CURRICULUM

INF3430 - fall 2013

Author:
JOAKIM MYRVOLL
ALEKSANDER POLLEN

User:
joakimmj
alekspo

Due: 09.12.13

Contents

Part I
VHDL

1 Logical Syntax

1.1 Logical Expressions

```
[not] <identifier> [
  [and | or | nor | nand | xor | xnor | ... ]
  [<identifier>]
];
-- this type of expression can, naturally, be combined using logical functions
```

Example:

```
not signal_1;
signal_1 and signal_2;
(not signal_1) and (signal_2 xor (signal_3 or (not signal_4)) xor signal_5);
```

1.2 If-Then-Else Statements

```
if <condition> then
  statements
  ...
[
elsif <condition> then
  statements
  ...
else
  statements
  ...
]
endif;
```

Example:

```
if boolean_v then
  output_1 <= '1';
end if;
-------------------------
if condition_v_1 = '1' then
  out_vector <= "001"
elsif condition_v_2 = '1' then
  out_vector <= "110"
  ...
else
  out_vector <= "000"
end if;
```

1.3 Case Statements

```
case <expression> is
  when <choice(s)> =>
    <expression>;
    ...
  when ...
  [when others => ... ]
end case;
```

Example:

```
case scancode is
  when x"14" =>
    integer_signal <= 1;
  when x"18" =>
    integer_signal <= 2;
  when x"19" | x"20" | x"21" =>
    integer_signal <= 3;
  when others =>
    integer_signal <= 0;
end case;
```

1.4 When Statements

```
<signal> <= <value/signal/etc> when <expression> else
            <value/signal/etc> when <expression> else
            <value/signal/etc> when <expression> else
            <value/signal/etc>;
```

Example:

```
output <= in0 when (s1 & s0)="00" else
          in1 when (s1 & s0)="01" else
          in2 when (s1 & s0)="10" else
          in3 when (s1 & s0)="11" else
          'X';
```

1.5 With Statements

```
with <expression> select
  output <= <value/signal/etc> when <choice(s)>,
            <value/signal/etc> when <choice(s)>,
            <value/signal/etc> when <choice(s)>,
            <value/signal/etc> when others;
```

Example:

```
with sel select
  output <= in0 when "00",
            in1 when "01",
            in2 when "10",
            in3 when "11",
            'X' when others;
```

2 Structural Syntax

2.1 Signal Assignments

```
<signal_name> <= <expression>;
-- the expression must be of a form whose result matches the type of the assigned signal
```

Example:

```
std_logic_signal_1 <= not std_logic_signal_2;
std_logic_signal <= signal_a and signal_b;
std_logic_signal <= std_logic(not_std_logic_signal);
large_vector(15 downto 6) <= small_vector(10 downto 0);
std_logic_vector_signal <= std_logic_vector(not_std_logic_vector_signal);
```

2.2 Variable Assignments

```
<variable_name> := <expression>
-- the expression must be of a form whose result matches the type of the assigned variable
```

Example:

```
boolean_v := true;
temp_v(3 downto 0) := sl_vector_signal(7 downto 4);
```

2.3 Processes

```
[<process_name>:]
process (<sensitive signals>)
  variable declarations
  constant declarations
  ...
begin

  statements
  ...
end process;
```

Example:

```
output_process:
process(flag_signal)
begin

if flag_signal = '1' then
  output_vector <= "010";
else
  output_vector <= "101";
end if; end process;
```

2.4 Component Instantiations

2.4.1 Port-mapping

```
<component_identifier> : <component_name> generic map(<assigned_signals>) port map(<assigned_signals>);
```

Example:

```
EC : example_component port map(mclk_i, rst_i);
```

2.4.2 Name-associated Port-mapping

```
<component_identifier>: <component_name>
port map(
  <port_name> => <assigned_signal>,
  ...
)
-- you can also just list assigned signal names in the order you declared them
-- in the component declaration, but this is not generally considered good
-- coding style
generic map(
  <generic_name> => <assigned_generic>,
  ...
);
```

Example:

```
EC : example_component
port map
(
  rst             => rst_i,
  mclk            => mclk_i
);
```

3 Data Types

3.1 Logical Types

3.1.1 std_logic

```
std_logic         -- one bit
std_logic_vector  -- vector of bits
```

Example:

```
std_logic_signal <= '1';
std_logic_signal <= '0';

sl_vector_signal_8 <= "11110000"; -- an 8 bit vector
sl_vector_signal_8 <= x"F0";      -- equivalent to above

-- you can assign from part of a larger vector
sl_vector_signal_8 <= sl_vector_signal_16(15 downto 8);
-- access a single bit
std_logic_signal <= sl_vector_signal_8(5)

-- set all bits to '0'
sl_vector_signal_8 <= (others => '0')
```

3.1.2 boolean

```
constant CONDITION_C: boolean := false;
variable bool_v: boolean := true;
```

3.2 Ranged Types

Type:

```
integer: -(231) to 231 - 1
-- note: this means it uses a 32 bit signed signal
-- these are the subtypes of integer:
positive: 1 to integer'high
natural:  0 to integer'high
```

Syntax:

```
signal <integer_name> : integer range <low> to <high>;
--(or variable or constant)
```

Example:

```
signal cycle_length: integer 0 to 15;
-- this will be a 4 bit signal
signal other_int: integer -1 to 15;
-- this is 5 bits because it needs a sign
variable ram_addr_v: natural 0 to ADDRMAX;
-- you can also base your range on defined constants
```

4 Module Structure

4.1 Library

```
library <library_name>;
use <library_name>.<package_name>.[all|<part>];
```

Example (standard):

```
library IEEE;
use IEEE.std_logic_1164.all;
-- numeric
use IEEE.numeric_std.all;
```

4.2 Entity

```
entity <entity_name> is
port(
  port assignments
  ...
  -- all port assignments are followed by semi-colons
  -- except the last one
);
generic(
  generic assignments
  ...
  -- all generic assignments are followed by semi-colons
  -- except the last one
);
end [entity / <entity_name>];
```

Example:

```
entity or_entity is
port(
  input_1: in std_logic;
  input_2: in std_logic;
  output: out std_logic
);
end or_entity;
```

4.2.1 Port

```
<signal_name>: in/out/inout type;
```

Example:

```
enable: in std_logic;
data_bus: out std_logic_vector(7 downto 0);
```

4.2.2 Generic

```
<generic_name>: type [:= <initial_value>];
```

Example:

```
bus_width: integer := 8;
my_boolean: boolean := false;
```

4.3 Architecture

```
architecure <architecture_name> of <entity_name> is
[
  component declarations (detail)
  function declarations (detail)
  signal declarations (detail)
  constant declarations (detail)
  variable declarations (detail)
  type declarations (detail)
  ...
  -- there are others, but these are what you'll need for this course
]
begin
[
  combinatorial statements
  sequential statements
  ...
]
end architecture;
```

