ECE-327: Digital Systems Engineering
Lab Manual

2018t2 (Spring)

Instructor: Nachiket Kapre
Lab Manual: Mark Aagaard

University of Waterloo
Department of Electrical and Computer Engineering

Table of Contents

The CAD tools for ece327 can be run on either the ECE Linux computers or the ECE Nexus computers. Section 3 describes how to setup your Linux account to run the tools. Section 5 is for Nexus.

1   Connecting to an ECE-Linux Computer

To access the ECE-Linux computers, we need to login from another computer and create an X-windows connection. You may login from any computer (ECE Nexus, other Nexus, home, etc). We describe three options:

Section 1.1   MobaXterm From Nexus
Section 1.2   From Linux off campus
Section 1.3   From Mac OS

1.1   Connection from MS-Windows: MobaXterm

On Nexus, MobaXterm is at: `Q:\eng\ece\util\mobaxterm.exe`.

On your personal computer, you can install it from: `http://mobaxterm.mobatek.net`.

To connext to an X-windows server, just:
1. run `mobaxterm.exe`
2. `ssh -X ecelinux.uwaterloo.ca`

1.2   Connection from Linux

For text-based interaction, you can just `ssh` into an ecelinux computer. For a graphical connection, use `ssh -X`.

1.3   Connection from Mac OS

Mac OS requires "XQuartz" in order to run X11 applications. Once XQuarts is installed, you can just "ssh -X ecelinux.uwaterloo.ca".

2   Editors and Shells

There are several editors available on either Linux or Nexus:

Linux	**Nexus**
• nedit (a very simple editor)	• emacs
• pico	• gvim
• emacs	• notepad++
• vi, vim, gvim	• notepad2
• xedit	

Emacs, vim, and gvim are the most powerful, and most complicated. Both emacs and vim/gvim have VHDL modes with syntax highlighting. For the lazy, emacs even has autocompletion for VHDL. On Nexus, notepad++ has a nice VHDL mode.

On Linux, if you are unfamiliar with emacs and vim, try starting with nedit for a simple, intuitive editor.

3   ECE-Linux Configuration

The simplest method to configure your ECE-Linux account is to run the ece327 beginning-of-term script:

```
/home/ece327/bin/ece327-bot
```

This will save a copy of your startup file (`.profile` for Bash and `.cshrc` for csh/tcsh), then create a one-line startup file that sources the ece327 setup script.

To reverse the effects of the beginning-of-term script, run:

```
/home/ece327/bin/ece327-bot-undo
```

- **You must now logout and then login again for the changes to take effect.**
- **Proceed to Lab 1.**

4   ECE-Linux Manual Configuration

This section describes the manual proces for configuring your ECE-Linux account. **Do *not* do this section if you did the simple configuration process in section 3.**

4.1   Finding Your Shell

After you have logged into an ECE-Linux computer, and before you can configure your environment, you need to find out which shell you are using. As your shell, you will be using csh, tcsh, or bash. To find out which shell you are using, type:

```
% echo $SHELL
```

If the output of the above command ends in `bash`, then follow the instructions in Section 4.2. Otherwise, the output should end in `csh` or `tcsh` and you should follow the instructions in Section 4.3.

If you wish to change your shell, go to:
https://eceweb.uwaterloo.ca/password

4.2   Bash Shell Configuration

1. If the following line is not already in your `.profile` file, then add the line to the *end* of the file:

   ```
   source /home/ece327/setup-ece327.sh
   ```

2. Leave your existing session connected, and start a new connection to ECE-Linux. You need to start a new connection to see the changes that you have made to your configuration.

3. Test the configuration:
   ```
   % which precision
       /opt-src/CMC/local/maagaard/bin/precision
   % which quartus_sh
       /opt-src/CMC/local/maagaard/bin/quartus_sh
   % which uw-clean
       /home/ece327/bin/uw-clean
   ```

4. If the above commands do not work, try putting `source /home/ece327/setup-ece327.sh` as the first line in your `.profile` file.

5. If you are unable to get your environment working correctly, then make a backup copy of your `.profile` and `.bashrc` files and install fresh ones as shown below:
```
% mv .bashrc bashrc.orig
% cp /home/ece327/setup/bashrc .bashrc
% mv .profile profile.orig
% ln -s .bashrc .profile
```

The above commands make a copy of `.bashrc` and then a symbolic link from `.bashrc` to `.profile`. If the fresh versions of these files work, then gradually add the code from your `profile.orig` and `bashrc.orig` files back into `.profile`

4.3 Csh and Tcsh Shell Configuration

1. If the following line is not already in your `.cshrc` file, then add the line to the *end* of the file:

`source /home/ece327/setup-ece327.csh`

2. Leave your existing session connected, and start a new connection to ECE-Linux. You need to start a new connection to see the changes that you have made to your configuration.

3. Test the configuration:
```
% which precision
      /opt-src/CMC/local/maagaard/bin/precision
% which quartus_sh
      /opt-src/CMC/local/maagaard/bin/quartus_sh
% which uw-clean
      /home/ece327/bin/uw-clean
```

4. If the above commands do not work, try putting `source /home/ece327/setup-ece327.csh` as the first line in your `.profile` file.

5. If you are unable to get your environment working correctly, then make a backup copy of your `.cshrc` and `.login` files and install fresh ones as shown below:
```
% mv .login login.orig
% cp /home/ece327/setup/login .login
% mv .cshrc cshrc.orig
% cp /home/ece327/setup/cshrc .cshrc
```

If the fresh versions of these files work, then gradually add the code from your `login.orig` and `cshrc.orig` files back into the fresh version.

5 Nexus Configuration

Follow these steps if you wish to run the ECE-327 tools directly on a Nexus computer.

1. Login to your Nexus account

2. Map the network location `\\eceserv\`*your_userid* to `P:`. This will allow you to switch easily between Nexus and Linux, because the `P:` volume on Nexus will be the same as your home directory on the ECE-Linux computers.

3. Environment Setup

   In each DOS window in which you will run the ece327 CAD tools, you need to run `327setup`:

   ```
   Q:\eng\ece\util\327setup
   ```

   The full name of the `327setup` script is: `Q:\eng\ece\util\327setup.bat`.

4. Use your favourite editor to edit the VHDL and .uwp files.

   Some of the available editors are:

   - emacs
   - gvim
   - notepad++
   - notepad2

   Emacs and gvim are the most powerful, and most complicated. Both emacs and gvim have VHDL modes with syntax highlighting. For the lazy, emacs even has autocompletion for VHDL. If you are unfamiliar with emacs and vim, notepad++ has a nice VHDL mode.

5. Proceed to Lab 1.

1 Scripts

To simplify the use of the tools used in the course, a variety of scripts have been written. All of the scripts are installed in `/home/ece327/bin` on ECE Linux and `Y:\bin` on Nexus . If you have completed the configuration steps described in Handout 1, you should be able to run the scripts.

The scripts are:

`uw-clean`	Deletes all intermediate files and directories generated by the tools.
`uw-com`	Performs syntax and type checking.
`uw-sim`	Simulation
`uw-synth`	Synthesis
`uw-dnld`	Download a `.sof` file to the FPGA board (**Nexus only**)
`uw-report`	Print out summary of area, speed, and power
`uw-add-pwr`	Add `vcd` log commands to a simulation script
`uw-loop`	Execute synthesis; timing simulation; power analysis

The main argument to `uw-com`, `uw-sim`, and `uw-synth` is the name of the file to compile, simulate, or synthesize. The file may be either a VHDL file or a UW Project file, which has a `.uwp` extension.

The scripts create several directoris for temporary files:

LOG Log files

RPT Area, timing, and power report files

uw_tmp Scripts and some generated temporary files

work-* Intermediate compiled or object versions of files

Synthesis flows:

FPGA synthesis Logic synthesis is done with Mentor Graphics PrecisionRTL. Physical synthesis (place and route) is done with Altera Quartus. PrecisionRTL writes a file `uw_tmp/`*name*`.edf` that contains the post-logic-synthesis design and a file `uw_tmp/`*name*`.tcl` with TCL commands for Quartus.

ASIC synthesis Logic synthesis is done with Synopsys Design Compiler. Physical synthesis is done with Synopsys IC Compiler or Cadence First Encounter.

Each simulation or synthesis script generate a Python and a TCL file in the `uw_tmp` directory that contains the actual commands that are run.

For simulation and synthesis, there are a variety of options that can be included.

Option	Sim	Synth	Description
`-h`	√	√	show a help message
`--gui`	√	√	start the GUI
`--nogui`	√	√	do not start the GUI
`--i`	√	√	Interactive: at end of running the script, enable interactive execution of commands (not available for FPGA synthesis)
`--generics=`*var=value,var=value,...* or `-G` *var=value,var=value,...*	√	√	assign a value to a generic parameter
`-p` or `--prog`	√		simulate the VHDL program
`-g` or `--gates`	√	√	synthesize to or simulate generic gates
`-l` or `--logic`	√	√	logic synthesis to and simultion of FPGA gates for the Cyclone-II chip on the DE2 board, which will be used for functional simulation and demos.
`-c` or `--chip`	√	√	Synthesis: logic+physical synthesis (place and route) to FPGA gates for the Cyclone-II chip chip on the DE2 board, which will be used for timing simulation and demos. Simulation: simulate the design after place and route without timing delays.
`-t` or `--timing`	√		simulate netlist (logic or chip) with timing info
`-o`		√	synthesize for optimality measurement
`--sim_script=`*SIM_SCRIPT*	√		simulation script (overrides SIM_SCRIPT in project file)

2   UW Project Files

A UW project file must end with a `.uwp` extension.

Entire lines may be commented out with `#`. Partially commented-out lines are not supported.

The syntax for a line of the file is:

variable = *value*	Assign *value* to *variable*
variable = *value*$_1$ *value*$_2$ `...`	Assign the list of *value*s to *variable*
Both value and list assignments may be used with:	
variable `+=` *value*	*Prepend* *value* to the beginning of the previously assigned value of *variable*
variable `=+` *value*	*Append* *value* to the end of the previously assigned value of *variable*

All of the values must appear on a single line.

The tags for source files are named `*_VHDL`, but they accept both VHDL and Verilog files.

