Master

E3.06
AC5
ISE3.5

IMPERIAL COLLEGE LONDON

DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING
EXAMINATIONS 2010

MSc and EEE/ISE PART III/IV: MEng, BEng and ACGI

VHDL AND LOGIC SYNTHESIS

Wednesday, 12 May 10:00 am

Time allowed: 3:00 hours

There are FOUR questions on this paper.

Question 1 is COMPULSORY
Answer question 1 and any TWO of questions 2-4
Question 1 carries 40% of the marks, questions 2-4 each carry 30% of the marks.

Any special instructions for invigilators and information for candidates are on page 1.

Examiners responsible First Marker(s) : T.J.W. Clarke, T.J.W. Clarke
Second Marker(s) : G.A. Constantinides, G.A. Constantinides

Special Information for Invigilators: none.

Information for Candidates

VHDL language reference can be found in the booklet VHDL Exam Notes.

Unless otherwise specified assume VHDL 1993 compiler.

All packages that must be explicitly referenced with USE, e.g. IEEE.numeric_std, must be explicitly noted in each answer: semantically correct LIBRARY and USE statements may be omitted where this is done.

The Questions

1.

a) Write a synthesisable process *P1* which implements an unsigned 10 bit positive edge triggered counter with output r and 10 bit unsigned inputs p, q, 1 bit inputs *reset,clk* and count sequence: p, p+1, p+2, ..., q-1, q, p, p+1. On *reset* = 1 the counter must synchronously load p. If $q < p$ the output will wrap round and

..., 1023, 0, ...

will be in the count sequence. State the VHDL types used for p, q, r.

[8]

b) Draw a waveform diagram of the signals *clk*, *b*, *c*, *d* from architecture TEST in Figure 1.1. Dimension your diagram with the simulation time and simulation delta of each signal transition.

[8]

c) Write a synthesisable combinational process *P2* with unsigned output $y(3:0)$, and input x(7:0). The value of y is equal to the number of inputs $x(7)$,..., $x(0)$ which are equal to 1. Thus if x = "10100000", $y = 2$.

[8]

d) State three rules which must be satisfied by a well-formed combinational VHDL process which are *not* usually required by a VHDL compiler.

[8]

e) A combinational process *P3* computes $q = a*b+c$ where a,b,c are integers in the ranges specified in Figure 1.2. State appropriate types and lengths for q, a, b, c signals and write a synthesisable VHDL definition of *P3*.

[8]

```
ARCHITECTURE TEST of XX IS
    SIGNAL clk, b,c,d: std_logic;
BEGIN

    PCLK: PROCESS
    BEGIN
        clk <= '0';
        WAIT FOR 5 ns;
        clk <= '1';
        WAIT FOR 5 ns;
    END PROCESS PCLK;

    b <= not clk;

    P1: PROCESS(clk,d,b)
        VARIABLE  x: std_logic;
    BEGIN
        d <=  clk xor b;
        x := d;
        c <= not x;
    END PROCESS P1;

END ARCHITECTURE TEST;
```

Figure 1.1

	min	max
a	-16	15
b	0	3
c	0	31

Figure 1.2

2. The entity *testfunc* in Figure 2.1 generates outputs *y* from input *clk*. This question concerns the operation of *testfunc* when simulated pre-synthesis, and when synthesised.

a) What are the registered (clocked) and combinational signals in *testfunc*? Indicate how the value of each registered signal is initialised at the start of pre-synthesis simulation.

[5]

b) Complete the diagram in Figure 2.2 showing the simulated outputs *y* when the inputs to *testfunc* are as shown in the Figure 2.2. Indicate the time of all transitions on the outputs.

[10]

c) Discuss, for each output *y*(*i*), whether or not you expect post-synthesis simulation to be similar to pre-synthesis simulation, giving reasons for your views.

[10]

d) Discuss, for each output dissimilar in post and pre synthesis simulation, how you would change the code to make post-synthesis and pre-synthesis waveforms coincide, or what problems prevent this.