Example:

```
architecture or_entity_arch of or_entity is
begin
  output <= input_1 or input_2;
end architecture;
```

5 Declarations

5.1 Signal Declarations

```
signal <signal_name> : type;
-- you can add an initial value, but these are only supported
-- in simulation and not for synthesis
```

Example:

```
signal port_i : std_logic;
signal bus_signal: std_logic_vector(15 downto 0);
signal count: integer range 0 to 31;
```

5.2 Constant Declarations

```
constant <constant-name> : type := <initial value>;
-- naturally, you should always have an initial value
```

Example:

```
-- state values
constant state_1 : std_logic_vector := "01";
constant state_2 : std_logic_vector := "10";

constant addr_max: integer := 1024; -- maximum address value
```

5.3 Function Declarations

```
function <function_name> (
  -- function arguments
  <signal_1> : type; . . . < signal_n> : type )
returns return type is
  constant declarations
  variable declarations
  ...
begin
  statements
  ...
end <function_name>;
```

Example:

```
function sign_extend (
  narrow_bus: std_logic_vector(15 downto 0)
)
return std_logic_vector(31 downto 0) is
  variable output: std_logic_vector(31 downto 0);
begin
  output(15 downto 0) <= narrow_bus(15 downto 0);
  -- lower 16 are the same
  output(31 downto 16) <= (others => narrow_bus(15));
  -- upper 16 are sign-extension of MSB
  return output;
end sign_extend;
```

5.4 Procedure Declarations

Procedures may have in, out or inout parameters. These may be signal, variable or constant. the default for in parameters is constant. For out and inout it is variable. In fact, a constant in parameter can be associated with a signal, variable constant or expression when the procedure is called:

```
procedure procedure_name (parameter_list) is
  declarations
begin
  sequential statements
end procedure_name;
```

Example:

```
procedure DISPLAY_MUX
 (ALARM_TIME, CURRENT_TIME : in digit;
  SHOW_A                   : in std_ulogic;
  signal DISPLAY_TIME : out digit) is
begin
  if (SHOW_A = '1') then
    DISPLAY_TIME <= ALARM_TIME;
  else
    DISPLAY_TIME <= CURRENT_TIME;
  end if;
end DISPLAY_MUX;
```

Usage:
Procedures may be called concurrently or sequentially. A concurrent procedure call executes whenever any of its in or inout parameters change:

```
architecture SUBPROG of DISP_MUX is
  ...
begin
  DISPLAY_MUX (ALARM_TIME, CURRENT_TIME,
                SHOW_A, DISPLAY_TIME);
end SUBPROG;
```

5.5 Component Declarations

```
component <component_name> is
port (
  port declarations as done
  in entity declarations
  ...
);
generic(
  generic declarations as done
  in entity declarations
  ...
);
end component;
```

Example:

```
component or_entity is
port(
  input_1: in std_logic;
  input_2: in std_logic;
  output: out std_logic
);
end component;
```

5.6 Variable Declarations

```
variable <variable_name> is : type;
```

Example:

```
variable count_v: integer range 0 to 15;
variable data_v: std_logic_vector(7 downto 0);
variable condition_v: boolean;
variable cnt : unsigned;
variable err : signed(7 downto 0);
```

5.7 Type Declarations

The type keyword allows you to define your own data type in VHDL. These are interpreted and subsequently synthesized by synthesis tools. You can use types to create your own data types or arrays of existing data types.

5.7.1 Type

```
type <type_name> is (<values>);

-- where values is a list of acceptable values
-- array types can be defined as follows:

type <type_name> is array (<low> to <high>) of <data_type>;
type <type_name> is array (<high> downto <low>) of <data_type>;
```

Example:

```
type enum_type is (a, b, c, ..., z);
type int_array is array(3 downto 0) of integer;
type sl_vector_array is array(0 to 15) of std_logic_vector;
```

5.7.2 Subtype

The subtype keyword is another way of defining types. Subtypes are used to restrict the values that a certain type can take.

```
subtype <subtype_name> is <type_name> range <low> to <high>;
subtype <subtype_name> is <type_name> range <high> downto <low>;
```

Example:

```
subtype addr_int is integer range 0 to 65535;
subtype sub_enum_type is enum_type range a to m;
```

6 Package

Package header:

```
<libraries>

package <package_name> is
  <functions>
end package <package_name>;
```

Package body:

```
<libraries>

package body <package_name> is
  <functions>
end package body <package_name>;
```

Usage:

```
library IEEE;
use IEEE.std_logic_1164.all;
use <work_library>.<package_name>.all;
```

Example:

Package header:

```
library ieee;
use ieee.std_logic_1164.all;

package seg7enc is
  function encode7segment (di  : std_logic_vector;
                           dec : std_logic)
                           return std_logic_vector;
end package seg7enc;
```

Package body:

```
library ieee;
use ieee.std_logic_1164.all;

package body seg7enc is
  function encode7segment (di  : std_logic_vector;
                           dec : std_logic) return std_logic_vector is
  variable seg : std_logic_vector(7 downto 0) := "ZZZZZZZZ";
  begin
    case di(3 downto 0) is
      when "0000" => seg := "0000001" & not dec; --0
      when "0001" => seg := "1001111" & not dec; --1
      when "0010" => seg := "0010010" & not dec; --2
      when "0011" => seg := "0000110" & not dec; --3
      when "0100" => seg := "1001100" & not dec; --4
      when "0101" => seg := "0100100" & not dec; --5
      when "0110" => seg := "0100000" & not dec; --6
      when "0111" => seg := "0001111" & not dec; --7
      when others => seg := "11110010"; --r.
    end case;

    return seg;
  end encode7segment;
end package body seg7enc;
```

Usage:

```
library IEEE;
use IEEE.std_logic_1164.all;
use <work_library>.seg7enc.all;
```

7 Testbench

```
<libraries>

entity tb_ram_pos_ctrl is
  -- empty;
end tb_ram_pos_ctrl;

architecture beh of tb_ram_pos_ctrl is
  <components to test>
begin
  <port_mapping> -- often called UUT (unit under testing)

  <processes for testing>
end beh;
```

Example:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity tb_ram_pos_ctrl is
  -- empty;
end tb_ram_pos_ctrl;

architecture beh of tb_ram_pos_ctrl is
  component async_256kx16 is
    generic
    (
      ADDR_BITS : integer := 18;
      DATA_BITS : integer := 16;
      depth     : integer := 256*1024;