The variables are:

`LIB_VHDL`	Files that are needed for both synthesis and simulation.
`DESIGN_ENTITY`	Name of the top-level design entity
`DESIGN_ARCH`	Archicture of the top-level design entity
`DESIGN_VHDL`	Space-delimited list of VHDL files containing the design.
`TB_ENTITY`	Name of the testbench entity
`TB_ARCH`	Archicture of the testbench
`TB_VHDL`	Space-delimited list of VHDL files containing the testbench.
`BOARD`	FPGA board or ASIC chip for implementation. Currently supported values are: DE2 (default) EXCAL old Altera Excalibur board ibm130nm 130nm ASIC library (research only) saed90nm 90nm ASCIC library (academic) cmos65nm 65nm ASIC library (research only)
`SIM_SCRIPT`	Simulation script to run
`PIN_FILE`	File containing mapping of top-level design entity signals to pins on the FPGA chip.

Example UWP file:

```
TB_ENTITY      = kirsch_tb
TB_ARCH        = main
TB_VHDL        = kirsch_synth_pkg.vhd string_pkg.vhd kirsch_unsynth_pkg.vhd kirsch_tb.vhd
SIM_SCRIPT     = kirsch_tb.sim

DESIGN_ENTITY  = kirsch
DESIGN_ARCH    = main
DESIGN_VHDL    = kirsch_synth_pkg.vhd my_units.vhd kirsch.vhd

BOARD          = DE2
# comment out old: BOARD = EXCAL
```

Many intermediate files are generated during the synthesis process. Look in `uw_tmp`, `LOG`, `RPT*`:

`*gate*`	generated from logic synthesis to generic gates
`*logic*`	generated from logic synthesis to FPGA or ASIC cells
`*chip*`	generated from physical (chip) synthesis
`*.sdf`	Back-annotated delay information
`*.vcd`	Value-change-dump file for power analysis

3   Locally Added Simulation Commands

`rerun`	rerun the current simulation
`reload`	reload (recompile) the source files
`rr`	reload the source files and rerun the simulation

Handout 3: Timing Simulation

1 Zero-Delay Circuit Simulation

The simulations you have been doing up until now are zero-delay in the sense that signals moving from one hardware component to another always arrived at their destinations at the same time. This simplification allows the computer to simulate your design quickly.

In zero-delay simulation, all signal paths are exactly the same length. You can imagine that signals are able to travel as fast or slow as necessary to reach their intended destinations at the exact same time. Regardless of how you interpret it, your design is implementation independent. While it can be simulated, the simulation does not take into account any of the particular characteristics of either the target implementation technology or your design in terms of placement and routing. However, this kind of simulation is what you should use for testing the logical correctness of your design. After all, there is no point in worrying about whether your design can be synthesized, if it does not work correctly in the first place. However, while you are working on achieving functional correctness, one should still observe proper VHDL coding practices to avoid constructs that cannot be synthesized into FPGA-centric hardware.

2 Where Timing Simulation Comes In

Once the point has been reached that your design is functionally correct, the next step in simulation fidelity is to accurately model the actual timing characteristics of the implementation technology you intend to use. For the purposes of this course, the technology will be an FPGA, but it need not be when you work in a specific industry. FPGA implementations of your design will have a very different layout and logical architecture than, say, custom VLSI, and will therefore have different timing characteristics as well. This will imply different performance limits because of the timing constraints that each implementation will allow, as well as the intelligence of the routing tools that place and route the signals and/or design elements. Its up to you as a good designer to make the judgement calls that balance cost and performance. Timing analysis is part of what this is all about.

3 What Timing Simulation Does

Timing simulation uses the layout and routing delay information (from a placed and routed design) to give a more accurate assessment of the behaviour of the circuit under worst-case conditions. The design which is loaded into the simulator for timing simulation contains worst-case routing delays based on the actual placed and routed design. For this reason, timing simulation is performed after the design has been placed and routed. Timing simulation is more time consuming than the idealized ones, especially for current-generation microprocessors consisting of millions of transistors. For example, 1 microsecond of real-time operation, can take as much as 15 minutes of simulation time! But this is what it takes to ensure that no glitches occur in the design and that all signals reach their destination on time.

4 Performing a Timing Simulation

In ECE-327, there are two steps to running a timing simulation: creating the files containing the timing information and execution of the simulation.

Use `uw-synth --logic` to synthesize your design.

Use `uw-sim --logic --timing` to do timing simulation with your synthesized design.

5 Debugging in Chip and Timing Simulation

If your design works in zero-delay simulation and you have followed the ece327 coding guidelines, then your design should work in timing simulation.

Debugging in chip and timing simulation is difficult, because the synthesis tools will have mapped all of your combinational logic signals to gates and will use automatically generated names for the resulting signals. Your registered signals will still be there with (almost) their original name, but arrays will be broken down into individual bits.

First, identify whether your design works for each the following:
- `uw-sim --prog`
- `uw-sim --logic`
- `uw-sim --logic --timing`
- on the FPGA board

Identify the first of the above situations where your design does not work correctly (e.g. works in `uw-sim --logic` but fails with `uw-sim --logic --timing`).

If your design works with `uw-sim --prog` but fails `uw-sim --logic`
1. check for warnings in synthesis log file (`LOG/uw-synth.log`).
2. ensure that your code follows the coding guidelines.

5.1 Bugs in Chip Simulation

Chip simulation is zero-delay simulation of the synthesized design. It uses the `_chip.vho` file for the design.

There are two common failure modes in `uw-sim --logic`:
- Xs on signals: common causes are:
- latches
- multiple drivers of a signal
- combinational loops
- forgot to reset an important flop
- no Xs, but different behaviour than in `uw-sim --prog`. common causes are:
  - incorrect interface to outside world (e.g. reading inputs in the wrong clock cycle, assuming that input values are stable for multiple clock cycles, modified testbench from original version.)
  - Beware of boolean conditions: the expression a = '0' evaluates to false if a is X. Thus, a value of X can be silently converted into a normal value of 0 or 1.

5.2   Bugs in Timing Simulation

If your design works with `uw-sim --logic` but fails `uw-sim --logic --timing`, the problem is probably that the clock is too fast (i.e. circuit is too slow). Try increasing the clock period in the test bench.

To debug with `uw-sim --logic` or `uw-sim --logic --timing` when Xs appear:

1. Pick a flop that should be reset and eventually gets Xs. Simulate and confirm that the flop is reset correctly (maybe testbench is not setting reset).
2. Trace backwards through the fanin of the flop to find the flop/input that causes X

To debug with `--logic` or `--logic --timing` when there aren't Xs, but you have different behaviour with `--prog`:

1. Trace flops in the buggy simulation compare with flops in `uw-sim --prog`.
2. Identify first clock cycle with different behaviours in the two simulations.
3. If you need to look at comb signals in `--logic` or `--logic --timing`, there's a difficulty because comb signals disappear during synth. A solution is to add debug signals to the entity for the design under test. Example:
   - In the entity:

     ```
     dbg_a, dbg_b, dbg_c : std_logic;
     dbg_vec_w, dbg_vec_x, dbg_vec_y : std_logic_vector(7 downto 0);
     ```

   - In the architecture

     ```
     dbg_a <= comb signal that you want to probe
     dbg_b <= ...
     ```

   - When the design works, comment out the debug signals in the entity and architecture

Handout 5: Warning and Error Messages

1 Warning Messages from PrecisionRTL (uw-synth)

- `Warning, signal is not always assigned.  Storage may be needed.`

  This is **bad**: *signal* will be turned into a latch. Check to make sure that you are covering all possible cases in your assignmemts, and use a fall-through "others" or "else".

- `Warning, signal should be declared on the sensitivity list of the process`

  If this is in a combinational process, then this is **bad**, because you will get a latch rather than combinational circuitry. If this is in a clocked process, then it is safe to ignore the warning.

- `Warning, Multiple drivers on signal; also line line`

  This is **bad**. The probable cause is that more than one process is writing to *signal*.

  Here's a sample situation with multiple drivers, and an explanation of how to fix the problem:

```
do_red:
  process (state) begin
    if ( state = RED ) then
      next_state <= ....
    end if;
  end process;

do_green:
  process (state) begin
    if ( state = GREEN ) then
      next_state <= ....
    end if;
  end process;
```

The goal here was to organize the design so that each process handles one situation, analogous to procedures in software. However, both `do_red` and `do_green` assign values to `next_state`. For hardware, decompose your design into processes based upon the signals that are assigned to, not based on different situations.

For example, with a traffic light system, have one process for the north-south light and one for the east-west light, rather than having one process for when north-south is green and another process for when east-west is green.

The correct way to think about the organization of your code is for each process to be responsible for a signal, or a set of closely related signals. This would then give:

```
  process (state) begin
    if ( state = GREEN ) then
      next_state <= ...
```

```
      elseif ( state = GREEN ) then
        next_state <= ...
      else ... [ other cases ] ...
      end if;
  end process;
```

2   Error Message

- `Component comp has no visible entity binding.`

  In your VHDL file, you have a component named *comp*, but you have not compiled an entity to put in for the component.

Handout 4: Debugging Latches and Combinational Loops

When doing synthesis, if you get the error message: "Design contains one or more combinational loops or latches.", first, look for any latches (Section 1). If you do not find any latches in your design, then look for combinational loops (Section 2).

1   Finding Latches

1. Run `uw-synth` with the GUI and synthesize for *generic gates* (not an FPGA): `uw-synth --gui` *filename*.

2. Open up the RTL schematic

3. On the left-side of the RTL schematic, expand **Instances** and look for any latches (the name will probably begin with "lat_" or you will see "LAT" after the name).

4. Click on any latch instances and the schematic window will zoom in on the latch.

5. The output signal from the latch will probably be an automatically generated name, so you will have to navigate around to figure out which signal in your VHDL corresponds to the latch output.

2   Finding Combinational Loops

A combinational loop can span multiple processes, and the combinational dependencies include conditionals, not just the signals on the right-hand-side of an assignment. For example, the following is a combinational loop through `a` and `d`:

```
process (a,b,c) begin
  if a = '0' then
    d <= b;
  else
    d <= c;
  end if;
end process;
process (d) begin
  a <= d;
end process;
```

There are several ways to find a combinational loop:

1. Turn combinational assignments into registed assignments until you find an assignment that, when combinational, results in a combinational loop and, when registered, removes the combinational loop.

2. Use a divide-and-conquer approach by deleting large chunks of code and trying to find a minimal subset of your program that contains the combinational loop.