[5]

```
ENTITY testfunc IS
PORT(
    y: OUT std_logic_vector(4 DOWNTO 0);
    clk: In std_logic
);
END ENTITY testfunc;

ARCHITECTURE xx OF testfunc IS

    TYPE state IS(as,bs,cs);
    SIGNAL z: std_logic := '0';
    SIGNAL s: state;
    SIGNAL u,v,w,clk1: std_logic;

BEGIN

    P0: PROCESS(clk)
        VARIABLE c : std_logic;
    BEGIN
        c := not clk;
        FOR n in 1 TO 4 LOOP
            v <= TRANSPORT c AFTER n * 1 ns;
            c := not c;
        END LOOP;
    END PROCESS P0;

    P1: PROCESS
    BEGIN
        WAIT UNTIL clk'EVENT and clk='1';
        clk1 <= not clk AFTER 10 ns;
        w <= not w;
        z <= not z;
        u <= '0';
        CASE s IS
            WHEN as => s <= bs; u <= '1';
            WHEN bs => s <= cs;
            WHEN cs => s <= as;
        END CASE;
    END PROCESS P1;

    y <= (clk1, z, u, v, w);

END ARCHITECTURE xx;
```

Figure 2.1. *Testfunc* entity and architecture

Figure 2.2. Pre-synthesis simulation waveforms

3. Entity *priority_encode8* in Figure 3.1 has 8 inputs x, 3 bit output y, and 1 bit input *pi*, output *po*. The table in Figure 3.2 indicates the combinational outputs y, *po* as a function of x, *pi*. The output y implements an 8 bit priority encoding function if $pi = 0$, such that the unsigned numeric value of y is the maximum index k for which $x(k) = 1$. If $pi = 1$, or all $x(k)$ are 0, y is high impedance. The output *po* is 1 if either *pi* is 1 or at least one of $x(k)$ is 1.

a) Write a synthesisable VHDL architecture for entity *priority_encode8*.

[15]

b) Figure 3.3 shows how 3 *priority_encode8* blocks *EMS*, *E1*, *E0* can be used to implement a 16 bit priority encoding function. Explain how this works. You may assume $x \neq 0$.

[5]

c) Write a synthesisable VHDL architecture for entity *big_priority_encode* in Figure 3.1 using $n + 1$ instances of entity *priority_encode8* where $2 \leq n \leq 8$, which implements an $8n$ bit priority encoding function. You may assume $x \neq 0$.

[10]

```
ENTITY priority_encode8 IS
PORT( x: IN std_logic_vector(7 DOWNTO 0);
      pi: IN std_logic;
      y: OUT std_logic_vector(2 DOWNTO 0);
      po: OUT std_logic );
END priority_encode8;
```

```
ENTITY big_priority_encode IS
GENERIC( n:INTEGER);
PORT( x: IN std_logic_vector(n*8-1 DOWNTO 0);
      y: OUT std_logic_vector(5 DOWNTO 0) );
END big_priority_encode;
```

Figure 3.1 *priority_encode8* and *big_priority_encode* entities

pi	*x*	*po*	*y*
1	XXXXXXXX	1	high impedance
0	00000000	0	high impedance
0	00000001	1	000
0	0000001X	1	001
0	000001XX	1	010
0	00001XXX	1	011
0	0001XXXX	1	100
0	001XXXXX	1	101
0	01XXXXXX	1	110
0	1XXXXXXX	1	111

Figure 3.2 Truth table of *priority_encode8*. X indicates don't care.