      TimingInfo   : boolean   := true;
      TimingChecks : std_logic := '1'
      );
    port
    (
      nCE : in    std_logic;            -- Chip Enable
      nWE : in    std_logic;            -- Write Enable
      nOE : in    std_logic;            -- Output Enable
      nUB : in    std_logic;            -- Byte Enable High
      nLB : in    std_logic;            -- Byte Enable Low
      A   : in    std_logic_vector(addr_bits-1 downto 0); -- Address Inputs A
      DQ  : inout std_logic_vector(DATA_BITS-1 downto 0) := (others => 'Z') -- Read/Write Data
      );
  end component async_256kx16;

  signal  refclk : std_logic := '0';      --Clock
  signal  arst   : std_logic;             --Asynchron rst

  --ram interface
  signal  adr      : std_logic_vector(17 downto 0); --Adresse
```

```
37    signal  dq       : std_logic_vector(15 downto 0);
38    signal  cs_ram_n : std_logic;          --Chip select RAM
39    signal  we_ram_n : std_logic;          --We enable strobe
40    signal  oe_ram_n : std_logic;          --Output enable (enabler SRAMs tristate utganger)
41    signal  lb_ram_n : std_logic;          --LSB
42    signal  ub_ram_n : std_logic;          --MSB
43
44    signal  load_run_sp : std_logic;
45
46    constant Half_Period : time := 10 ns;
47
48  begin
49    UUT : async_256kx16 generic map(18, 16, 256*1024, true, '1') port map(cs_ram_n, we_ram_n, oe_ram_n, ub_ram_n, lb_ran
50
51    refclk <= not refclk after Half_Period;
52
53    STIMULI :
54    process
55    begin
56      <do something>
57      wait for 6*Half_Period;
58      <do something>
59      wait for 6*Half_Period;
60    end process;
61  end beh;
```

8 State Machine (Mealy/Moore)

```
1   architecture beh of p_ctrl is
2     type state_type is (idle_st, some_st, other_st);
3     signal present_state, next_state: state_type;
4
5   begin
6     process(rst,clk)
7     begin
8       if (reset='1') then
9         present_state <= idle_st;  --default state on reset.
10      elsif (rising_edge(clk)) then
11        present_state <= next_state;   --state change.
12      end if;
13    end process;
14
15    state_machine :
16    process(rst,clk)
17    begin
18      if(rst = '1') then
19        <body>
20      elsif(rising_edge(clk)) then
21        if(present_state = idle_st) then
22          <body>
23        elsif(present_state = some_st) then
24          <body>
25        elsif(present_state = other_st) then
26          <body>
27        end if;
28      end if;
29    end process mes;
30  end architecture beh;
```

8.1 Mealy

- In a Mealy state machine outputs are a function of the current state and the inputs.
- Faster
  - Less delay (latency) from input to output
- More complicated decoding of outputs

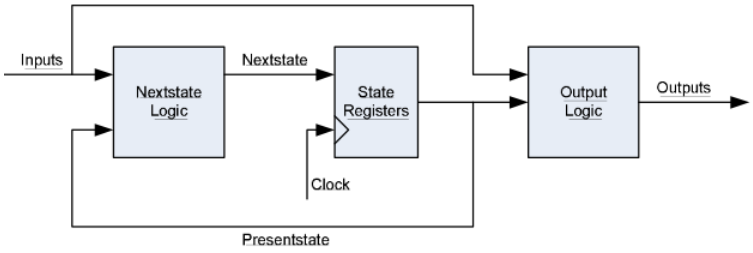

Figure 1: Mealy state machine

8.2 Moore

- In a Moore state machine outputs is a pure function of the current state (Present State)
- In a Moore machine it's one clock period delay from input to output

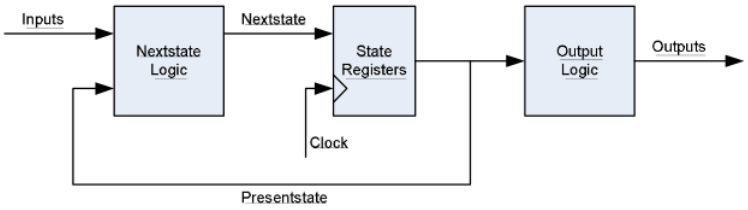

Figure 2: Moore state machine

9 ASM Chart

9.1 Condition Box

- Taking a clock period to implement
- X = 1
  - Assign each state
- Y
  - Set to 1 in this state and 0 otherwise
- Z ← A
  - Is assigned the value A in the shift from one state to another. The value is stored until it possibly changes the value in another state change (register storage)

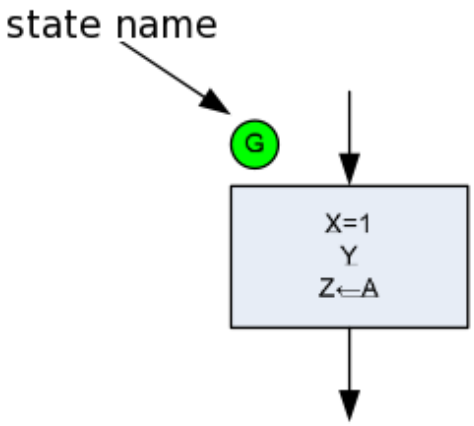

Figure 3: Condition box

9.2 Decision Box

- Two or more branchings based on the value of one or more inputs
- Decision box must follow a condition box and be associated with a condition
- Therefore, the decisions will be taken in the same period as the condition
  - Inputs must be stable at the start of clock flank (setup and hold times)

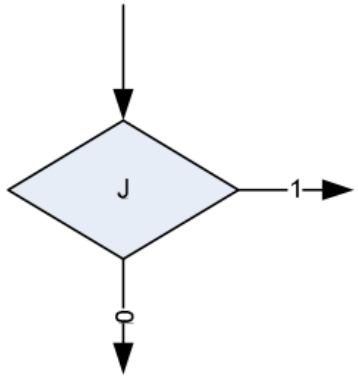

Figure 4: Decision Box

9.3 Conditional output values

- Must follow a decision box
- Output can change value during state
- Mealy output

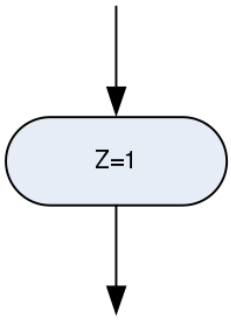

Figure 5: Conditional output values

9.4 Timing diagram

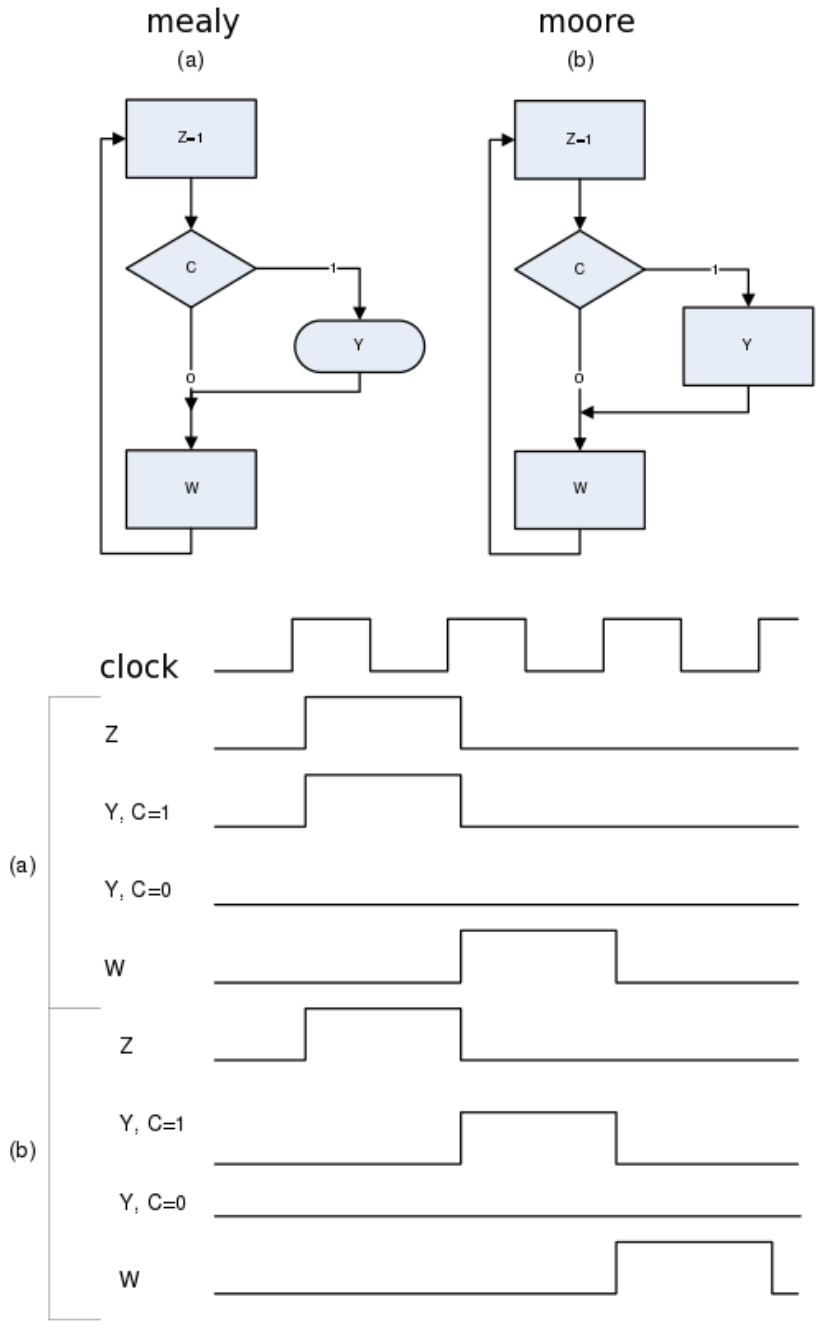

Figure 6: ASM - timing diagram

10 Synchronize signals from other systems

sp and *pos* comes from another system.