3. Use the schematic browser in PrecisionRTL to visually hunt for the combinational loop. To do this, bring up the RTL schematic and:

   (a) Right click on the signal you want to include in the trace.

   (b) Select add to trace.

   (c) On the left-side of the screen, choose Schematics → View RTL Trace.

   (d) In the RTL Trace window, right click on a signal that you are interested in and choose Trace Backward for the number of levels you are interested in. One level at a time is usually best, because it prevents crowding your screen with irrelevant signals and gates.

Lab 1: Adders and Flip-Flops

1 Overview

This lab is divided into two main problems. In Section 3 you will construct several circuits to a simple adder. In Section 4 you will construct a variety of flip-flops.

Req 1.1:   Students shall work individually or in groups of two.

2 Lab Setup

Action: Read Handout-1: Environment Configuration

Action: Read Handout-2: Local Scripts

Action: Copy the provided files to your local account. The commands below show how this can be done on an ECE-Linux computer if the desired target directory is /home/your_userid/ece327/lab1. In the commands, the symbol "%" means the Linux prompt.

```
% mkdir ~/ece327
% cd ~/ece327
% cp -rL /home/ece327/lab1 lab1
```

Action: Check that you have all of the files listed below:

```
add2_tb.sim        carry.vhd          myflipflop.vhd      sum.uwp sum.vhd
add2_tb.vhd        fulladder.uwp      sum_tb.sim
add2.uwp add2.vhd  fulladder.vhd      sum_tb.vhd
```

Code 1:   Register your group on Coursebook.

3 Adders

3.1 Sum

Code 2:   Complete the VHDL code in sum.vhd to implement the behaviour of the following Boolean expression:
sum = a xor b xor cin.

Action: Use the command below at the operating system command prompt to check for syntax and typechecking errors. Fix any errors in the source code until the code synthesizes.

```
% uw-com sum.vhd
```

Action: Use the command below to synthesize to generic gates and bring up the graphical interface for PrecisionRTL so that you can see the schematic:

```
% uw-synth -g --gui  sum.vhd
```

Q1: Look at the RTL schematic and describe the hardware that was synthesized (pins, gates, etc). Embed your answer as a comment after "`-- question 1`" in `sum.vhd`.

Action: Use the testbench in `sum_tb.vhd` to simulate your `sum` circuit. Run the simulation (i.e., generate the output waveforms) by using the following command at the command prompt:

```
% uw-sim sum.vhd
```

Note: When given a VHDL file *name*`.vhd`, `uw-sim` automatically looks for a testbench file *name*`_tb.vhd` and an optional simulation script *name*`_tb.sim`.

Q2: Describe the input and output waveforms using 1s and 0s and insert your answer as comments after "`-- question 2`" in `sum_tb.vhd`. An example answer is shown below, where the simulation was run for 30ns.

```
signal | waveform description
a        0 1 0
b        0 0 0
cin      0 0 1
sum      1 U 1
```

Q3: What happens when you run the simulation for 100 ns? Look at the testbench code in `sum_tb.vhd`. From the code, it is easy to see what values the signals `a`, `b`, and `cin` will have for the first 30ns. How do the VHDL semantics define what values these signals have after the first 30ns have passed? Embed your answer as a comment after "`-- question 3`" in `sum_tb.vhd`.

3.2 Carry

Code 3: Complete the VHDL code in `carry.vhd` to implement the behaviour of the following Boolean expression: `cout = (x and y) or (x and cin) or (y and cin)`.

Req 3.1: The entity shall be named `carry`. The architecture shall be named `main`.

Req 3.2: The circuit shall have three input ports and one output port. The ports shall be named to correspond with the signals in the equation for `cout` given above and shall be named to follow the convention where input ports begin with `i_` and output ports begin with `o_`.

Action: Use `uw-com` to ensure your code is legal VHDL.

3.3 One-bit Full Addder

Code 4: Using your `sum` and `carry` circuits, complete the code in `fulladder.vhd` to build a 1-bit full adder.

Req 3.3: The implementation shall use entity instantiations of the `sum` and `carry` circuits.

Action: Use the command below to ensure that your code is legal VHDL. *Note that the file extension is* `.uwp`.

```
% uw-com fulladder.uwp
```

Note: The command above uses the `uwp` project file, not the VHDL file. This is because we need to compile multiple files (`fulladder.vhd`, `add.vhd`, and `sum.vhd`). The project file lists the files to be compiled in the `DESIGN_VHDL` line.

In this particular case, you might be able to compile `fulladder.vhd` directly, because previously compiled versions of the `sum` and `carry` circuits are stored in the `uw_tmp` directory. However, if you run `uw-clean`, which removes `uw_tmp` and other temporary files and directories, then you will need to use the project file and not be able to compile `fulladder.vhd` by itself.
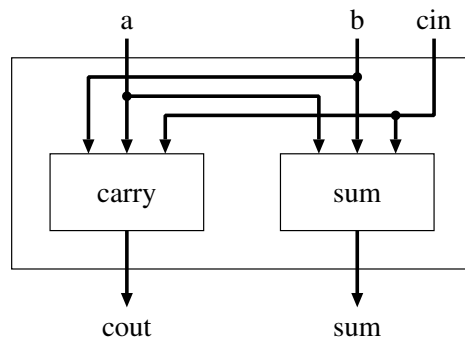
Figure 1: One-bit full adder

3.4 Two-bit Addder

Code 5: Using your `fulladder` circuit, complete the VHDL code in `add2.vhd` to build a 2-bit full adder as shown in Figure 2.

Req 3.4: The entity shall be named `add2` and the architecture shall be named `main`.

Req 3.5: The implementation shall use VHDL-93 style component instantiations of the `fulladder` circuit.

Action: Use `uw-synth` to ensure your code is legal and synthesizable VHDL.
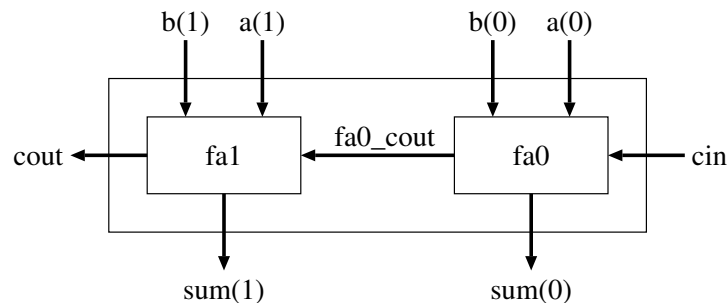
Figure 2: Two-bit adder

Code 6: Complete the VHDL code in `add2_tb.vhd` to implement a testbench that can be used to simulate your 2-bit full adder.

Req 3.6: The testbench shall use a VHDL-93 style component instantiation of the unit under test (`add2`).

Req 3.7: The testbench shall assign the input sequence below to the unit under test:

- Each column shall represent 10 ns of time

- `a` and `b` shall be 2-bit vectors.

```
signal | waveform description
a(0)      0 1 1 0
b(0)      0 0 1 0
a(1)      0 1 0 1
b(1)      0 0 0 1
cin       0 0 1 1
```

Action: Simulate your testbench using the following command at the command prompt:

```
% uw-sim add2.uwp
```

Q4: Describe the output waveforms using 1s and 0s in the same format as the input waveforms above. Insert your waveforms as comments after "`-- question 4`" in add2_tb.vhd.

4 Flip Flops

In this section, you will explore the basic flip-flop and the four variations shown in Figure 3.

Basic flip-flop   a) Flip-flop with synchronous reset   b) Flip-flop with chip enable

c) Flip-flop with mux on input   d) Flip-flop with feedback
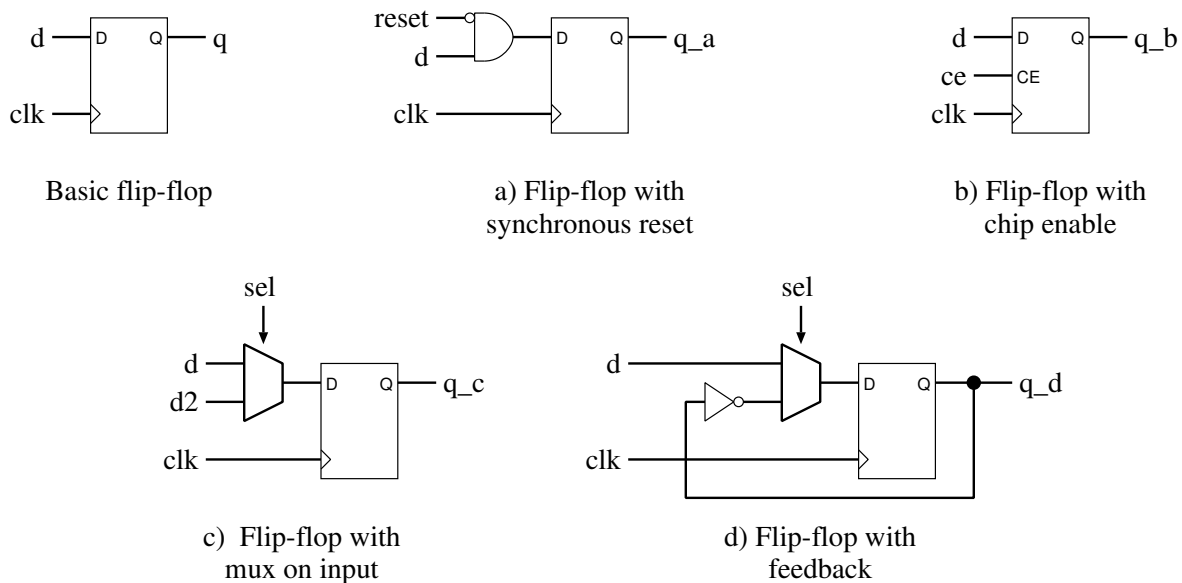
Figure 3: Flip flops

4.1 Flip-flop with Synchronous Reset

Code 7: Write the VHDL code to create a flip-flop with a synchronous reset pin as shown in Figure 3(a). The circuit may show up as drawn, or it may show up as a flip-flop named "DFFRSE". For this flip-flop, the reset pin is synchronous.

Req 4.1: If the value on the reset pin is '1' on the rising edge of the clock, the output of the flip-flop shall be a '0'. Otherwise, the output of the flip-flop output shall match that of the basic flip-flop.

Req 4.2: The code shall be added to the process labelled proc_a in myflipflop.vhd.