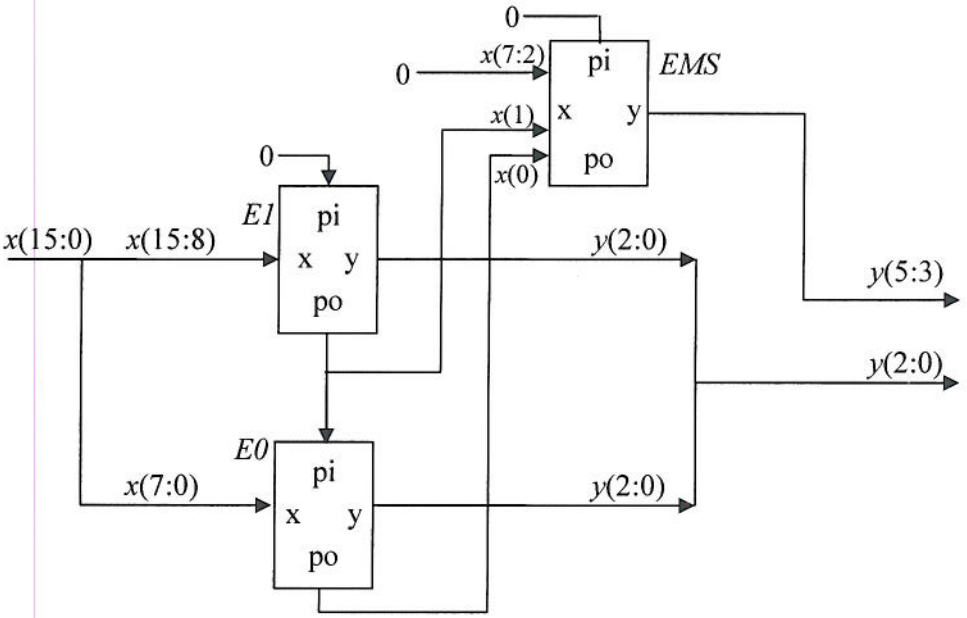


Figure 3.3. 16 bit priority encoding block

4. A CPU entity *datapath* consists of three units: *three_port_ram*, *rotate*, and *alu*, defined as in Figure 4.1, and connected as in Figure 4.2.

a) Write a VHDL entity for *datapath* (all ports except *clk* must be std_logic_vector). State inputs to *datapath* which will initialise word 0 of the RAM to 0 in a single clock cycle.

[6]

b) Write a synthesisable architecture for *datapath* using one or more process statements to represent the constituent units.

[24]

three_port_ram

16 word X 32 bit RAM

Ports

P	read	combinational
Q	read	combinational
R	write	on *clk* positive edge

rotate

$0 \le n \le 31$

i	Y
$0 \le i \le 31 - n$	$Y(i) = X(n+i)$
$32 - n \le i \le 31$	$Y(i) = X(n+i-32)$

alu

or, **and**, **not**, are bitwise logical operators.

func	F
0	B
1	A+ B
2	A **or** B
3	A **and** B
4	A - B
5	**not** B
6	A **or not** B
7	(A * B)(31:0)

Figure 4.1. Definition of units *three_port_ram*, *rotate*, *alu*

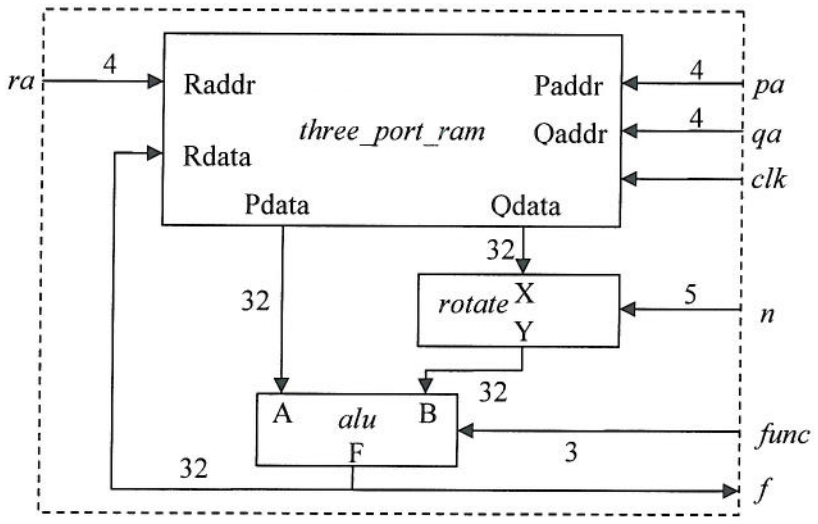


Figure 4.2. Entity *datapath*

VHDL Sequential (Behavioural) Statements - inside PROCESS body

Meta-language	
[]	*optional part*
*	*0 or more repetitions*
{}	*grouping brackets*