```
architecture beh of ctrl is

signal tmp_sp : signed(7 downto 0);
signal tmp_pos : signed(7 downto 0);
signal syncd_sp : signed(7 downto 0);
signal syncd_pos : signed(7 downto 0);

begin
  sync :
  process(clk)
  begin
    if(rising_edge(clk)) then
        tmp_sp <= sp;
        tmp_pos <= pos;
        syncd_sp <= tmp_sp;
        syncd_pos <= tmp_pos;
    end if;
  end process sync;
end architecture beh;
```

11 CRU

Clock divider:

```
library ieee;
use ieee.std_logic_1164.all;
entity clkdiv is
  port (
    rst : in std_logic; -- Reset
    mclk : in std_logic; -- Master clock
    mclk_div : out std_logic -- Master clock div. by 128
    );
end clkdiv;
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
architecture rtl of clkdiv is
  signal mclk_cnt : unsigned(6 downto 0);
begin
  P_CLKDIV: process(rst, mclk)
  begin
    if rst='1' then
      mclk_cnt <= (others => '0');
    elsif rising_edge(mclk) then
      mclk_cnt <= mclk_cnt + 1;
    end if;
  end process P_CLKDIV ;
  mclk_div <= std_logic(mclk_cnt(6));
end rtl;
```

Sync reset:

```
library ieee;
use ieee.std_logic_1164.all;
```

```
entity rstsynch is
  port (
    arst : in std_logic; -- Asynch. reset
    mclk : in std_logic; -- Master clock
    mclk_div : in std_logic; -- Master clock div. by 128
    rst : out std_logic; -- Synch. reset master clock
    rst_div : out std_logic -- Synch. reset div. by 128
    );
end rstsynch;
```

```
library ieee;
use ieee.std_logic_1164.all;
architecture rtl of rstsynch is
  signal rst_s1, rst_s2 : std_logic;
  signal rst_div_s1, rst_div_s2 : std_logic;
begin
  P_RST_0 : process(arst, mclk)
  begin
    if arst='1' then
      rst_s1 <= '1';
      rst_s2 <= '1';
    elsif rising_edge(mclk) then
      rst_s1 <= '0';
      rst_s2 <= rst_s1;
    end if;
  end process P_RST_0;
  P_RST_1 : process(arst, mclk_div)
  begin
    if arst='1' then
      rst_div_s1 <= '1';
      rst_div_s2 <= '1';
    elsif rising_edge(mclk_div) then
      rst_div_s1 <= '0';
      rst_div_s2 <= rst_div_s1;
    end if;
  end process P_RST_1;
  rst <= rst_s2;
  rst_div <= rst_div_s2;
end rtl;
```

CRU:

```
library ieee;
use ieee.std_logic_1164.all;
entity cru is
  port (
    arst : in std_logic; -- Asynch. reset
    refclk : in std_logic; -- Reference clock
    rst : out std_logic; -- Synchronized arst_n for mclk
    rst_div : out std_logic; -- Synchronized arst_n for mclk_div
    mclk : out std_logic; -- Master clock
    mclk_div : out std_logic -- Master clock div. by 128.
    );
end cru;
```

```
library ieee;
use ieee.std_logic_1164.all;
library unisim;
use unisim.all;
architecture str of cru is
  component bufg
```