Req 4.3: Use the following signals:

clk	clock
i_d	data input
reset	reset
o_q_a	data output

Action: Use uw-synth to synthesize the circuit and confirm that the schematic is functionally equivalent to the Figure 3(a).

4.2 Flip-flop with Chip Enable

Code 8: Write the VHDL code to create a flip-flop with a chip-enable pin as shown in Figure 3(b).

Req 4.4: If the value on the chip-enable pin is '0' on the rising edge of the clock, the data output shall remain unchanged. Otherwise, the data output shall match that of the basic flip-flop.

Req 4.5: The code shall be added to the process labelled `proc_b` in `myflipflop.vhd`.

Req 4.6: Use the following signals:

`clk`	clock
`ce`	chip-enable
`i_d`	data input
`o_q_b`	data output

Action: Use `uw-synth` to synthesize the circuit and confirm that the schematic is *functionally* equivalent to Figure 3(b). (The schematic might not be structurally identical to the figure.)

4.3 Flip-flop with Mux on Input

Code 9: Write the VHDL code to create a flip-flop with a mux on the input of the flip-flop as shown in Figure 3(c).

Req 4.7: If `sel = 0`, the mux shall select `d`. Otherwise, the mux selects `d2`.

Req 4.8: The output of the mux shall be connected to the input of the flip-flop.

Req 4.9: The code shall be added to the process labelled `proc_c` in `myflipflop.vhd`.

Req 4.10: The code shall use the following signals:

`clk`	clock
`i_d`	data input
`i_d2`	alternative data input
`i_sel`	multiplexer select
`o_q_c`	data output

Action: Use `uw-synth` to synthesize the circuit and confirm that the schematic is functionally equivalent to Figure 3(c).

4.4 Flip-flop with Feedback

Code 10: Write the VHDL code to create a flip-flop with feedback through an inverter (with a mux on the input to flip-flop) as shown in Figure 3(d).

Req 4.11: If `sel = 0`, the mux shall select `d`. Otherwise, the mux shall select the output of the inverter.

Req 4.12: The output of the mux shall be connected to the input of the flip-flop.

Req 4.13: The code shall be added to the process labelled `proc_d` in `myflipflop.vhd`.

Req 4.14: The code shall use the following signals

`clk`	clock
`i_d`	data input
`i_sel`	multiplexer select
`o_q_d`	data output

Note: You may also need to introduce an intermediate signal if you cannot read directly from the `o_q_d` signal.

Action: Use `uw-synth` to synthesize the circuit and confirm that the schematic is functionally equivalent to Figure 3(d).

5   Lab Submission

Action: To submit the lab run the following command at the command prompt *from the directory where your lab files are located*:

```
% ece327-submit-lab1
```

Req 5.1:   Submit the lab by the due date listed on the schedule web page for the course. The penalty for late submissions is 50% of the lab mark. Labs submitted more than three days past the due date will receive a mark of zero.

1 Overview

This lab is divided into two main tasks: the design and analysis of a heating system and the design and analysis of a shift-add multiplier.

Req 1.1: Students shall work individually or in groups of two.

2 Lab Setup

Action: Copy the provided files to your local account. The commands below show how this can be done on an ECE-Linux computer if the desired target directory is /home/*your_userid*/ece327/lab2. In the commands, the symbol "%" means the Linux prompt.

```
% cd ~/ece327
% cp -rL /home/ece327/lab2 lab2
```

Action: Check that you have all of the files listed below:

```
heatingsys.uwp          mult.uwp                util.vhd
heatingsys.vhd          mult.vhd
heatingsys_tb.vhd       util_unsynth.vhd
```

Action: Register your group on Coursebook.

3 Heating System

3.1 Heating System Implementation

Code 1: Create a state-machine-based system to control a furnace, using the state diagram in Figure 1 and the VHDL code provided in heatingsys.vhd.

Req 3.1: The system shall switch between the three states based on the transition equations in the diagram.

Req 3.2: You shall add your design to the existing VHDL code in heatingsys.vhd.

Req 3.3: All state transitions shall occur on a rising edge of the clock.

Req 3.4: The system shall have a synchronous reset. That is, if reset has the value of '1' on a rising edge of the clock, the system shall return to the "off" heating mode.

Note: You may include more than one process in your design.

Note: If you are building the state machine using case statements, add a case for others at the end to handle the exceptional cases.

Action: Synthesize your system using `uw-synth -g --gui`, then open up the RTL schematic to examine the hardware synthesized from your VHDL code.

Note: Be sure to look at *all* of the pages in the schematic.

Note: The number of flip-flops should be consistent with the number of states in the state-machine diagram.

Q1: How many of each of the following gates are present in your design? 1-bit flip-flops, 1-bit latches, ANDs, ORs, XORs, NOTs, adders, subtracters, comparators, multiplexers. Embed your answer as a comment in `heatingsys.vhd`.

Note: Depending upon your VHDL code, the synthesis tool might synthesize your arithmetic operations to arithmetic units (*e.g.*, "`c <= a - b;`" would be synthesized into a subtractor) or the synthesis tool might decompose the operation into Boolean equations and synthesize the operation down to random-logic of ANDs, ORs, etc. You do not need to reverse-engineer the primitive gates to figure out what arithmetic operation they implement. Simply give your answer in terms of the gates that you see in the schematic. For combinational gates, you do not need to report their width (number of inputs).

Note: For flip-flops, you must check how wide they are and report the number of equivalent 1-bit flops. For example, one 2-bit flop would be reported as two 1-bit flops.
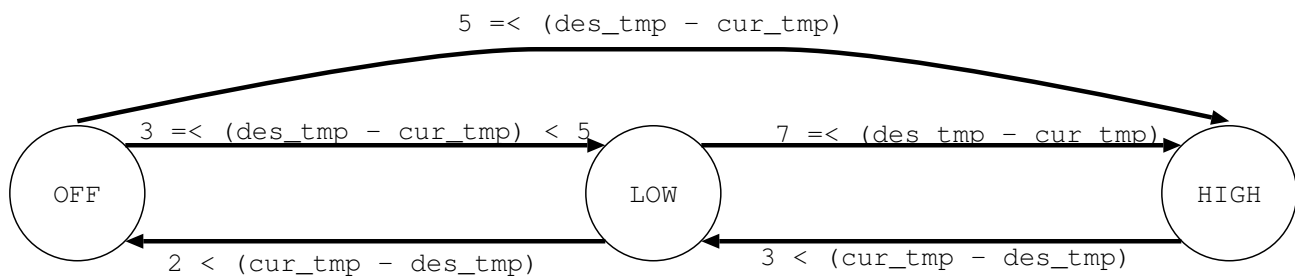
Figure 1: Heating-system state-transition diagram

3.2 Heating System Testbench

Code 2: Create a testbench to stimulate your implementation.

Req 3.5: The testbench shall test *each* state transition and the "reset" functionality.

Req 3.6: The testbench entity shall be named `heatingsys_tb` and the architecture shall be named `main`.

Req 3.7: The testbench code shall be in a file named `heatingsys_tb.vhd`.

Action: Use `uw-sim` to simulate and debug your testbench and system.

Note: You might want to create a file named `heatingsys_tb.sim` that contains the simulation commands (see the `.sim` files from previous simulations for sample commands). Alternatively, you may use the GUI to select signals and enter commands.

4 Shift-Add Multiplier

In this section, you will build a shift-add multiplier.

4.1 Algorithm

The shift-add algorithm is an iterative algorithm to compute the product of two numbers (a and b). The result (z) is initialized to 0.

In each iteration, three things happen.

1. If a is odd (least-significant bit is 0), then the current value of b is added to the result.

2. a is divided by two (shifted to the right by one bit)

3. b is multiplied by two (shifted to left by one bit).

Pseudocode:

```
z=0
while a > 0 {
  if a(0) == '1' then {
    z = z + b;
  }
  a = a >> 1;
  b = b << 1;
}
```

Example in Decimal:

$$a = 13$$
$$b = 5$$

$$
\begin{aligned}
a \times b &= 13 \times 5 \\
&= (8 + 4 + 1) \times 5 \\
&= \quad (1 \times 5) \\
&+ \quad (4 \times 5) \\
&+ \quad (8 \times 5) \\
&= 65
\end{aligned}
$$

Example in Binary:

a	b	z
00001101	00000101	00000000
00000110	00001010	00000101
00000011	00010100	00000101
00000001	00101000	00011001
00000000	00010000	01000001

4.2 Design

Req 4.1: Your system shall implement the shift-add algorithm to compute the product of two 8-bit natural numbers.

Req 4.2: Your system shall ignore overflow.

Req 4.3: The inputs and outputs of the system shall be:

Inputs	**Outputs**		
reset			
clk			
i_valid	o_valid		
i_a, i_b	o_z	:	unsigned(7 downto 0)

Req 4.4: The parcel schedule shall be "unpredictable number of bubbles".

Action: Draw a state-machine diagram to implement the system and save the picture as "`mult-prelim.pdf`" in your `lab2` directory. This drawing is part of the deliverables for this lab.

4.3 Simulation and Testbench

Action: Simulate the testbench and the skeleton implementation of the multiplier:

```
% uw-sim mult.uwp
```

Look at the waveforms and notice that the signals `a` and `b` remain 'X' throughout the simulation. This is caused by a bug in the testbench.

Code 3: Fix the bug in the testbench.

Action: Explain the bug and how your change fixed the problem. Insert your answer as a comment after `question 1` in `mult_tb.vhd`.

4.4   Implementation

Code 4:   Complete the implementation of the shift-add multiplier in `mult.vhd`.

Action: Use the command below to synthesize your multiplier to generic gates:
```
% uw-synth --gates mult.uwp
```

Action: Use the command below to simulate the generic-gate netlist of your multiplier. Fix any bugs that you discover.
```
% uw-sim --gates mult.uwp
```

Action: Use the command below to do logic-synthesis of your multiplier to FPGA cells:
```
% uw-synth --logic mult.uwp
```

Action: Use the command below to simulate the logic-synthesis netlist of your multiplier. Fix any bugs that you discover.
```
% uw-sim --logic mult.uwp
```

Action: If you discover any mistakes in your "`mult-prelim.pdf`" drawing while doing the coding, do a second drawing "`mult-final.pdf`". On the drawing, explain the errors and how you corrected them.

