Synthesis of Digital Architectures
Coursework Assignment

george.constantinides@ieee.org

This document, and all files related to it, can be downloaded from the course website http://cas.ee.ic.ac.uk/~gac1/Synthesis

1. Revision history

This document was created 27/01/03 (gac1); revised 16/02/04, 07/02/05, 31/01/06, 07/11/07 (gac1); 22/10/08 (gac1); 10/11/09 (gac1); 22/11/10 (gac1).

2. Introduction

For your coursework, you need to be divided into groups of approx. three students. To arrange this, please email george.constantinides@ieee.org with your name and course (i.e. EEE, ISE, MSc). If you already have already arranged a group yourselves, please indicate this and let me know the other members of the group.

You will undertake one of the following practical "mini-projects". Under exceptional circumstances, it is possible for you to propose an alternative mini-project, but it must be approved before you start work.

The deadline for the project is **5pm, Friday 14 January 2011**. You should hand in a report, detailing your approach to the problem and explaining your solution/code, by email to gac1@ic.ac.uk, attaching a single zip file with a copy of your code and a working implementation that should run on the departmental lab machines. No extensions will be given except under exceptional circumstances beyond your control.

Mini Project List

1. Write a program that takes a scheduled, bound non-hierarchical CDFG and generates a datapath in VHDL.
2. Write a program that takes a scheduled, bound non-hierarchical CDFG and generates an FSM controller in VHDL.
3. Write a program that takes a scheduled, bound non-hierarchical CDFG and generates a micro-sequencer in VHDL.
4. Write a program that performs ASAP, ALAP, and list-scheduling on a non-hierarchical CDFG, and compare design-space exploration results for CDFG benchmarks of your own creation.
5. Write a program that takes a CDFG and writes-out and solves optimum scheduling in a linear-program format (using lp_solve or CPLEX for the actual solution).
6. Write a program that takes a scheduled CDFG and performs a resource sharing (non-hierarchical and conditional-hierarchical at least).
7. Write a program to write-out and solve the retiming formulation (using lp_solve or similar for the actual solution).
8. Write a program that reads in a textual description of a CDFG containing function calls such as sin(.) and cos(.) and replaces these calls with an appropriate polynomial evaluations, given user-supplier error criteria.

9. Write a program that reads in a netlist description of a floorplanning problem and uses a simulated annealing or ILP-based solution to write out coordinates for a floorplan.

Some of these projects can be conducted within a well-defined framework, which is provided for you. If you are familiar with C++ or any other object-oriented programming language, then use of this framework will make your project easier. *However please feel free to add extra features to this framework, or ignore it completely, as required.*

To write your synthesis software, you are free to use any language you choose. For some projects, use of C++ is recommended, however, as you will benefit from the pre-written framework to be described below. For projects 1 – 3, your project will generate VHDL. For those of you unfamiliar with VHDL, do not worry as only a small subset of the language is required. Clear examples have been provided in the following text. For those of you familiar with Handel-C, it is also acceptable that Handel-C be used in place of VHDL.

The remainder of this document is split into two sections: a description of the VHDL examples, and a description of the C++ framework.

3. VHDL example files

This section of the document is only of relevance to those students attempting projects 1 – 3.

As an example for those of you unfamiliar with VHDL, and as a template for all of you, a complete implementation of the behaviour used as an example in Lecture 2 has been implemented.

As a reminder, this behaviour can be expressed as:

```
a = i+j;
t1 = 2*a;
b = t1+j;
c = a*b;
d = a*a;
```

You can download the following files from http://cas.ee.ic.ac.uk/~gac1/Synthesis

1. toplevel.vhd: an example description of the architecture used as an example in Lectures 2 and 4
2. datapath.vhd: the datapath of the above example
3. controller_microcode.vhd: a horizontal microcode controller implementation for this example
4. controller_FSM.vhd: a FSM controller implementation for this example
5. reg_enable.vhd: a simple register with enable input
6. adder.vhd: a simple single-cycle adder
7. mult.vhd: a simple two-cycle multiplier

Of these files, numbers 1-4 will vary from design to design, whereas numbers 5-7 will be the same for all designs. Each file is described below.