- See also the dataflow statements which can also occur inside a PROCESS - previous slide
- Sequential statements (except WAIT) take zero simulation time to execute

```
variable := value ; -- variable assignment
NULL; -- empty statement
WAIT [ ON signal ]  [ UNTIL condition ] ;
WAIT FOR time ;

IF condition THEN statements
[ ELSIF condition THEN statements ]
[ ELSE statements ]
END IF ;

FOR var IN range LOOP
   for-statements
END LOOP;

CASE var IS
    WHEN case1 => statements
    [ WHEN case2 => statements ]
    [ WHEN OTHERS => statements ]
END CASE ;
```


(Sequential also) VHDL Dataflow statements

```
signal <= [ TRANSPORT ] value [AFTER time ];

subprogram( para1 [, para2 [ , ... ] ] );

[ ASSERT condition ] REPORT message-if-false
   [ SEVERITY level ] ;

level ::= note | warning | error | failure
```

(Dataflow only)

{} meta-language grouping brackets - not VHDL

```
signal <=  value WHEN condition
         [ ELSE value WHEN condition ]
           ELSE value ;

WITH sel SELECT
  signal <=  value WHEN sel-case
           [ , value WHEN sel-case ]
             [ , value WHEN OTHERS ] ;

label :FOR var IN range GENERATE
     dataflow-statements
END GENERATE;

IF condition GENERATE
     dataflow-statements
END GENERATE;
```

```
label :  {ENTITY entity-name} | component-name
   GENERIC MAP( gen-map [ , gen-map ] )
   PORT MAP( port-map  [, port map] ) ;

[ label : ] PROCESS [ ( sensitivity-list ) ]
         process-declarations
   BEGIN sequential statements
   END PROCESS [ label ];

[ label : ] BLOCK local-declarations
   BEGIN dataflow-statements
   END [ label ] ;
```


Signal Attributes & Design Units

Signal Attributes

SIGNAL x has type T

Expression	Type	Description
x'EVENT	Boolean	TRUE if event on x
x'DELAYED(*del*)	T	x delayed by time *del*
x'LAST_VALUE	T	value of x before last or current event
x'LAST_EVENT	TIME	elapsed time from last event
x'STABLE(*tim*)	Boolean	FALSE if event on x within time *tim*

Design Units

```
ENTITY myentity IS
GENERIC( signal-list );
PORT( port-signal-list );
END myentity;

ARCHITECTURE archname OF entityname IS
declaration-statements
BEGIN
dataflow-statements
END archname;

PACKAGE mypackage IS
declarations
END [mypackage];

PACKAGE BODY mypackage IS
function and procedure body definitions
END PACKAGE BODY mypackage;
```


VHDL array syntax

```
unconstrained_array_type ::= STD_LOGIC_VECTOR | SIGNED | UNSIGNED | etc
range ::=   low TO high | high DOWNTO low |  array_signal 'RANGE

TYPE my_type IS ARRAY range OF base_type;
SUBTYPE my_subtype IS unconstrained_array_type( range ) ;

SIGNAL | VARIABLE | CONSTANT name : unconstrained_array_type( range );

my_array( range )   -- array slice on LHS or RHS
my_array( index )   -- array element on LHS or RHS
( val1, value2, value3) -- array value using element values specified via position on RHS
( index1=>val1, index2=>val2, ...., OTHERS=>valn ) -- array value on RHS

array'LEFT          array'LOW
array'RIGHT         array'HIGH
array'LENGTH
array'RANGE (see definitions of range above)
```