Req 4.5:   Your implementation shall have the same behaviour as either "`mult-prelim.pdf`" or "`mulf-final.pdf`".

5   Lab Submission

Action: Run the following command at the prompt *from the directory where your lab files are located*:
```
% ece327-submit-lab2
```

Req 5.1:   Submit the lab by the due date listed on the schedule web page for the course. The penalty for late submissions is 50% of the lab mark. Labs submitted more than three days past the due date will receive a mark of zero.

Lab3: Preview of the Project

1   Overview

Lab 3 is a simplified version of the ece327 course project.

In the project, you will design a *Kirsch Edge Detector* for image processing. For the project, the image will be sent to your system byte-by-byte from a PC and your system will send the output image back to the PC.

For lab3, a two-dimensional matrix of data will be sent byte-by-byte from a PC to your system, your system will run some calculations on the data, and the result will be output to a seven-segment display on the FPGA board.

Req 1.1: Students shall work individually or in groups of two.

Note: This lab is much more challenging than Labs 1 and 2, so start early and be prepared to spend more time than on the earlier labs.

Note: Your code will be marked only on functionality and good coding practices. You do not need to optimize your code for area or clock speed.

Note: If you do a good job in this lab, you will probably be able to re-use most of your code in the project.

2   Algorithm

The purpose of this lab is to iterate over the rows and columns of a 16 by 16 matrix (M) and calculate the number of times that M[y-2,x] - M[y-1,x] + M[y,x] is non-negative. Figure 1 illustrates a 16×16 byte matrix with random data, and pseudo code to implement the algorithm in software.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	8	7	1	2	5	0	9	0	8	7	1	2	5	0	0
1	3	4	5	1	4	8	2	6	4	6	3	2	6	3	3	2
2	1	5	0	9	0	8	7	1	2	5	0	0	8	7	1	7
3	4	6	3	2	6	3	3	4	8	2	6	4	7	3	3	3
4	3	3	4	6	4	4	0	9	0	0	6	3	2	6	8	0
5	6	3	2	6	3	3	0	9	3	4	8	6	6	2	6	0
6	0	9	0	9	0	0	9	0	9	0	0	0	6	4	7	9
7	4	6	3	2	6	3	3	2	3	4	8	2	6	4	7	3
8	2	6	4	6	3	9	9	0	0	8	7	3	3	0	3	9
9	8	2	6	4	6	3	6	3	2	2	6	3	3	8	7	6
10	2	6	4	0	0	9	0	6	4	1	4	8	8	7	2	0
11	0	6	3	2	5	0	2	6	4	8	9	0	0	7	2	2
12	4	8	6	2	6	3	6	4	0	9	3	4	8	7	3	6
13	0	0	0	6	3	3	0	6	3	3	2	3	4	2	8	0
14	2	6	4	3	3	3	3	8	6	3	3	2	6	3	9	3
15	4	6	3	2	6	3	3	2	3	4	8	2	6	4	7	3

```
count = 0;
for y = 0 to 15 {
  for x = 0 to 15 {
    M[y,x] = a;
    p     = M[y-2,x] - M[y-1,x] + M[y,x];
    if p >= 0 and y >= 2 then {
      count = count + 1;
    }
    z = count;
  }
}
```

Figure 1: Example matrix and pseudo-code for algorithm

The data comes in, byte by byte, in the following order:

$$M_{0,0},\ M_{0,1},\ M_{0,2},\ \ldots,\ M_{0,13},\ M_{0,14},\ M_{0,15},\ M_{1,0},\ M_{1,1}, \ldots, M_{15,13},\ M_{15,14},\ M_{15,15}$$

As an example using the above matrix, the bytes come in the following sequence:

$$0, 8, 7, 1, 2, 5, 0, 9, 0, 0, 8, 7, 1, 2, 5, 0, 0, 3, 4, 5, 1, 4, 8, 2, \ldots$$

We begin updating `count` as soon as row 2 column 0 is ready. Starting from this point, we calculate `p` for each incoming byte, and increment `count` if `p` is greater than or equal to zero.

As soon as we begin processing row 3, we no longer need the data from row 0. This allows us to use a memory array with just 3 rows, rather than a full set of 16 rows.

The processing of data proceeds as follows:

1. Write the new byte into the appropriate location in a 3×16 byte memory as shown below. The first input byte after reset is written into row 0 column 0. The next input byte is written into row 0 column 1, and so on. Proceed to the first column of the next row when the present row of memory is full.

<center>16 columns</center>

–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–

<center>3 rows</center>

2. The following shows a snapshot of the memory when row 2 column 0 is ready, which is the first byte for which we update `count`.

Virtual row index

0	a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}	a_{12}	a_{13}	a_{14}	a_{15}
1	b_0	b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_8	b_9	b_{10}	b_{11}	b_{12}	b_{13}	b_{14}	b_{15}
2	c_0	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–

3. Starting from this point, perform the operation $p = a_i - b_i + c_i$, (where $i = 0$) for column 0 as shown above. If $p \geq 0$ then, increment the counter.

4. When row 2 is full, the next available byte is written into row 0 column 0. Although physically this is row 0 column 0, virtually you can imagine that it is row 3 column 0. The operation $p = a_i - b_i + c_i$ can be performed based on the virtual row index:

Virtual row index

3	c_0	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
1	a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}	a_{12}	a_{13}	a_{14}	a_{15}
2	b_0	b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_8	b_9	b_{10}	b_{11}	b_{12}	b_{13}	b_{14}	b_{15}

5. The same concepts apply to the rest of incoming bytes. At the end, `o_data` has the value of the counter (the total number of "$p \geq 0$").

The internal memory for your design shall be constructed from instances of the 1-byte by 16-entry memory array in `mem.vhd`. The benefit of using multiple instances of the memory, rather than a single 3-byte by 16-entry memory is to allow multiple reads and/or writes to different addresses in the same clock cycle.
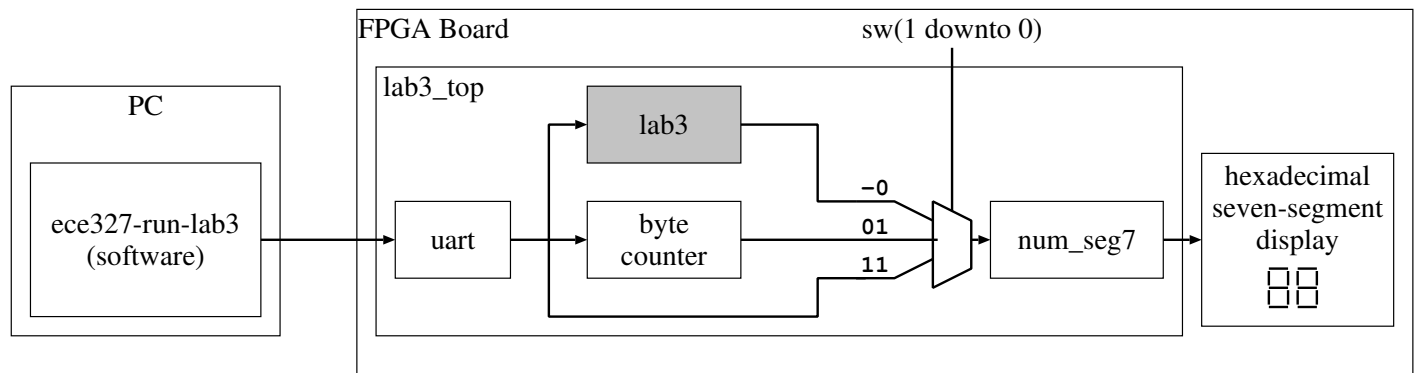
ece327-run-lab3	Software program to send data from PC to the FPGA board.
lab3_top	Top-level entity, which instantiates the other entities as components.
uart	Universal asynchronous receiver/transmitter (UART) to receive data from the PC.
lab3	The core of the system. This is the part of the system that you will implement. The rest of the system is provided to you.
byte counter	Counts the number of input bytes received.
num_seg7	Seven-segment display controller. Converts an eight-bit unsigned value to hexadecimal digits to be displayed on the seven-segment display.

Figure 2: Lab3 system architecture

Core Files

`util.vhd`	Utility functions (synthesizable)
`mem.vhd`	VHDL code for the memory array
`lab3.vhd`	Template for your design
`util_unsynth.vhd`	Utility functions used in the testbench (unsynthesizable)
`lab3_unsynth_pkg.vhd`	Types and functions used by testbench (unsynthesizable)
`lab3_tb.vhd`	Testbench
`lab3_tb.sim`	Simulation script
`lab3.uwp`	Project file for simulation and synthesis of your part of the design (the core)

Wrapper Files

`components.vhd`	Library including uart and seven-segment display controller
`lab3_top.vhd`	Top level code for hardware to run on FPGA
`lab3_top.uwp`	Project file for the top-level code with uart and seven-segment display

Test Files

`tests/test*.txt`	Text-format test files

Table 8.1: Provided files

3 System Architecture and Provided Code

Figure 2 shows the architecture of the system. You will implement the core of the system, which is named `lab3`. The other modules are provided for you (Table 8.1).

Table 8.2 shows the input and output signals in the `lab3` entity, which is the core of the design.

Functional Inputs		
`i_valid`	`std_logic`	Input data is valid.
`i_data`	`unsigned(7 downto 0)`	Input data.

Functional Outputs		
`o_done`	`std_logic`	Done processing
		Processing is completed and `o_data` has the final value.
`o_data`	`unsigned(7 downto 0)`	Output data.
		The final value of the counter is available on this output port at the end of processing.

Table 8.2: Interface signals for `lab3` entity

4 System Requirements

Req 4.1: When `reset` is asserted (`'1'`) on the rising edge of `clk`, the system shall be reset. After reset is deasserted (drops to `'0'`), the next byte to arrive will be the first byte of a new matrix.

Req 4.2: Your design shall allow multiple matrices to be sent consecutively without reset being pushed between the end of one matrix and start of the next matrix.

Req 4.3: The system shall calculate the correct result for the full range of input data values (e.g. from $0 - 255 + 0$ to $255 - 0 + 255$).

Note: The input and output data are *unsigned*. To calculate the results correctly, your internal calculations must be *signed*.

Constraint: The `i_valid` signal is asserted (`'1'`) for exactly one clock cycle. The output data from the UART will be valid only during the one clock cycle when `i_valid` is asserted.