3.1 toplevel.vhd

This file implements the overall design. There are two 8-bit inputs: i and j, and two 8-bit outputs: c and d, as well as a clock input clk and a reset input reset. The toplevel design consists of two components: the datapath and the controller. toplevel.vhd is responsible for wiring-up these two components.

```
-- example VHDL toplevel file
-- implements behaviour used as an example in Lecture 2
-- and Lecture 4
-- George Constantinides, 27/1/03, george.constantinides@ieee.org

library IEEE;
use IEEE.std_logic_1164.all;

entity toplevel is
  port(
       clk : in std_logic;
       reset : in std_logic;
       i : in std_logic_vector(7 downto 0);
       j : in std_logic_vector(7 downto 0);
       c : out std_logic_vector(7 downto 0);
       d : out std_logic_vector(7 downto 0)
     );
end toplevel;

architecture struct of toplevel is

  component controller
    port(
          clk        : in std_logic;
          reset      : in std_logic;
          a_enable   : out std_logic;
          t1_enable  : out std_logic;
          d_enable   : out std_logic;
          b_enable   : out std_logic;
          c_enable   : out std_logic;
          adder1_sel : out std_logic;
          mult1_sel  : out std_logic
        );
  end component;

  component datapath
    port(
          clk        : in std_logic;
          i          : in std_logic_vector(7 downto 0);
          j          : in std_logic_vector(7 downto 0);
          a_enable   : in std_logic;
          b_enable   : in std_logic;
          t1_enable  : in std_logic;
          d_enable   : in std_logic;
          c_enable   : in std_logic;
          adder1_sel : in std_logic;
          mult1_sel  : in std_logic;
          c          : out std_logic_vector(7 downto 0);
          d          : out std_logic_vector(7 downto 0)
        );
  end component;

  signal a_enable   : std_logic;
  signal t1_enable  : std_logic;
  signal d_enable   : std_logic;
  signal b_enable   : std_logic;
  signal c_enable   : std_logic;
  signal adder1_sel : std_logic;
  signal mult1_sel  : std_logic;
```

```
begin

  cont_inst : controller port map( clk => clk, reset => reset, a_enable => a_enable,
                                   t1_enable => t1_enable, d_enable => d_enable,
                                   b_enable => b_enable, c_enable => c_enable,
                                   adder1_sel => adder1_sel, mult1_sel => mult1_sel );

  dp_inst : datapath port map( clk => clk, i => i, j => j, a_enable => a_enable,
  t1_enable => t1_enable, d_enable => d_enable, c_enable => c_enable, adder1_sel =>
  adder1_sel, mult1_sel => mult1_sel, c => c, d => d, b_enable => b_enable );

end struct;
```

