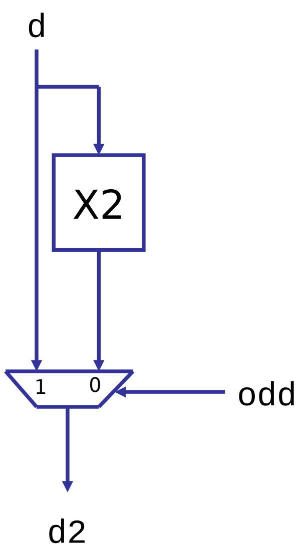


Decade counter

```
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;
```

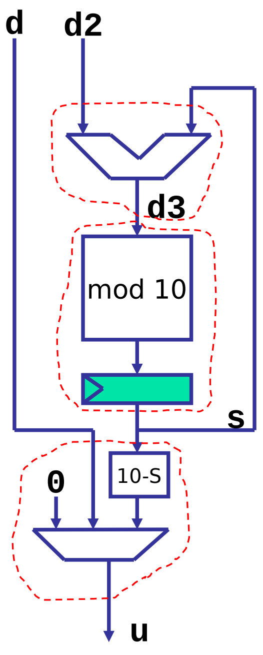
X2 (multiply and add digits when odd=0)

```
always_comb begin
  if (~odd)
    case (d)
     4'h1:
       d2 = 4'h2;
     4'h2:
       d2 = 4'h4;
     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
```



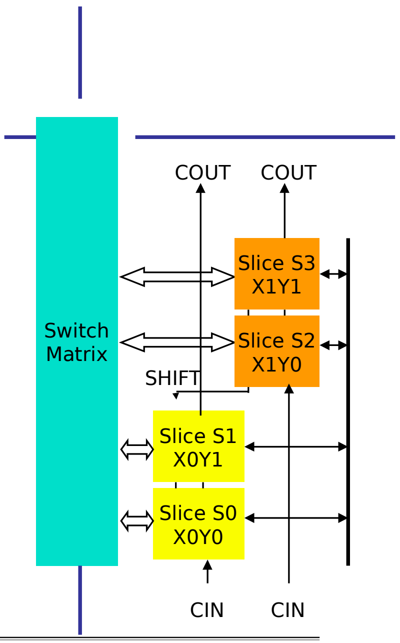
ADD Reg2, Mod10 K

```
// 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;
```



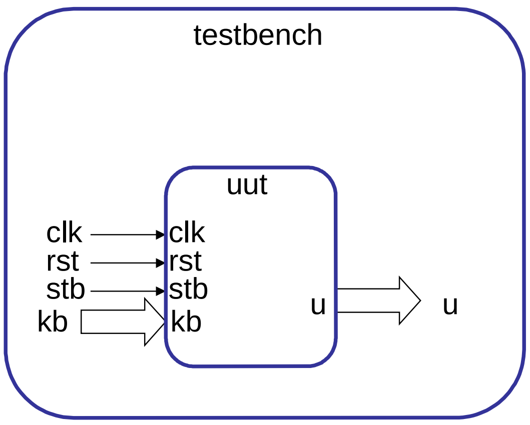
CLB contains four slices

- Each CLB is connected to one switch matrix
  - 1 slice = 2 LUT/FF + …
- High level of logic integration
  - Wide-input functions
    - 16:1 multiplexer in 1 CLB
    - 32:1 multiplexer in 2 CLBs
  - Fast arithmetic functions
    - 2 look-ahead carry chains per CLB column
  - Adressable shift register in LUT
    - 16-b shift register in 1 LUT
    - 128-b shift register in 1 CLB



Testbench for PNR (1 of 2)

```
`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));
```



```
// SystemVerilog instantiation
// automatic mapping port-wire
pnr uut(.*);
```

Testbench for PNR (2 of 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
```

Add more digits and strobe pulses to test a complete number www.liu.se

LINKÖPING
UNIVERSITY