Req 4.4: Your design shall work for an "unpredictable number of bubbles" parcel schedule with a minimum of 3 bubbles.

Req 4.5: The signal `o_done` shall be reset to `'0'` and shall remain `'0'` until `o_data` has the final result for processing a matrix.

Req 4.6: After processing a matrix, the `o_data` shall hold final result and `o_done` shall remain `'1'` until the next matrix enters the system or reset is asserted.

Note: The value of `o_data` is unconstrained while `o_done='0'`.

Req 4.7: The files below shall not be modified. The synthesis and simulation of your design for marking will be done with the provided versions of these files. Any modifications that you make will be overwritten.

```
components.vhd          mem.vhd
lab3_tb.vhd             util_unsynth.vhd
lab3_unsynth_pkg.vhd    util.vhd
```

5 Suggested Design Procedure

5.1 FSM Design

Note: Requirement 2 (consecutive images sent without reset) and Requirement 6 (hold o_data constant after finish processing) are lower priority than the other requirements. It is probably a good idea to postpone working on these requirements until the rest of your design is working correctly all the way through to the FPGA board. Not meeting one of these requirements will result in only a 2% deduction in the mark on the lab. Not meeting both of these requirements will result in a deduction of 4%. Remember that this lab is worth only 3% of the total mark in the course.

Action: Draw a state machine for the system with a full 16 rows of memory.

Action: Write pseudo-code for the system using only three rows of memory.

Action: Modify the pseudo-code so that your index into the three rows of memory uses a one-hot encoding.

Action: Draw a FSM for your system. Save the drawing as lab3_prelim.pdf. **This is a deliverable due at the end of the first week of the lab. Check the course web page for the exact due date.**

5.2 Lab Setup

Action: Copy the provided files to your local account. The commands below show how this can be done on an ECE-Linux computer if the desired target directory is /home/your_userid/ece327/lab3. In the commands, the symbol "%" means the Linux prompt.

```
% cd ~/ece327
% cp -rL /home/ece327/lab3 lab3
```

Action: Check that you have all of the files listed below:

```
components.vhd          lab3_top.vhd            mem.vhd
lab3_tb.sim             lab3_unsynth_pkg.vhd    util_unsynth.vhd
lab3_tb.vhd             lab3.uwp                util.vhd
lab3_top.uwp            lab3.vhd
```

Action: Register your group on Coursebook.

5.3 Coding

Action: Write the VHDL code for your FSM.

Note: Decompose your design into *conceptual* modules such that each module writes to a set of closely related signals. These modules do *not* need to be implemented as separate entity/architectures. A "module" can be a process or a set of closely related processes. The important thing is that your concept of your design is modular and not monolithic. For example, the memory address is not closely related to count.

Note: Your final implementation should require less than 200 lines of code. If you find your code growing to be more than 200 lines: stop, revisit your FSM, and re-examine your code. The most common cause of code bloat is control circuitry. Use the concepts and examples from class.

5.4 Functional Simulation

Action: Read the testbench to learn its behaviour and coding techniques. Pay particular attention to the comments.

Action: Five test files (`test1.txt...test5.txt`) are given where each test contains a $16{\times}16$ matrix. To switch between test cases, use a `-G` or `--generics` flag with `uw-sim`:

```
% uw-sim -Gtest_num=4 lab3.uwp
```

Note: When `o_done='1'`, the testbench checks `o_data` signal and prints a PASS/FAIL message. If you do not see a message:

1. Check the waveforms to see if `o_done` is stuck at `'0'`
2. If the waveforms show `o_done='1'`, then study the testbench, paying particular attention to the comments.

Note: There is a generic value in `lab3_tb.vhd` named `bubbles` that determines the number of bubbles between valid data. You may use the `-G` or `--generics` flag with `uw-sim` to modify the value of `bubbles` (Requirement 4).

Note: If your implementation generates the correct outputs for all five tests, you can assume that your design is functionally correct.

5.5 Synthesis and Logic Simulation

Action: Synthesize your design

Action: Perform gate and logic simulation on your design

Note: Make sure your design works with logic simulation before proceeding to running on the FPGA board. It is much easier to debug in simulation than to debug on the board.

Q1: What are the number of flip-flops and lookup-tables used by your design after logic synthesis? (Embed your answer to this question and the ones that follow as comments in `lab3.vhd`)

Q2: What is the maximum clock frequency in MHz after logic synthesis?

Q3: What are the source and the destination signals of your design's critical path after logic synthesis?

Note: The information required to answer the questions above should be done in this part before going to part 3 (where `lab3_top` will be wrapped around your code).

5.6 Download to FPGA and Running on the FPGA

For this part, you will use `lab3_top.uwp`, not `lab3.uwp`. The difference between the two project files is that `lab3_top.uwp` includes a UART and seven-segment display controller that connects your core `lab3` to the serial port and seven-segment display on the FPGA board.

Action: On an *ecelinux* computer, synthesize your design for the FPGA by running the command below

```
% uw-synth -c lab3_top.uwp
```
confirm that the file lab3_top.sof has been created

Action: *In a DOS Window on a Nexus PC* that is connected to an FPGA, download the design to the FPGA board by running the commands below:

```
% y:\ece327-setup.bat
```
If the above command is not found, then run: `Q:\eng\ece\util\327setup.bat`
```
% uw-dnld lab3_top.uwp
```

Note: Press the `reset` button on the FPGA board to reset your system.

Action: On the FPGA board, turn switches 1 and 0 *off*.

Action: Go to the tests directory and run the first test case:
```
% cd tests
% ece327-run-lab3 test1.txt
```
The program will send test number 1 to the board. The result that your circuit generates will be displayed on the seven-segment display on the board.

Note: The program `ece327-run-lab3` and `lab3_top.vhd` have several features to help with debugging your design on the FPGA:

- Switches 1 and 0 on the FPGA board control what data is sent to the seven-segment display:

sw(1 downto 0)	display
-0	o_data
10	the number of pixels received from the PC
11	the most recently received pixel from the PC

- `ece327-run-lab3` has a `-s` flag to control the number of pixels that are sent contiguously to the FPGA. For example, to send just one pixel at a time to the FPGA when running test 3, use:
```
% ece327-run-lab3 -s1 test3.txt
```

Note: The data transfer from the PC to the FPGA is slow compared to the clock speed on the FPGA (115kbps vs 50 MHz). When running on the FPGA, there are several hundred bubbles between each incoming byte.

Q4: Was your design successful for all 5 tests? If your results do not match, explain where you think the error is and how you would attempt to debug and fix the problem if given more time.

Action: Draw a final FSM of your system. Briefly describe the major changes from your preliminary design. Save the drawing as `lab3_final.pdf`.

6 Deliverables and Submission

This lab has two deliverables, the state-machine diagram due at the end of the first week and the VHDL source code due at the end of the second week.

FSM To submit the state-machine diagram of your system, you must have a PDF file named "`lab3_prelim.pdf`". If you are submitting a scanned image or photo, ensure that the image is sharp, in focus, and well lit. Marks will be deducted for submissions that are difficult to read.

Run the following command on an ecelinux computer from the directory where your `lab3_prelim.pdf` file is located:
```
% ece327-submit-fsm
```

Source code To submit your lab, run the following command on an ecelinux computer from the directory where the .vhd files are located:
```
% ece327-submit-lab3
```

The penalty for late labs is 50% of the lab mark. Labs submitted more than 3 days past the due date will receive a mark of 0.

ECE-327	Digital Systems Engineering	2018t2 (Spring)
Project: Kirsch Edge Detecter		

1   Overview

The purpose of this project is to implement the Kirsch edge detector algorithm in VHDL. The design process includes exploring a reference model of the algorithm, creating a dataflow diagram, implementing the design in VHDL, optimizing and verifying the design, and finally downloading the optimized design to the FPGA Board. The project is to be done in groups of four.

	wk1	wk2	wk3	wk4
Algorithm design and algebraic optimizations	▓			
Dataflow diagram	▓▓			
Implement thin-line: UART⇒mem⇒UART		▓		
Test and debug thin-line in simulation		▓		
Test and debug thin-line on FPGA board		▓		
RTL coding			▓	
Test and debug functional simulation			▓	
Area optimizations			▓	
Speed optimizations				▓
Test and debug timing simulation				▓
Test and debug on FPGA board				▓
Peephole optimizations				▓
Write report				▓

Figure 1: Suggested schedule

There are five deliverables for the project. The list below describes each deliverable, the method of submission, and the approximate due date. The specific due dates are listed on the course web site.

Deliverable	Due Date	Submission Method
Group registration	End of week 2	Coursebook
Dataflow diagram	End of week 2	`ece327-submit-dfd`
Main Project and Report	End of week 4	`ece327-submit-proj`
Demo	1 week after project due date	Coursebook

1.1 Edge Detection

In digital image processing, each image is quantized into pixels. With gray-scale images, each pixel indicates the level of brightness of the image in a particular spot: 0 represents black, and with 8-bit pixels, 255 represents white. An edge is an abrupt change in the brightness (gray scale level) of the pixels. Detecting edges is an important task in boundary detection, motion detection/estimation, texture analysis, segmentation, and object identification.

Edge information for a particular pixel is obtained by exploring the brightness of pixels in the neighborhood of that pixel. If all of the pixels in the neighborhood have almost the same brightness, then there is probably no edge at that point. However, if some of the neighbors are much brighter than the others, then there is a probably an edge at that point.

Measuring the relative brightness of pixels in a neighborhood is mathematically analogous to calculating the derivative of brightness. Brightness values are discrete, not continuous, so we approximate the derivative function. Different edge detection methods (Prewitt, Laplacian, Kirsch, Sobel etc.) use different discrete approximations of the derivative function. In the E&CE-327 project, we will use the Kirsch edge detector algorithm to detect edges in 8-bit gray scale images of 256×256 pixels. Figure 2 shows an image and the result of the Kirsch edge detector applied to the image.
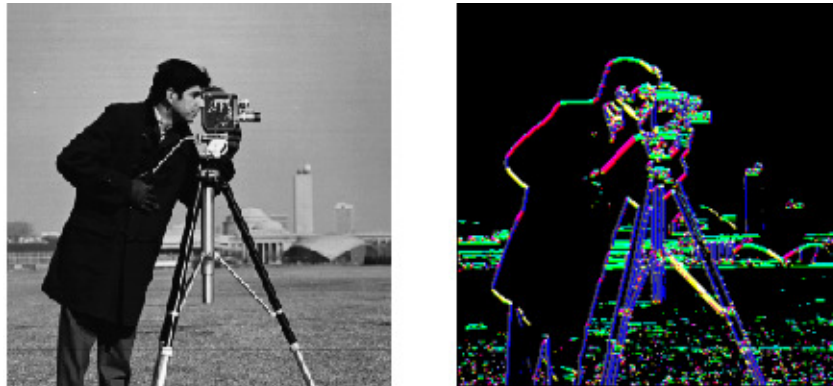
Figure 2: Cameraman image and edge map

The Kirsch edge detection algorithm uses a 3×3 table of pixels to store a pixel and its neighbors while calculating the derivatives. The 3×3 table of pixels is called a *convolution table*, because it moves across the image in a convolution-style algorithm.