3.2 datapath.vhd

datapath.vhd implements the datapath of the design. It consists of the usual components studied in the lectures: resources, multiplexers, and registers. In addition, there is a portion of the design to output those registers representing final results.

```
-- example VHDL datapath
-- implements behaviour used as an example in Lecture 2 (worked problem)
-- George Constantinides, 27/1/03, george.constantinides@ieee.org

library IEEE;
use IEEE.std_logic_1164.all;

entity datapath is
  port(
        clk        : in std_logic;
        i          : in std_logic_vector(7 downto 0);
        j          : in std_logic_vector(7 downto 0);
        a_enable   : in std_logic;
        b_enable   : in std_logic;
        t1_enable  : in std_logic;
        d_enable   : in std_logic;
        c_enable   : in std_logic;
        adder1_sel : in std_logic;
        mult1_sel  : in std_logic;
        c          : out std_logic_vector(7 downto 0);
        d          : out std_logic_vector(7 downto 0)
      );
end datapath;

architecture struct of datapath is

  component adder
    port( op1 : in std_logic_vector(7 downto 0);
          op2 : in std_logic_vector(7 downto 0);
          res : out std_logic_vector(7 downto 0)
        );
  end component;

  component mult
    port( clk : in std_logic;
          op1 : in std_logic_vector(7 downto 0);
          op2 : in std_logic_vector(7 downto 0);
          res : out std_logic_vector(7 downto 0)
        );
  end component;

  component reg_enable
    port( clk  : in std_logic;
          en   : in std_logic;
          data : in std_logic_vector(7 downto 0);
          res  : out std_logic_vector(7 downto 0)
        );
  end component;

  -- resources
  signal adder1_op1 : std_logic_vector(7 downto 0);
  signal adder1_op2 : std_logic_vector(7 downto 0);
  signal adder1_res : std_logic_vector(7 downto 0);
  signal mult1_op1  : std_logic_vector(7 downto 0);
  signal mult1_op2  : std_logic_vector(7 downto 0);
```

```
  signal mult1_res  : std_logic_vector(7 downto 0);
  signal mult2_op1  : std_logic_vector(7 downto 0);
  signal mult2_op2  : std_logic_vector(7 downto 0);
  signal mult2_res  : std_logic_vector(7 downto 0);

  -- register outputs
  signal areg : std_logic_vector(7 downto 0);
  signal breg : std_logic_vector(7 downto 0);
  signal creg : std_logic_vector(7 downto 0);
  signal dreg : std_logic_vector(7 downto 0);
  signal t1reg : std_logic_vector(7 downto 0);

begin

  -- resources
  adder1 : adder port map( op1 => adder1_op1, op2 => adder1_op2, res => adder1_res );
  mult1  : mult port map( clk => clk, op1 => mult1_op1, op2 => mult1_op2, res =>
mult1_res );
  mult2  : mult port map( clk => clk, op1 => mult2_op1, op2 => mult2_op2, res =>
mult2_res );

  -- registers
  ra  : reg_enable port map( clk => clk, en => a_enable, data => adder1_res, res =>
areg );
  rb  : reg_enable port map( clk => clk, en => b_enable, data => adder1_res, res =>
breg );
  rc  : reg_enable port map( clk => clk, en => c_enable, data => mult1_res, res =>
creg );
  rd  : reg_enable port map( clk => clk, en => d_enable, data => mult2_res, res =>
dreg );
  rt1 : reg_enable port map( clk => clk, en => t1_enable, data => mult1_res, res =>
t1reg );

  -- multiplexers
  adder1_op1 <= j;
  adder1_op2 <= i when adder1_sel='0' else t1reg;
  mult1_op1  <= areg;
  mult1_op2  <= "00000010" when mult1_sel='0' else breg;
  mult2_op1  <= areg;
  mult2_op2  <= areg;

  -- datapath outputs
  c <= creg;
  d <= dreg;

end struct;
```

3.3 controller_microcode.vhd

This file is a horizontal microcode implementation of the controller for the design. The design consists of two portions, the counter (whose current value is available as the signal “count”) and the ROM, which simply converts count to an appropriate output value. The individual bits of the output value are sent to the right locations by the final section of code.

```
-- example VHDL controller
-- implements behaviour used as an example in Lecture 4 (horizontal microcode)
-- George Constantinides, 27/1/03, george.constantinides@ieee.org

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity controller is
  port(
        clk        : in std_logic;
        reset      : in std_logic;
        a_enable   : out std_logic;
        t1_enable  : out std_logic;
        d_enable   : out std_logic;
        b_enable   : out std_logic;
        c_enable   : out std_logic;
```

```
        adder1_sel : out std_logic;
        mult1_sel  : out std_logic
      );
end controller;

architecture microcode of controller is

signal count : std_logic_vector(2 downto 0);
signal rom_data : std_logic_vector(6 downto 0);

begin

  counter : process( clk )
  begin
    if reset = '1' then
      count <= (others => '0');
    elsif clk'event and clk='1' then
      count <= count+1;
    end if;
  end process;

  rom : process( count )
  begin
    case count is
      when "000" =>
        rom_data <= "1000000";
      when "001" =>
        rom_data <= "0000000";
      when "010" =>
        rom_data <= "0110000";
      when "011" =>
        rom_data <= "0001010";
      when "100" =>
        rom_data <= "0000001";
      when "101" =>
        rom_data <= "0000100";
      when others =>
        rom_data <= (others => '0');
    end case;
  end process;

  a_enable   <= rom_data(6);
  t1_enable  <= rom_data(5);
  d_enable   <= rom_data(4);
  b_enable   <= rom_data(3);
  c_enable   <= rom_data(2);
  adder1_sel <= rom_data(1);
  mult1_sel  <= rom_data(0);

end microcode;
```