VHDL Declarations

Declarations

SIGNAL *sname* : *stype* [:= *init_val*];
CONSTANT *cname*: *ctype* := *init_val*;
VARIABLE vname: vtype [:= init_val];
SHARED VARIABLE vname: vtype [:= init_val];
FILE *fname*: *ftype* [OPEN *file_open_kind* IS *file_name_string*];
TYPE *tname* : *tspec*;

Types

INTEGER 1, 123, -3456
NATURAL <non-negative integer>
REAL 1.21, -0.033
CHARACTER '*', '0',
STRING "my string"
BOOLEAN FALSE, TRUE
TIME 10.1 ns, 11 fs, 10 min (units fs, ps, ns, us, ms, s, min, hr) – physical time constant
INTEGER RANGE *low* TO *high* – fixed range integer type
TYPE *enumeration_type* IS (*value_name-1*, *value_name-2*, ..., *value_name-n*); –enumeration type

VHDL Operators

Logical (not is unary)	**and, or, nand, nor, xor, xnor, not** S X S -> B, B X B -> B (unary S->S, B->B)
Relational	=,/= (any type) <,<=,>,<= (scalar or discrete types)
Shift	**sll,srl,sla,sra,rol,ror** V X I -> V Shift Left/Right Logical/Arithmetic Rotate Left/Right
Addition	+, - N->N, N X N -> N (all same type) Also in *numeric_std* for V X V -> V etc
Multiply	*,/ N X N->N (all same type) **mod, rem** Integer
Misc **: exponentiation **abs**: absolute value	** N X N->N (all same type) a ** b = a^b **abs** N->N (both same type)
Concatenation	& V X V -> V, V X S -> V, S X V -> V

Key to Types

V: std_logic_vector
N: integer or real
I: integer
S: std_logic
B: Boolean

Vector lengths a,b:
length result
a + b -> max(a,b)
a * b -> a+b
a mod b -> b

Logical, Relational, Shift, Additive, Concatenation ops are synthesisable on *vectors* and *fixed range integers*

VHDL Text File I/O

TYPE *file_type* IS FILE OF *element-type*;

FILE *file_object*: *file_type* OPEN *file_open_kind* IS *file_name_string*;
– "Automatic" file open & close in object declaration

–TEXT file access uses Package STD.TEXTIO

–Use statement required (but no library statement, since STD is always available)

–Defines TEXT file type, LINE linebuffer type

TYPE SIDE IS (right, left);

TYPE FILE_OPEN_KIND IS (read_mode, write_mode, append_mode);

TYPE FILE_OPEN_STATUS IS (open_ok, status_error, name_error, mode_error);

file_open(VARIABLE status: OUT FILE_OPEN_STATUS;
FILE *f*: TEXT: name: IN STRING;
mode: IN FILE_OPEN_KIND := read_mode);

file_close(FILE f: TEXT);

endfile(FILE f: TEXT) RETURN boolean;

readline(FILE *f*: TEXT; *line*: OUT LINE);
– read the next line from f into line

writeline(FILE *f*: TEXT; *line*: INOUT: LINE);
– write a line from line to f, clearing line

read(*line*: INOUT LINE; *value*: OUT <*any-type*>);

write(*line*: INOUT LINE;
value: IN <*any-type*>;
justified: IN SIDE := right;
field: IN WIDTH := 0);

<*any-type*> ::= BIT, BIT_VECTOR, CHARACTER, INTEGER, REAL, TIME, STRING

Functions & Procedures

Meta-language
[] *optional part*
* *0 or more repetitions*
{} *grouping brackets*

```
PACKAGE mypack IS
   <function header>;
END PACKAGE mypack;
```