Figure 3 shows the convolution table at three different locations of an image: the first position (calculating whether the pixel at [1,1] is on an edge), the last position (calculating whether the pixel at [254,254] is on an edge, and at the position to calculate whether the pixel at $[i, j]$ is on an edge.

Figure 4 shows a convolution table containing the pixel located at coordinate $[i, j]$ and its eight neighbors. As shown in Figure 3, the table is moved across the image, pixel by pixel. For a 256×256 pixel image, the convolution table will move through 64516 (254×254) different locations. The algorithm in Figure 5 shows how to move the 3×3 convolution table over a 256×256 image. The lower and upper bounds of the loops for i and j are 1 and 254, rather than 0 and 255, because we cannot calculate the derivative for pixels on the perimeter of the image.

First position
of convolution
table

0   *j*   255

0 —

i —

255 —

Position
of convolution
table for pixel [i,j]

Last position
of convolution
table
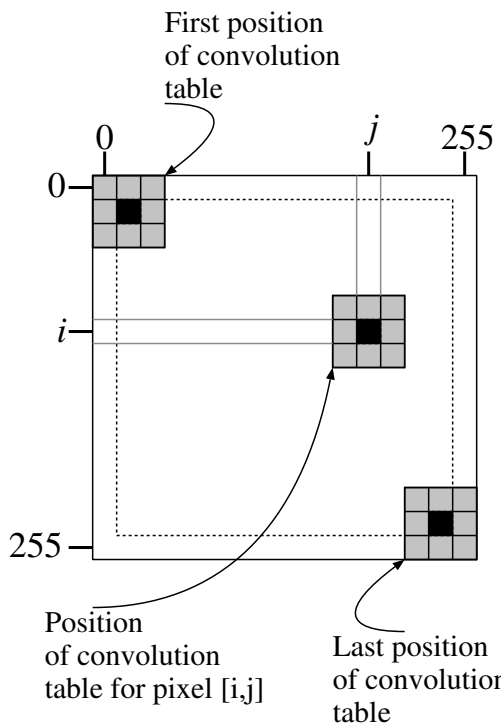
Figure 3: 256×256 image with 3×3 neighborhood of
pixels

Im[i-1, j-1]	Im[i-1, j]	Im[i-1, j+1]
Im[i , j-1]	Im[i , j]	Im[i , j+1]
Im[i+1, j-1]	Im[i+1, j]	Im[i+1, j+1]

Figure 4: Contents of convolution table to detect edge at
coordinate $[i, j]$

```
for i = 1 to 254 {
  for j = 1 to 254 {
    for m = 0 to 2 {
      for n = 0 to 2 {
        table[m,n] = image[i+m-1, j+n-1];
      }
    }
  }
}
```

Figure 5: Nested loops to move convolution table over image

[0,0]	[0,1]	[0,2]
[1,0]	[1,1]	[1,2]
[2,0]	[2,1]	[2,2]

Figure 6: Coordinates of 3×3 convolution table

a	b	c
h	i	d
g	f	e

Figure 7: Short names for elements of 3×3 convolution table

The Kirsch edge detection algorithm identifies both the presence of an edge and the direction of the edge. There are eight possible directions: North, NorthEast, East, SouthEast, South, SouthWest, West, and NorthWest. Figure 8 shows an image sample for each direction. In the image sample, the edge is drawn in white and direction is shown with a black arrow. Notice that the direction is *perpindicular* to the edge. The trick to remember the direction of the edge is that the direction points to the brighter side of the edge.
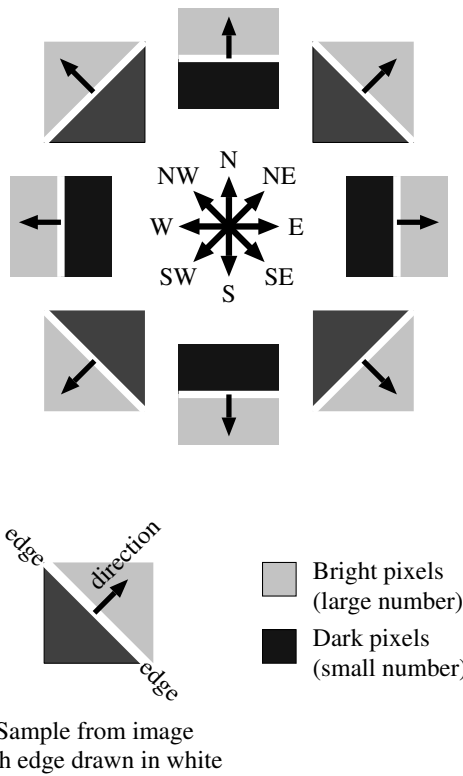
Figure 8: Directions
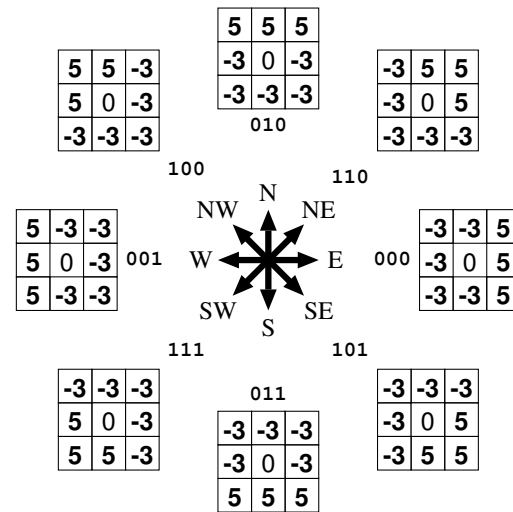
Figure 9: Masks

The equation for each derivative is defined in terms of a *convolution mask*, which is a 3×3 table of constants that are used as coefficients in the equation. Figure 9 shows the convolution mask and encoding for each direction. For example, the equation for the Northeast derivative is given below using coordinates shown in Figure 6:

$$
\begin{aligned}
Deriv\_NE \;=\;& 5\times(\text{table}[0,1] + \text{table}[0,2] + \text{table}[1,2]) - \\
& 3\times(\text{table}[0,0] + \text{table}[1,0] + \text{table}[2,0] + \text{table}[2,1] + \text{table}[2,2])
\end{aligned}
$$

For a convolution table, calculating the presence and direction of an edge is done in three major steps:

1. Calculate the derivative for each of the eight directions.

2. Find the value and direction of the maximum derivative.

$$
\begin{aligned}
\text{EdgeMax} \;&=\; \text{Maximum of eight derivatives} \\
\text{DirMax} \;&=\; \text{Direction of EdgeMax}
\end{aligned}
$$

Note: *If more than one derivative have the same value, the direction is chosen based on the following order, from highest priority to lowest priority:*
W, NW, N, NE, E, SE, S, SW.
For example, if Deriv_N and Deriv_E are equal, Deriv_N shall be chosen.

3. Check if the maximum derivative is above the threshold.

> if EdgeMax > 383 then
>   Edge = true
>   Dir = DirMax
> else
>   Edge = false
>   Dir = 000

1.2 System Implementation

Your circuit (`kirsch`), will be included in a top-level circuit (`kirsch_top`) that includes a UART module to communicate through a serial line to a PC and a seven-segment display controller (ssdc) to control a 2-digit seven-segment display. The overall design hierarchy is shown in Figure 10. The entity for `kirsch` is shown in Figure 11. To reduce the complexity of the project, we have provided you two wrapper files `kirsch_lib.vhd` and `kirsch_top.vhd` that contain the seven-segment display and UART.
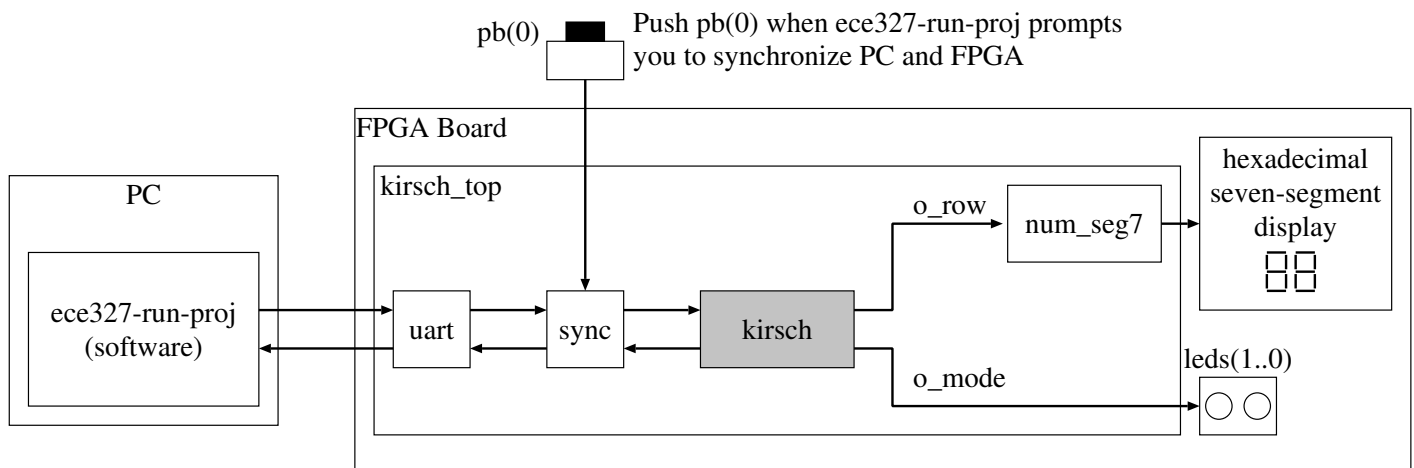
Figure 10: Kirsch system

1.3 Running the Edge Detector

You will run the Kirsch edge detector both in simulation and on the FPGA board. For simulation, you will use the provided testbench (`kirsch_tb.vhd`), which you may modify, or write your own testbench. For running on the FPGA, you will use the program `ece327-run-proj` to send an image to the FPGA board and receive the results.