3.4 controller_FSM.vhd

This design file implements an alternative version of the controller (using a different architecture name), as a finite state machine. Only one of {controller_microcode.vhd, controller_FSM.vhd} should be compiled. The design consists of two components, some next-state and data-output combinational logic and a process defining the state register.

```
-- example VHDL controller
-- implements behaviour used as an example in Lecture 4 (FSM)
-- George Constantinides, 27/1/03, george.constantinides@ieee.org

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity controller is
  port(
        clk        : in std_logic;
        reset      : in std_logic;
```

```
        a_enable   : out std_logic;
        t1_enable  : out std_logic;
        d_enable   : out std_logic;
        b_enable   : out std_logic;
        c_enable   : out std_logic;
        adder1_sel : out std_logic;
        mult1_sel  : out std_logic
      );
end controller;

architecture FSM of controller is

type state is (cycle1, cycle2, cycle3, cycle4, cycle5, cycle6);
signal present_state, next_state : state;

signal data_out : std_logic_vector(6 downto 0);

begin

  logic : process( present_state )
  begin
    case present_state is
      when cycle1 =>
        next_state <= cycle2;
        data_out   <= "1000000";
      when cycle2 =>
        next_state <= cycle3;
        data_out   <= "0000000";
      when cycle3 =>
        next_state <= cycle4;
        data_out   <= "0110000";
      when cycle4 =>
        next_state <= cycle5;
        data_out   <= "0001010";
      when cycle5 =>
        next_state <= cycle6;
        data_out   <= "0000001";
      when cycle6 =>
        next_state <= cycle1;
        data_out   <= "0000100";
    end case;
  end process;

  state_transition : process( clk )
  begin
    if reset = '1' then
      present_state <= cycle1;
    elsif clk'event and clk='1' then
      present_state <= next_state;
    end if;
  end process;


  a_enable   <= data_out(6);
  t1_enable  <= data_out(5);
  d_enable   <= data_out(4);
  b_enable   <= data_out(3);
  c_enable   <= data_out(2);
  adder1_sel <= data_out(1);
  mult1_sel  <= data_out(0);

end FSM;
```

3.5 reg_enable.vhd

This is just a simple implementation of a register with an enable signal, as used by datapath.vhd.

```
-- A simple register with enable
-- George Constantinides, 27/1/03, george.constantinides@ieee.org

library IEEE;
use IEEE.std_logic_1164.all;

entity reg_enable is
```

```
  port( clk  : in std_logic;
        en   : in std_logic;
        data : in std_logic_vector(7 downto 0);
        res  : out std_logic_vector(7 downto 0)
      );
end reg_enable;

architecture rtl of reg_enable is
begin

  process( clk )
  begin
    if clk'event and clk='1' then
      if en='1' then
        res <= data;
      end if;
    end if;
  end process;

end rtl;
```

3.6 adder.vhd

This is a simple implementation of a combinational 8-bit adder.

```
-- A simple unsigned adder - completes in the same clock cycle
-- George Constantinides, 27/1/03, george.constantinides@ieee.org

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity adder is
  port(
        op1 : in std_logic_vector(7 downto 0);
        op2 : in std_logic_vector(7 downto 0);
        res : out std_logic_vector(7 downto 0)
      );
end adder;

architecture rtl of adder is
begin

  res <= op1 + op2;

end rtl;
```

3.7 mult.vhd

This is a behavioural model of a two-cycle multiplier. You would not actually use a file like this for synthesising your final circuit, as we achieve here the behaviour of a multiplier taking two cycles by using a combinational multiplier and delaying the result by one cycle! However it is entirely appropriate as a model to use for simulation purposes.