```
7       port (i : in std_logic;
8             o : out std_logic);
9     end component;
10    component rstsynch is
11      port (arst : in std_logic; -- Asynch. reset
12            mclk : in std_logic; -- Master clock
13            mclk_div : in std_logic; -- Master clock div. by 128
14            rst : out std_logic; -- Synch. reset master clock
15            rst_div : out std_logic); -- Synch. reset div. by 128
16    end component rstsynch;
17    component clkdiv is
18      port (rst : in std_logic; -- Reset
19            mclk : in std_logic; -- Master clock
20            mclk_div : out std_logic); -- Master clock div. by 128
21    end component clkdiv;
22
23    signal rst_i : std_logic;
24    signal rst_local : std_logic;
25    signal rst_div_local : std_logic;
26    signal rst_div_i : std_logic;
27    signal mclk_i : std_logic;
28    signal mclk_div_local : std_logic;
29    signal mclk_div_i : std_logic;
30  begin
31    bufg_0: bufg
32      port map (
33        i => refclk,
34        o => mclk_i);
35    rstsynch_0: rstsynch
36      port map (arst => arst, -- [in] Asynch. reset
37                mclk => mclk_i, -- [in] Master clock
38                mclk_div => mclk_div_i, -- [in] Master clock div. by 128
39                rst => rst_local, -- [out] Synch. reset master clock
40                rst_div => rst_div_local); -- [out] Synch. reset mclk div. by 128
41    bufg_1: bufg
42      port map (
43        i => rst_local,
44        o => rst_i);
45    bufg_2: bufg
46      port map (
47        i => rst_div_local,
48        o => rst_div_i);
49    clkdiv_0: clkdiv
50      port map (rst => rst_i, -- [in] Reset
51                mclk => mclk_i, -- [in] Master clock
52                mclk_div => mclk_div_local); -- [out] Master clock div. by 128
53    bufg_3: bufg
54      port map (
55        i => mclk_div_local,
56        o => mclk_div_i);
57  -- Concurrent statements
58    rst <= rst_i;
59    rst_div <= rst_div_i;
60    mclk <= mclk_i;
61    mclk_div <= mclk_div_i;
62  end str;
```

12 Name Rules for VHDL source files

Fil-innhold	**Filnavn**
Entity	<design_unit>_ent.vhd
RTL architecture	<design_unit>_rtl.vhd
STR architecture	<design unit>_str.vhd
Behavorial architecture (simuleringsmodell ikke ment for syntese)	<design_unit>_beh.vhd
Configuration	<design_unit>_cfg.vhd
package def	<library_unit>_pkg.vhd
package body	<library_unit>_bdy.vhd
Testbench	tb_<design_unit>.vhd
testbench config	tb_<design_unit>_cfg.vhd

Part II
Theory

13 LUT (lookup table)

Example:
4-input LUT with content “7FFF” (hex)

in_1	in_2	in_3	in_4	LUT	LUT_{bin}
0	0	0	0		1
0	0	0	1	F	1
0	0	1	0		1
0	0	1	1		1
0	1	0	0		1
0	1	0	1	F	1
0	1	1	0		1
0	1	1	1		1
1	0	0	0		1
1	0	0	1	F	1
1	0	1	0		1
1	0	1	1		1
1	1	0	0		1
1	1	0	1	7	1
1	1	1	0		1
1	1	1	1		0

As we can see, this is a NAND-port.

13.1 Tasks

1. En 4-input Xilinx LUT med innhold "7FFF" (hex) realiserer en:

A	AND funksjon	
B	NAND funksjon	X
C	NOR funksjon	
D	XOR funksjon	
E	OR funksjon	

2. En 3-input Xilinx LUT med innhold "01" (hex) realiserer en:

A	AND funksjon	
B	NAND funksjon	
C	NOR funksjon	X
D	XOR funksjon	
E	OR funksjon	

3. En 3-input Xilinx LUT med innhold "80" (hex) realiserer en:

A	AND funksjon	X
B	OR funksjon	
C	NAND funksjon	
D	NOR funksjon	

4. En 3-input Xilinx LUT (look-up table) med innhold "7F" (hex) realiserer en

A	AND funksjon	
B	OR funksjon	
C	NAND funksjon	X
D	NOR funksjon	

5. En 3-input Xilinx LUT (look-up table) med innhold "FE" (hex) realiserer en

A	AND funksjon	
B	OR funksjon	X
C	NAND funksjon	
D	NOR funksjon	

14 Multiple Choice

1. Høyhastighets serielinker

A	Differensielle signaler brukes for å redusere støy problemer.	X
B	Høyhastighetslinker har i tillegg til de differensielle datalinjene også differensielle klokkelinjer.	
C	De differensielle linjene fra en sender kan gå til opptil 4 mottagere.	
D	8B/10B signal koding brukes for å unngå flere enn 8 påfølgende like bit.	(5 bit)
E	For PCIe gen. 1 er faktisk datarate 2.0 Gbit/s med 8B/10B koding som gjør at linjens baudrate blir 2.5 Gbit/s.	X ($\frac{2.5}{10} \cdot 8 = 2.0$)
F	Grunnen til at en overført firkantpuls ved høy datarate kan bli lik et sinussignal er at høyfrekvent frekvensinnhold har blitt kraftig dempet	X
G	Konfigurasjon av parametere i transceiver muliggjør design med forskjellige kommunikasjonsstandarder	X
H	Pre-emphasis motvirker dempning i overført signal	X
I	"Comma"-tegn brukes for å dele opp lange bitstrenger	

2. Design

A	En hard IP kjerne tar vanligvis mere plass enn en tilsvarende myk IP kjerne.	
B	Med en Digital Clock Manager (DCM) modul kan man øke klokkesignalet til det firedobbelte og det genererte klokkesignalet vil være i fase med inngangsklokken.	X
C	I en Xilinx FPGA har set inngangen til en flip-flop lavere prioritet enn reset inngangen.	X
D	Initialverdien etter deklarasjon av et signal av typen std_logic vil være 'X'.	
E	To signaler av typen std_logic med verdiene 'Z' og '0' som driver samme signal får verdien 'X'.	
F	Block RAM som ikke brukes kan fjernes fra FPGA'en ved syntese.	
G	En Xilinx Block RAM har to uavhengige porter som begge kan leses fra og skrives til samtidig.	X
H	Tilbakekoblingssløyfer med flip-flop'er kan brukes i en FPGA.	X
I	Asynkront design anbefales ikke i en FPGA.	X
J	En BUFG modul kan bare brukes til klokkesignaler.	
K	Integrering av et helt system med prosessor på en krets gir en mer kompakt løsning som prismessig kan være gunstig.	X
L	En hard prosessorkjerne er vanligvis raskere (høyere klokkefrekvens) enn en myk prosessorkjerne.	X
M	Gigabit Transceivere finnes som myke kjerner.	
N	ROM kan ikke lages av harde Block RAM (BRAM) moduler.	
O	RAM kan lages av LUT'er.	X
P	En Xilinx Block RAM har to uavhengige porter som begge kan leses fra og skrives til samtidig.	X