<function header> ::=
[IMPURE] FUNCTION *myfunc*[(<par> { ; <par> }*)]
RETURN *rtype*;
<par> ::= [<pspec>] *pname* : [<fmode>] *ptype* ;
<pspec> ::= FILE | SIGNAL | VARIABLE | CONSTANT
– omit <pspec> for value IN parameter
– may omit <pspec> for VARIABLE OUT mode
<fmode ::= IN | OUT | INOUT – default mode is IN
– functions can only have IN parameters

```
PACKAGE BODY mypack IS
   <function_header> IS
      <function-declarations>
   BEGIN
      <function-statements>
   END FUNCTION;
END PACKAGE BODY mypack;
```

<function-statement>::= <sequential_statement> |
RETURN *expression*;

- **PROCEDURE as for FUNCTION but**
  - **No RETURN *rtype* in header**
  - **WAIT allowed in body**
  - **OUT, INOUT parameters are allowed**
- **FUNCTIONs & PROCEDUREs can be defined in package body without duplicating header in package - in this case their scope is restricted to package body.**
- **Functions with zero parameters have no brackets in header or function call.**

Built-in functions

now -- returns current simulation time
'POS -- CHARACTER -> ASCII code
'VAL -- ASCII code -> CHARACTER

<scalar_type_name>'IMAGE(<value>)
-- <value> -> printable string

Master
E3.06
AC5
Ex 3.5

Answers

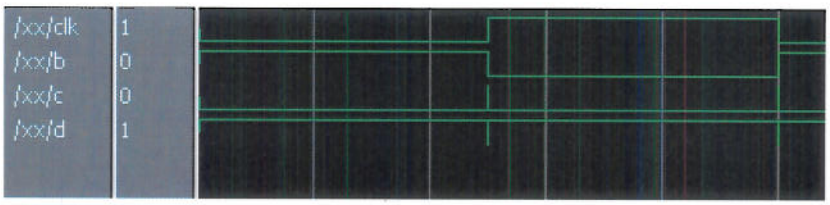
Question 1.

a)

```
SIGNAL p,q,r: UNSIGNED(9 DOWNTO 0);

P1: PROCESS
 BEGIN
   WAIT UNTIL clk'EVENT and clk='1';
   IF reset = '1' or r = q THEN
     r <= p;
   ELSE
     r <= r + 1;
   END IF;
 END PROCESS P1;
```

b) 2 marks for each waveform+deltas. Consequent mistakes are allowed



clk changes delta 1

b changes delta 2

c changes delta 2,3

d changes delta 3,4

c)

(y is 4 bit unsigned)

```
P2: PROCESS(x)
  VARIABLE v: INTEGER;
BEGIN
   v := 0;
   FOR i IN x'range LOOP
     IF x(i)='1' THEN v := v+1; END IF;
   END LOOP;
   y <= to_unsigned(v,4);
END PROCESS P2;
```

d) One correct: 3 marks, two correct: 6 marks, 3 correct: 8 marks.

Any 3 of:

(i) All input signals must be in sensitivity list
(ii) Driven outputs must be driven on all paths through process body
(iii) No cycle is allowed between output and input signals.
(iv) All variables must be written before they are read.
(v) Lengths must match across assignments
(vi) FOR LOOP range must be constant width

Half marks for:

(vii) Outputs driven by at most one process (50%)
(viii) No WAIT FOR statements

e) (1 mark each type)

```
SIGNAL a: SIGNED(4 DOWNTO 0);
SIGNAL b: UNSIGNED(1 DOWNTO 0);
SIGNAL c: UNSIGNED(4 DOWNTO 0);
SIGNAL q: SIGNED(7 DOWNTO 0);
```

```
P3: PROCESS(a,b,c) -- (4 marks)
BEGIN
  q <= a*signed("0" & b)+signed("0" & c);
END PROCESS P3;
```

- Could use signed types for all variables in which case b,c must be one bit longer & the initial appended zeros are not needed.
- Could use resize instead of appended zeros.

Question 2.