Note that it is the least significant 8 bits of the 16-bit multiplier result which form the product of this 8x8-bit multiplier.

```
-- A simple unsigned multiplier - takes two cycles to complete

-- Note that this is really a hack to provide an example for the students: in reality
-- the multiplication takes one cycle, but we delay it for a cycle to provide the
-- illusion of a two cycle multiplier. For a real implementation, this design would
-- be replaced by a technology-specific core.

-- George Constantinides, 27/1/03, george.constantinides@ieee.org

library IEEE;
use IEEE.std_logic_1164.all;
```

```
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity mult is
  port(
        clk : in std_logic;
        op1 : in std_logic_vector(7 downto 0);
        op2 : in std_logic_vector(7 downto 0);
        res : out std_logic_vector(7 downto 0)
      );
end mult;

architecture rtl of mult is

  signal res_int : std_logic_vector(15 downto 0);

begin

  res_int <= op1 * op2;

  process( clk )
  begin
    if clk'event and clk='1' then
      res <= res_int(7 downto 0);
    end if;
  end process;

end rtl;
```

3.8 Testing it: Performing a simulation

You can try out these files on any machine with ModelSim (a VHDL simulator) installed. ModelSim can be run either from the command-line or from the GUI. From the command line you must perform the following (I am assuming your path has been set up correctly)

1. Create a library called “work” which will hold your compiled design:

```
vlib work
```

2. Compile each VHDL file for ModelSim (only one of the two different versions of controller).

```
vcom mult.vhd
vcom adder.vhd
vcom reg_enable.vhd
vcom datapath.vhd
vcom controller_FSM.vhd
vcom toplevel.vhd
```

3. Simulate the compiled top-level design:

```
vsim toplevel
```

vsim starts the GUI, from which you may explore your design. Click View->Signals and View->Wave. Drag and Drop each of the signals from the “signals” window to the “wave” window.

Since our testing will be very simple, we can do it entirely within the ModelSim environment.

1. Set up “clk” as a clock signal: highlight clk in the signals window, select Edit->Clock, accept the defaults and click “OK”.

2. Give values “2” and “3” to “i” and “j” respectively: highlight “i” in the signals window, select Edit->Force. In the value field type “00000010” and select OK. Highlight “j” in the signals window, select Edit->Force. In the value field type “00000011” and select OK.
3. Create a reset signal which is high for a while, and then goes low: highlight “reset” in the signals window, select Edit->Force. In the value field type “1” and in the “Cancel after” field type “100”. Select OK. Again highlight “reset” and select Edit->Force. In the value field type “0” and in the “Delay for” field type “100”. Select OK. The joint effect of these is to make the reset signal high for 100ns and then low for the remainder of the time.
4. In the main ModelSim command window type “run 700ns” to simulate for 700ns.

In the wave window you will see how the signals have changed over time. At the end of the 700ns, all the signals have the correct values: a = 5, t1 = 10, b = 13, c = 65, d = 25.

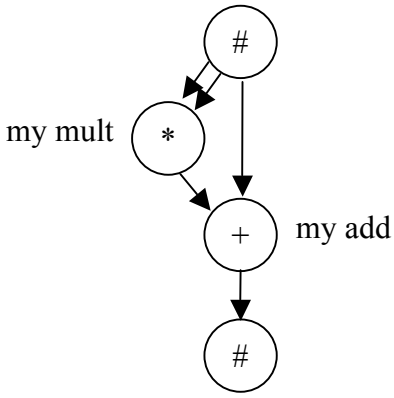
4. C++ class library

This portion of the document is of relevance to most students – even if you don't wish to use C++, the structure of your code will undoubtedly be similar.

The C++ class library consists of several files that allow us to construct and manipulate control/data-flow graphs (CDFGs). These can be downloaded from http://infoeng.ee.ic.ac.uk/~gac1/Synthesis. Each class has its own source file (.cpp) and its own header file (.hpp). The header file should contain all the information you need on how to use the class. We will look "under the hood" at each of the header files in turn, after first examining an example usage (test_cdfg.cpp).