Q	Selv om antall input til en funksjon er konstant, øker forbruket av LUT'er alltid ved økning av kompleksiteten til funksjonen.	
R	I Spartan 3 teknologi er vanligvis forsinkelsen gjennom en LUT lengre enn typisk forsinkelse mellom LUT'er.	
S	Det er begrenset hvor mange adskilte klokkerlinjer fra BUFG'er som finnes i en FPGA i forhold til i en ASIC.	X
T	I en Xilinx FPGA har den asynkrone set inngangen til en flipflop/register høyere prioritet enn den asynkrone reset inngangen.	
U	Med en Digital Clock Manager modul kan man øke klokkesignalet til det dobbelt og det genererte signalet kan være i fase med inngangsklokken.	X
V	En hard prosessorkjerne tar vanligvis mindre plass enn en myk prosessorkjerne.	X
W	Xilinx Block RAM'er kan fjernes fra FPGA'en hvis de ikke brukes for å spare plass.	
X	I et klokkedomene klokkes alle registre (flipflop'er) med samme klokkesignal.	X
Y	FPGA er egnet for asynkron logikk.	
Z	Bruk av dedikert mentekjede (carry chain) gjør at det blir mindre tilgjengelig logikk i FPGA'en og bruken bør derfor begrenses.	
Æ	I serielle linker har 8B/10B encoding mindre overhead enn 64B/66B enkoding.	X
Ø	Serielle linker er vanligvis enveissignaler (uni-direksjonale).	
Å	DSP-modulene "MAC" står for "Multiply-And-Compare	

1	DSP-modulene til Xilinx er myke moduler som kan fjernes for å spare plass i FPGA'en	
2	Intellectual Property (IP) er i FPGA teknologi en betegnelse på myke ferdigutviklede moduler.	X
3	En BUFG modul kan bare brukes til klokkesignaler.	
4	Retiming brukes av syntheseverktøy for å lettere oppnå timingkrav og det gir aldri økt forbruk av registere (flip-flop'er).	
5	I Spartan 3 teknologi er vanligvis forsinkelsen gjennom en LUT lengre enn typisk forsinkelse mellom LUT'er.	
6	En hard IP kjerne tar mindre plass enn en tilsvarende myk IP kjerne.	X
7	Det er samme begrensning med antall klokkenett i ASIC teknologi som i FPGA teknologi.	
8	Et problem med DCM moduler er at "jitter" øker.	
9	En hard IP kjerne som ikke brukes frigjør plass til annen logikk.	
10	En Flash FPGA er umiddelbart aktiv etter strømtilkobling.	X
11	Det må vanligvis brukes registere (flip-flop'er) i en tilbakekobling i FPGA.	X
12	Det er vanligvis kortere delay i wires mellom to LUT'er enn gjennom en LUT.	
13	I en Xilinx FPGA har set inngangen til en flipflop/register lavere prioritet enn reset inngangen.	X
14	FPGA egner seg dårlig for pipelining pga. få registere.	
15	En Xilinx Block RAM har to uavhengige porter som begge kan leses fra og skrives til.	X

3. FPGA-teknologi

A	Forbindelseslinjer mellom LUT'er har vanligvis større tidsforsinkelse enn tidsforsinkelsen gjennom LUT'er i SRAM-teknologi.	X
B	En FPGA krets basert på Flash er umiddelbart aktiv etter strømtilkobling.	X
C	SRAM-teknologi er vel så motstandsdyktig mot stråling som antifuse teknologi.	
D	Konfigurasjonsfiler er alltid så små at det er raskt å bytte til en ny konfigurasjon.	
E	JTAG porten kan brukes både til konfigurasjon og til debugging.	X
F	En SRAM FPGA er det ikke mulig å finne ut hvordan fungerer (dvs. ved "reverse-engineering") ut i fra konfigurasjonsfilen.	
G	Block RAM'er har en kjent initialverdi etter konfigurering.	X
H	JTAG porten er eneste metode for å få konfigurert en Xilinx FPGA.	
I	En antifuse FPGA kan reprogrammeres 1 gang.	
J	En FPGA i master-modus styrer selv nedlastning av konfigurasjonen ved oppstart.	X

4. Klokkenet, DCM og design

A	To klokkesignaler ut av Xilinx DCM modulen hvor en klokke er multiplisert 1.0 med original frekvens og den andre klokken er multiplisert 2.5 med original frekvens er i fase med hverandre.	
B	DCM kan forsinke en generert klokke slik at den er i fase med inngangsklokken.	X
C	Antall nivåer med logikk i en FPGA mellom klokkede flipflop'er/registere har betydning for maksimal klokkefrekvensen.	X
D	Et differensielt ledningspar er mere følsomt for støy fra eksterne kilder enn en enkeltleder.	
E	FPGA egner seg for pipelining pga. mange registere.	X

5. Metastabilitet og timing constraints

A	Det er ikke enkelt å oppdage metastabilitet ved simulering.	X
B	Etter en tid i metastabil tilstand vil alle flip-flop'er alltid gå til verdien '0'.	
C	Deaktivering av et eksternt reset signal må alltid synkroniseres for alle klokkedomener hvor reset brukes.	X
D	PERIOD timing constraints har høyere prioritet enn FROM-TO constraint.	
E	OFFSET IN constraint kan bruke internt genererte klokker fra DCM.	

6. Signalverdier i VHDL av typen std_logic:

A	Initialverdien etter deklarasjon til et signal av typen std_logic er '0'.	
B	To signaler av typen std_logic med verdiene '0' og '0' som driver samme signal får verdien '0'.	X
C	To signaler av typen std_logic med verdiene '0' og '1' som driver samme signal får verdien 'X'.	X
D	To signaler av typen std_logic med verdiene 'Z' og '1' som driver samme signal får verdien 'Z'.	
E	To signaler av typen std_logic med verdiene 'Z' og '1' som driver samme signal får verdien 'X'.	

7. Konfigurasjon og lagringsteknologi

A	JTAG porten kan brukes både til konfigurasjon og til debugging.	X
B	En FPGA i master-modus styrer selv nedlasting av konfigurasjon ved oppstart.	X
C	En FPGA må alltid konfigureres serielt hvis den er i slave-modus	
D	SRAM FPGA'er har høyere sikkerhet enn Antifuse FPGA'er	
E	Antifuse FPGA'er kan bare konfigureres en gang.	X

8. VHDL og simulering

A	Det er raskere og enklere å simulere med variabler enn med signaler i en process.	X
B	Initialverdien etter deklarasjon av et signal av typen std_logic vil være ’0’.	
C	Alle variabler som er deklarert i en process vil ikke bli satt tilbake til sin initialverdi neste gang processen utføres.	X
D	For hver deltacycle går tiden et picosekund.	