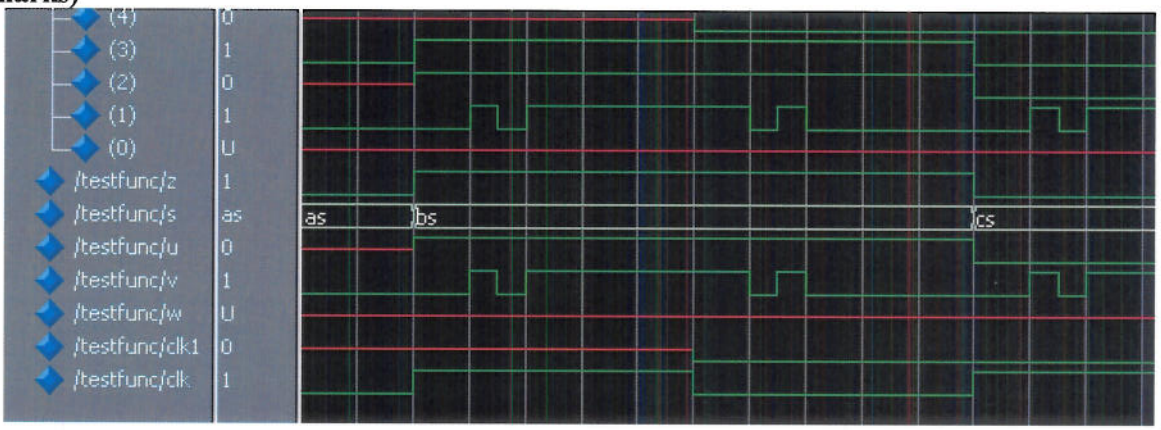
This question tests understanding of the relationship between pre-synthesis simulation and synthesis.

a) (5 marks)

P1: clk1,u,w,z,,s are all registered signals
clk1 - changed every cycle
u - changed every cycle
w - never initialised
s - initialised implicitly to first value in enumeration type at start of simulation
z - initialised in signal declaration

P0: v is a combinational signal
[also y[0] to y[4] ports, driven from dataflow statement, are combinational]

b) (10 marks)



c) (10 marls, 2 each) & d) (5 marks, 1 each)

y(0) - **w** - undefined in pre-synthesis simulation. Will probably (but not necessarily) be undefined in post-synth simulation also.

y(1) - **v** - the last transition is combinational equal to not clock. Various earlier transitions, preserved because of the TRANSPORT keyword, result in the shown waveform (note that the first driven value results in no transition, hence there are only three transitions per clk edge. Post-synthesis only the last transition will be used and so the two waveforms will differ around each edge of clk. This cannot simply be mended since AFTER keyword has no hardware equivalent.

y(2) - **u** - set from s - since s if undefined psot-synthesis this will be undefined. This can be mended by explicitly initialising s from a reset signal.

y(3) - **z** - defined in simulation due to initialisation of signal - will be undefined in post-synthesis simulation. This can be mended by explictly initialising z in a reset signal.

y(4) - **clk1** - pre-synthesis is delayed by 10ns - this is ignored by synthesis which is therefore same waveform but 10ns earlier. This cannot be mended since after has no hardware equivalent.

Question 3.

The code written here (can be) short, but both parts of the question involve quite difficult conceptual issues. Part a) can be simplified conceptually with an unrolled implementation: this will be allowed.

a) (15 marks)

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;