4.0 test_dfg.cpp

This is a very simple test program. The idea is to create the CDFG shown below, hold it in an internal representation, and create a VCG file so we can plot the graph if we so choose. Note that we have relaxed the definition of "graph" (actually "digraph") somewhat, to allow two edges between the same pair of vertices in the same direction. This makes things easier for us, and doesn't cause a problem for scheduling, binding, etc.




```
////////////////////////////////////////
// test_cdfg.cpp
// Demo the CDFG routines
// (for use in 4th year / MSc course)
//
// George Constantinides, November 2002
////////////////////////////////////////

#include "CDFG.hpp"
#include "mult.hpp"
#include "add.hpp"
#include <fstream>

int main()
{
  CDFG my_cdfg;

  //
  // Create the nodes in a simple CDFG
  //

  mult *m = new mult("my mult");
  my_cdfg.add_node( m );
  add  *a = new add("my add");
  my_cdfg.add_node( a );
```

```
  //
  // Create the edges
  //

  // source node
  my_cdfg.source_node()->connect_output( m );

  // node "m"
  m->connect_inputs( my_cdfg.source_node(), my_cdfg.source_node() );
  m->connect_output( a );

  // node "a"
  a->connect_inputs( m, my_cdfg.source_node() );
  a->connect_output( my_cdfg.sink_node() );

  // sink node
  my_cdfg.sink_node()->connect_input( a );

  //
  // Now create a VCG file for visualization
  //

  ofstream vcgfile("test_cdfg.vcg");
  my_cdfg.write_vcg( vcgfile );

  return 0;
}
```

4.1 class cdfg_node

cdfg_node is an abstract class, representing a node in a CDFG. No actual CDFG node will have type cdfg_node, but all real CDFG nodes are inherited from this class. There are two methods of interest defined so far:

4.1.1 cdfg_node(const string&)

This is the constructor – it creates a node, and you pass the name of the node to create.

4.1.2 virtual void write_vcg(ostream&)

This is a placeholder for a function to write a description of a CDFG in a form called "VCG" readable by a program which draws graphs. This program, also called "VCG", is available from http://rw4.cs.uni-sb.de/users/sander/html/gsvcg1.html and I believe a free port for those who use "the other operating system" (i.e. MS Windows) is available from http://www.absint.com/aisee.

4.2 class nop

The nop class represents "no operation" or dummy nodes, used in CDFGs to mark the "start" and "end" nodes. It is derived from class cdfg_node.

4.2.1 nop(const string&)

Constructor – creates a node, and you pass the name of the node to create.

4.2.2 virtual void write_vcg(ostream&)

The VCG visualization output routine for NOP nodes. These are visualized as a node containing the "#" symbol (as in the lecture notes), together with the node name.

4.2.3 void connect_input(cdfg_node *)

Connects an edge from the passed CDFG node pointer to this node.

4.2.4 void connect_output(cdfg_node *)

Connects an edge from this node to the passed CDFG node pointer.

4.3 class binop

The binop class is used to represent binary operations, i.e. those taking in two input values. It is derived from class cdfg_node.

4.3.3 void connect_inputs(cdfg_node *, cdfg_node *)

Connects an edge from the two passed CDFG node pointers to this node.

4.3.4 void connect_output(cdfg_node *)

Connects an edge from this node to the passed CDFG node pointer.

4.4 class add

The add class is used to represent additions. It is derived from class binop.

4.5 class mult

The mult class is used to represent multiplications. It is derived from class binop.

4.6 class CDFG

This is the "main" class, used to represent a CDFG. It has the following pre-defined methods of interest. Other method names have been commented out – these correspond to your work in the project.

4.6.1 void add_node(cdfg_node *)

Add the passed CDFG node to the graph

4.6.2 void write_vcg(ostream&)

Write the entire CDFG to a VCG file for visualization

4.6.3 nop *source_node(), nop *sink_node()

Returns the source (start) node of this CDFG, returns the sink (end) node of this CDFG, respectively.