E	Etter en tid i metastabil tilstand vil alle flip-flop’er alltid gå til verdien ’0’.	

9. Konfigurasjon og rekonfigurasjon av Xilinx FPGA

A	En SRAM FPGA er det ikke mulig å finne ut hvordan fungerer (dvs. ved "reverse-engineering") ut i fra konfigurasjonsfilen.	
B	Ved delvis rekonfigurasjon vil de delene av kretsen som ikke blir rekonfigurert også være inaktive under rekonfigureringen.	
C	Et problem ved dynamisk rekonfigurering er vanligvis for lang rekonfigureringstid.	X
D	En mikroprocessor kan konfigurere en FPGA som er i slave mode.	X
E	Block RAM’er har ikke en kjent initialverdi etter konfigurering.	

10. Høyhastighets-linker

A	Høyhastighetslinker har i tillegg til de differensielle datalinjene også differensielle klokkelinjer.	
B	For PCIe gen. 1 er faktisk datarate 2.0 Gbit/s med 8B/10B koding som gjør at linjens baudrate blir 2.5 Gbit/s.	X
C	Det er bare når commategn sendes at sender utfører preemphasis.	
D	Commategn brukes for å dele opp lange bitstrenger med mange påfølgende ’1’ og mange påfølgende ’0’.	
E	De differensielle linjene fra en sender kan gå til opptil 4 mottagere.	

11. Design og ASIC

A	FPGA egner seg bedre enn ASIC i produkter med krav om lavt effektforbruk.	
B	I en FPGA skal det alltid brukes register i en tilbakekoblingssløyfe.	X
C	Det er ingen begrensning av antall klokkedomener i en FPGA.	
D	En ASIC krets kan inneholde analog og digital elektronikk i samme krets.	X

12. Konfigurasjon av FPGA

A	JTAG porten er eneste metode for å få konfigurert en Xilinx FPGA.	
B	Alle registere i en FPGA får en kjent verdi ved konfigurering.	X
C	En antifuse FPGA kan reprogrammeres 1 gang.	
D	Konfigurasjonsfiler er alltid så små at det er raskt å bytte til en ny konfigurasjon.	
E	En FPGA i master-modus styrer selv nedlastning av konfigurasjonen ved oppstart	X
F	"Daisy-chaining" gjør at flere FPGA-er kan ha et felles konfigurasjonsminne	X
G	En FPGA må alltid konfigureres parallelt hvis den er i slave-modus	
H	JTAG-porten er egentlig tiltenkt testing men kan også brukes til konfigurasjon	X

13. Verktøy og metodikk

A	Formell verifikasjon kan brukes for å sjekke at VHDL koden og ferdig nettliste er like.	X
B	Statisk timing analyse gjør at RTL simulering av designet i en VHDL testbenk er unødvendig.	
C	En BFM (Bus Functional Model) kan erstatte prosessor bus interface i en testbenk	X
D	Retiming kan utføres under syntese	X

14. DCM og klokkenett for Xilinx

A	Klokkesignalene ut av Xilinx DCM modulen er ikke i fase med hverandre.	
B	Klokkenett kan bare brukes for klokke signaler.	
C	DCM kan forsinke en generert klokke slik at den er i fase med inngangsklokken.	X
D	En Xilinx BUFG modul brukes for hvert klokkenett.	X

15. DSP konstruksjon

A	Xilinx har harde DSP moduler som har MAC (Multiply and Accumulate) funksjon.	X
B	Med verktøy fra Xilinx og Mathworks/Matlab kan det genereres FPGA moduler uten at konstruktøren trenger å skrive VHDL kode.	X
C	Flere DSP operasjoner kan gjøres i parallell i en DSP prosessor enn i en FPGA.	
D	Det er vanlig at FPGA har harde DSP moduler for Analog-til-Digital og Digital-til-Analog konvertering.	

16. Virtuelle componenter og IP (Intellectual Property) for Xilinx FPGA

A	Gigabit Transceivere finnes som myk kjerne.	
B	ROM kan lages av harde Block RAM (BRAM) moduler	X
C	En myk prosessorkjerne har vanligvis lavere maksimal klokkehastighet enn en hard prosessorkjerne.	X
D	RAM kan lages av LUT'er.	X
E	En IP gitt som ikke-kryptert kildekode er normalt mer effektiv enn en IP gitt som forhåndsrutet IP	
F	Intellectual Property er betegnelsen på ferdigutviklede blokker	X
G	MicroBlaze er eksempel på en IP	X
H	Det er enkelt å gjennbruke en IP fra en FPGA-produsent på kretser fra andre produsenter	

17. Timing constraints for Xilinx

A	FFS og PADS er eksempler på forhåndsdefinerte timing grupper.	X
B	From-To constraint har lavere prioritet enn Period constraint.	
C	Timing krav spesifiseres i en User Constraint File (UCF).	X
D	Ved å "constraine" inngangsklokken til Xilinx DCM modulen vil alle utgangsklokker være timing "constrainet".	X

18. En Xilinx FPGA kan inneholde harde kjerner for:

A	Multiplikasjon	X
B	MicroBlaze prosessor	
C	MAC	X
D	Divisjon	

19. Gigabit Transceivere

A	En Transceiver modul har FIFO i senderetningen og FIFO i mottaksretningen.	X
B	Såkalte "comma" karakterer brukes i forbindelse med utjevning (equalization) i mottageren.	
C	8B/10B signal koding brukes for å unngå flere enn 8 påfølgende like bit.	
D	Differensielle signaler brukes for å redusere støy problemer.	X

20. Kan variabler i VHDL deklareres i:

A	Process	X
B	Procedure	X
C	Architecture	
D	Function	X

21. Kretsteknologier

A	En logikkblokk i en FPGA består normalt av en Look-Up Table (LUT) etterfulgt av en vippe (flip-flop)	X
B	En PLA består av OP-porter etterfulgt av en vippe AND-porter	
C	I en PAL er tilkoblingene til AND-portene ikke programmerbare	X
D	I en "full custom" ASIC har designeren full kontroll over hvert maskelag i kretsen	X
E	FPGA har normalt mindre logikk enn en CPLD	
F	FPGA er mer praktisk å programmere enn (S)PLD	X
G	Celler i CPLD har mye til felles med PAL	X
H	CPLD har eksistert lenger enn SPLD	

22. Lagringsteknologi

A	I en CPLD lagres normalt konfigurasjonen i SRAM	
B	En FPGA basert på antifuse-teknologi er ikke reprogrammerbar	X
C	En FPGA basert på antifuse-teknologi kan ikke slettes med UV-lys	X
D	En EPROM kan slette sitt innhold med en høy spenning	
E	En SRAM kan kun programmeres et begrenset antall ganger	
F	Antifuse-teknologien baserer seg på å opprette forbindelser når en krets programmeres	X
G	EPROM er basert på å lagre ladning på en floating gate i en transistor	X
H	SRAM er velegnet til permanent lagring	
I	Flash teknologien er en videreutvikling av (E)EPROM	X