ARCHITECTURE xx OF priority_encode8 IS
BEGIN -- many other less optimal implementations possible
    PROCESS(x,en)
    BEGIN
        po <= pi;
        y <= (OTHERS=>'Z');
        FOR i in 0 To 7 LOOP
           IF x(i)='1' THEN
              y <= std_logic_vector(to_unsigned(i,3));
              po <= '1';
           END IF;
        END LOOP;
    END PROCESS;
END ARCHITECTURE xx;
```

b) (5 marks) E1,E0 each handle 8 bits, with the pi/po connection meaning that E0 is disabled if E1 finds any bit 1, in which case E1 provides the correct lower bits prioity encoded output. If E1 has all bits 0 it will be disabled, and E0 will provide the correct output. EMS takes the po outputs from E0,E1, which indicate whether or not any of the corresponding bits are high, and use these to generate the correct MS bit pattern. Note that if all bits are 0 the LS 3 bits will be tri-state, but this condition is not allowed.

c) (10 marks)

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;
USE WORK.ALL;

ARCHITECTURE xx OF big_priority_encode IS
    SIGNAL pp: std_logic; --dummy signal for unused output
    SIGNAL p: std_logic_vector(8 DOWNTO 0);
BEGIN
    MSBITS: ENTITY priority_encode8
       PORT MAP(pi=>'0',po=>pp,x=>p(7 DOWNTO 0),y=>y(5 DOWNTO 3));

    G1: FOR i in n TO 8 GENERATE
           p(i) <= '0'; --set undriven carry signals to 0
        END GENERATE;

    G: FOR i in 0 TO n-1 GENERATE
       LSBITS: ENTITY priority_encode8
          PORT MAP(pi=>p(i+1),po=>p(i),x=>x(i*8+7 DOWNTO i*8),
                   y=>y(2 DOWNTO 0));
       END GENERATE;
END ARCHITECTURE xx;
```

Question 4.

This question is quite long but straightforward RTL design.

a)

ra = 0, pa=qa = 0 (could be anything), n = 0, func = 4 or 7 **(2 marks)**

```
ENTITY datapath IS -- (4 marks)
PORT(
   clk: IN std_logic;
   func: IN std_logic_vector(2 DOWNTO 0);
   n: IN std_logic_vector(4 DOWNTO 0);
   paddr,qaddr,raddr: IN std_logic_vector(3 DOWNTO 0);
   f: OUT std_logic_vector(31 DOWNTO 0) );
END ENTITY datapath;
```

b)

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;

ARCHITECTURE xx OF datapath IS -- (6 marks)
   SUBTYPE word IS std_logic_vector(31 DOWNTO 0);
   TYPE ramtype IS ARRAY (0 TO 15) OF word;
   SIGNAL ram: ramtype;
   SIGNAL pbus, qbus, ybus, fbus: word;

BEGIN

f <= fbus;

-- RAM definition  (6 marks)
RAMW: PROCESS
BEGIN
    WAIT UNTIL clk'EVENT and clk = '1';
    ram(to_integer(unsigned(raddr))) <= fbus;
END PROCESS RAMW;

RAMR: PROCESS(paddr,qaddr,ram)
BEGIN
    pbus <= ram(to_integer(unsigned(paddr)));
    qbus <= ram(to_integer(unsigned(qaddr)));
END PROCESS RAMR;

-- Rotate unit definition (6 marks)
ROTATE: PROCESS(n,qbus)
BEGIN
    FOR i in word'RANGE LOOP
       ybus(i) <= qbus((i+to_integer(unsigned(n))) MOD 32);
    -- or qbus ror unsigned(n)
    END LOOP;
END PROCESS ROTATE;

--ALU definition (6 marks)
ALU: PROCESS(pbus,ybus,func)
VARIABLE up, uy: unsigned(31 DOWNTO 0);
BEGIN
    up := unsigned(pbus); uy := unsigned(ybus);
    CASE to_integer(unsigned(func)) IS
        WHEN 0 => fbus <= ybus;
        WHEN 1 => fbus <= std_logic_vector(up+uy);
        WHEN 2 => fbus <= pbus or ybus;
        WHEN 3 => fbus <= pbus and ybus;
        WHEN 4 => fbus <= std_logic_vector(up-uy);
        WHEN 5 => fbus <= not ybus;
        WHEN 6 => fbus <= pbus or not ybus;
        WHEN 7 => fbus <= std_logic_vector((up * uy)(31 DOWNTO 0));
        WHEN OTHERS => NULL;
    END CASE;
END PROCESS ALU;
END ARCHITECTURE xx;
```