J	Reprogrammeringstiden for SRAM er lenger enn for Flash/EPROM	

23. Optimalisert FPGA design

A	Selv om antall input til en funksjon er konstant, øker forbruket av logikk med kompleksiteten til funksjonen	
B	Antall nivåer med logikk i en FPGA mellom klokkede vipper har betydning for maksimal klokkefrekvensen	X
C	Klokketre i en FPGA bør unngåes hvis en skal lage et effektivt synkront design	
D	Dedikert mentelogikk kobler sammen logikk for hurtig menteforplantning	X
E	Bruk av dedikert mentelogikk gjør at det blir mindre tilgjengelig logikk i FPGA-en og bruken bør derfor begrenses	

24. Prosessorkjerner

A	En hard kjerne er implementert fysisk i FPGA-en ved produksjon av kretsen	X
B	Kombinasjon av prosessor og logikk på en FPGA gir liten fleksibilitet i bestemmelsen av hva som blir programvare og hva som blir maskinvare	X
C	Separat buss mellom prosessor og minne gir lite gevinst og bør unngåes	
D	Integrering av et helt system på en krets gir en mer kompakt løsning som også prismessig kan være gunstig	X

25. Sykelbasert simulering

A	Dette er et alternativ til hendelsesbasert simulering	X
B	En dropper å simulere hver hendelse i en krets men benytter boolske uttrykk på inngangene til registre	X
C	Metoden kan kombineres med hendelsesdrevet simulering for simulering av en krets	X
D	En ulempe, sammenlignet med alternative måter å simulere på, er at tiden for simulering øker betydelig	

26. Syntese

A	Syntese gjøres normalt etter "place-and-route"	
B	Syntese med informasjon om faktiske tidsforsinkelser i FPGA-en kan gi høyere maksimal klokkefrekvens	X
C	Plassering av registre (vipper) i forhold til logikk har normalt ingen betydning for ytelsen	
D	Resyntese for optimalisering av kritisk signalvei kan være gunstig	X
E	Mengden logikk og forbindelseslinjer mellom flip-floper i et design påvirker hva som blir maksimal klokkefrekvens	X
F	Endring av hvilke flip-floper som benyttes i en FPGA kan påvirke maksimal klokkehastighet til et design	X
G	Den viktigste grunnen til å justere på plassering av et design i en FPGA er å minske størrelsen på designet	

27. SystemC

A	Språket er definert av en spesifikk verktøyleverandør som selger designverktøy	
B	Språket er basert på C/C++	X
C	Språket er bedre egnet til verifikasjon enn syntese	X
D	Språket kan spesifisere kode på flere abstraksjonsnivåer enn VHDL	X
E	SystemC brukes i dag like ofte som VHDL for FPGA design	

28. Programmerings-teknologier for programmerbar logikk

A	En krets basert på Flash krever ekstern rekonfigureringsfil ved oppstart	
B	Antifuse bruker lite effekt (i et system i drift)	X
C	FPGA med antifuse-teknologi egner seg godt til prototyping	
D	En krets basert på Flash er umiddelbart aktiv etter strømtilkobling	X
E	SRAM-teknologi er vel så motstandsdyktig mot stråling som antifuse	

29. Størrelse på FPGA logikkblokker

A	En finkornet (fine grained) FPGA-blokk kan kun realisere enkle funksjoner	X
B	Fordelen med finkornede blokker er at rutingressursene som kreves blir begrenset	
C	Brukeren konfigurerer en gitt FPGA til å enten være grovkornet eller finkornet	
D	Utfordringene med grovkornede (coarse grained) blokker er å utnytte dem fullt ut	X

30. Klokkestyring

A	Klokkesignal brukes normalt ikke i et synkront design med flip-floper	
B	Klokketre skal begrense at klokkeflanker ankommer til forskjellig tid rundt i en krets	X
C	"Clock managers" kan generere klokker med forskjellig frekvens	X
D	En ulempe med "Clock managers" er at problemet med jitter øker	

31. (A)synkront design

A	I et synkront design klokkes normalt alle flip-floper med samme klokkesignal	X
B	Problemet med asynkron logikk er at spesifikasjon av timing blir vanskelig og uforutsigbar	X
C	Innføring av ekstra flip-floper for synkronisering bør unngås i design	
D	Det er ingen ulemper med å kombinere synkron og asynkron styring (set/reset) av en flip-flop med hensyn på forbruk av ressurser i en FPGA	

32. Verifikasjon

A	Hendelsebasert (event driven) simulering er normalt uten timinginformasjon	
B	I statisk timinganalyse modelleres normalt alle porter med lik tidsforsinkelse	X
C	Formell verifikasjon kan finne andre feil enn de som finnes ved simulering	X
D	Design beskrevet i høynivåspråk gir raskere simulering enn for tilsvarende beskrivelse i lavnivåspråk	X

33. Myke og harde prosessorkjerner

A	Samme type prosessor finnes både som myk og hard kjerne til Xilinx FPGA-er	
B	En myk kjerne er raskere (høyere klokkefrekvens) enn en hard kjerne	
C	En myk kjerne er ikke så plasseffektiv som en hard kjerne	X
D	EDK kan benyttes til design med prosessorkjerner	X
E	MicroBlaze er eksempel på en hard kjerne	

34. Kodestil for FPGA og ASIC

A	Samlebåndsprosessering (pipelining) kan være med på å øke maksimal klokkefrekvens i et design	X
B	Samlebåndsprosessering (pipelining) vil ofte medføre at en bruker færre vipper i et design	
C	Tilbakekoblingssløyfer der vipper inngår må ikke brukes i en FPGA	
D	Asynkront design er mulig i en ASIC, men anbefales ikke i en FPGA	X

35. Valg mellom ASIC og FPGA

A	FPGA er bedre enn ASIC ved komplekse design	
B	Det er bedre plass i en ASIC enn i en FPGA når kretsene har omtrent samme fysiske størrelse	X
C	Prototyping av ASIC på FPGA bør unngåes på grunn av forskjell i kodestil	
D	ASIC har lang utviklingstid men de første kretsene er billige å produsere	

36. Rekonfigurering av aktiv FPGA

A	Virtuell maskinvare er en betegnelse som brukes om denne teknikken	X
B	Teknikken muliggjør å kunne utføre en større oppgave enn det kretsen tilsynelatende har logikk til	X
C	Effektforbruket kan ofte øke ved denne metoden	
D	Lang rekonfigureringstid er en av hovedutfordringene	X
E	Det vil være ønskelig med denne metoden å rekonfigurere hele kretsen og ikke kun en begrenset del av den	

Index