Four 256×256 images are provided in text format for simulation and bmp format viewingand for download to the FPGA.

The testbench (`kirsch_tb.vhd`) can be used for functional simulation of the reference model and both functional and timing simulation of your design. The testbench reads an image from a text file, passes the data to the Kirsch circuit byte by byte, receives the outputs of the circuit (edges and directions) and stores them in a `.ted` text file.

In the `.ted` file, there are four columns: edge (0 or 1), direction (0 to 7 for each direction), row number (in the image), and column number. The ted file can be converted into a bitmap and vice versa using `ece327-run-proj`. The extension "ted" means "text edge direction".

There are several programs to analyze and debug `ted` files:

```
entity kirsch is
   port (
      clk        : in std_logic;                       -- clock
      reset      : in std_logic;                       -- reset signal
      i_valid    : in std_logic;                       -- is input valid?
      i_pixel    : in unsigned(7 downto 0);            -- 8-bit input
      o_valid    : out std_logic;                      -- is output valid?
      o_edge     : out std_logic;                      -- 1-bit output for edge
      o_dir      : out std_logic_vector(2 downto 0);   -- 3-bit output for direction
      o_mode     : out std_logic_vector(1 downto 0);   -- 2-bit output for mode
      o_row      : out unsigned(7 downto 0));          -- row number of the input image
      o_col      : out unsigned(7 downto 0))           -- col number of the input image
   );
end entity;
```

Figure 11: Entity for kirsch edge detector

diff_ted compares two .ted files and outputs the lines where they differ.

diff_ted_to_bmp Generates a .bmp bitmap image showing the differences between two .ted files:

diff_ted_to_bmp *file1.ted file2.ted diff.bmp*.

The colors in the resulting image are:

file1 edge	file2 edge	direction	bmp
1	1	same	white
1	1	diff	green
0	0	same	black
0	0	diff	pink
1	0	–	red
0	1	–	blue

ted_to_bmp converts a .ted file to a .bmp bitmap image. Each pixel in the input image is converted to a 2x2 block of pixels in the output, so that details in ouput image will be easier to see.

txt_to_bmp converts a .txt file to a .bmp image. NOTE: there might be some bugs in this code.

bmp_to_txt converts a .bmp image to a .txt file.

1.4   Provided Code

Top-level wrappers

`kirsch_top.uwp`	Project file with kirsch, uart, num_seg7 for download to FPGA board
`kirsch_top.vhd`	Top level code for for download to FPGA board
`kirsch_lib.vhd`	Source code for uart, ssdc.

Components and Packages

`mem.vhd`	VHDL code for the memory
`kirsch_synth_pkg.vhd`	Constants and types (synthesizable)
`kirsch_unsynth_pkg.vhd`	Constants, types, reading/writing images (unsynthesizable)
`string_pkg.vhd`	Functions for string conversion

Testbenches and simulation scripts

`kirsch_tb.vhd`	Main testbench
`kirsch_tb.sim`	Simulation script
`kirsch_spec.xls`	Spreadsheet reference model for debugging.

Kirsch core

`kirsch_spec.uwp`	Project file for Kirsch reference model specification
`kirsch_spec.vhd`	Reference model specification for Kirsch core
`kirsch.uwp`	Project file for your Kirsch core
`kirsch.vhd`	Source code for your Kirsh core
`tests (directory)`	4 test images:
	`*.txt` input images in text format for simulation
	`*.bmp` input images in bitmap format for running on FPGA

> **Note:** *Do **not** modify the following files:*
>
> | | |
> |---|---|
> | *componnets.vhd* | *kirsch_unsynth_pkg.vhd* |
> | *kirsch_spec.vhd* | *mem.vhd* |
> | *kirsch_synth_pkg.vhd* | *string_pkg.vhd* |
> | *kirsch_top.vhd* | *util.vhd* |
>
> *When your design is marked, we will use the original versions of these files, not the versions that are from your directory.*

> **Note:** *You may use any of the above files in your testbench. You may use* `kirsch_synth_pkg.vhd` *in your design. The files* `string_pkg.vhd` *and* `kirsch_unsynth_pkg.vhd` *are not synthesizable and therefore, your synthesizable code cannot use these files.*

2   Requirements

2.1   System Modes

The circuit shall be in one of three modes: idle, busy, or reset. The encodings of the three modes are shown in Table 9.1 and described below. The current mode shall appear on the o_mode output signal. The o_mode signal is connected to the LEDs.

- Idle mode: When the circuit is in idle mode, it either has not started processing the pixels or it has already finished processing the pixels.

mode	o_mode
idle	"10"
busy	"11"
reset	"01"

Table 9.1: System modes and encoding

- Busy mode: Busy mode means that the circuit is busy with receiving pixels and processing them. As soon as the first pixel is received by the circuit, the mode becomes busy and it stays busy until all the pixels (64KB) are processed, after which the mode goes back to the idle state.

- Reset mode: If reset='1' on the rising edge of the clock, then o_mode shall be set to "01" (and your state machine shall be reset) in the clock cycle after reset is asserted. The mode shall remain at "01" as long as reset is asserted. In the clock cycle after reset is deasserted (reset='0' on the rising edge of the clock), the mode shall be set to idle and the normal execution of your state machine shall begin. You may assume that reset will remain high for at least 5 clock cycles.

2.2   Input/Output Protocol

Pixels are sent to the circuit through the i_pixel signal byte by byte. The input signal i_valid will be '1' whenever there is a pixel available on i_pixel. The signal i_valid will stay '1' for exactly one clock cycle and then it will turn to '0' and wait for another pixel.

Your design shall support an unpredictable number of bubbles parcel schedule where the minimum number of bubbles is In general, the rate at which valid data arrives is unpredictable. Your design shall work with any number of bubbles at least 3.

When you run your design on the FPGA, there will be several hundred bubbles between parcels. The large number of bubbles is because the communication with the PC is quite slow compared to the FPGA clock speed.

The maximum latency through the edge detector *without penalty* is 8. Latencies greater than 8 will be penalized, as described in Section 5.3. The measurement for latency begins as soon as the first 3x3 table becomes available. For example, the latency will be 1, if the corresponding outputs of the first table become available in the immediate next clock cycle following the first valid table. In another words, as soon as the pixel in the third row and third column of the image arrives, the measurement for the latency begins. The number of "bubbles" is completely independent from the latency.

Whenever an output pair (o_edge and o_dir) become ready, o_valid shall be '1' for one clock cycle to indicate that the output signals are ready. If the pixel under consideration is located on an edge, o_edge shall be '1', otherwise the signal shall be '0'. o_dir shall be "000" if o_edge is '0' (no edge) and it shall show the direction of the edge if o_edge is '1'. The values of the signals o_edge and o_dir are don't cares if o_valid is '0'.

Your circuit shall **not** output a result for the pixels on the perimeter. That is, for each image, your circuit shall output $254 \times 254 = 64516$ results with o_valid='1'.

At the end of sending an image, at least 20 clock cycles will elapse before the first pixel of the next image will be sent.

Your circuit shall be able to process multiple images without a reset being asserted between the end of one image and the start of the next image.

2.3 Row Count of Incoming Pixels

The output signal o_row shall show the row number (between 0 and 255) for the most recent pixel that was received from the PC. The seven-segment controller in kirsch_top architecture displays the value of o_row on the seven segment display of the FPGA board.

The signal o_row shall be initialized to 0. When the last pixel of the image is sent to the FPGA, o_row shall be 255. At the end of processing an image, o_row shall remain at 255 until reset is asserted or pixels from the next image are received.

2.4 Memory

As illustrated below, you can do Kirsch edge detection by storing only a few rows of the image at a time. To begin the edge detection operations on a 3×3 convolution table, you can start the operations as soon as the element at 3rd row and 3rd column is ready. Starting from this point, you can calculate the operations for every new incoming byte (and hence for new 3×3 table), and generate the output for edge and direction.

Some implementation details are given below, where we show a 3×256 array. It is also possible to use a 2×256 array.

1. Read data from input (i_pixel) when new data is available (i.e. if i_valid = '1')

2. Write the new data into the appropriate location as shown below. The first byte of input data (after reset) shall be written into row 1 column 1. The next input data shall be written into row 1 column 2, and so on. Proceed to the first column of the next row when the present row of memory is full.

256 bytes

a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}	a_{12}	...	a_{254}	a_{255}
b_0													...		
													...		

3 rows (label for the table above)

3. The following shows a snapshot of the memory when row 2 column 2 is ready.

Row Idx

0	a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}	a_{12}	...	a_{255}	a_{256}
1	b_0	b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_8	b_9	b_{10}	b_{11}	b_{12}	...	b_{255}	b_{256}
2	c_0	c_1	c_2											....		

4. At this point, perform the operations on the convolution table below:

a_0	a_1	a_2
b_0	b_1	b_2
c_0	c_1	c_2

This requires 2 or 3 memory reads to retrieve the values from the memory (depending on how you design your state machine). Come up with a good design so that the write and read can be done in parallel.

5. When the next pixel (c_3) arrives, you will perform the operation on the next 3×3 convolution table:

a_1	a_2	a_3
b_1	b_2	b_3
c_1	c_2	c_3

6. When row 2 is full, the next available data shall be overwritten into row 0 column 0. Although physically this is row 0 column 0, virtually it is row 3 column 0. Note that the operations will not proceed until the 3rd element of the 4th row (d_2) is available, in which case the operation will be performed on the following table based on the virtual row index as depicted in the following figure.

Virtual
Row Idx

3	d_0	d_1	d_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}	a_{12}	...	a_{254}	a_{255}
1	b_0	b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_8	b_9	b_{10}	b_{11}	b_{12}	...	b_{254}	b_{255}
2	c_0	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_{10}	c_{11}	c_{12}	...	c_{254}	c_{255}

b_0	b_1	b_2
c_0	c_1	c_2
d_0	d_1	d_2

Your memory arrays shall be formed using instances of the 1×256 entry memory (provided in `mem.vhd`), where each entry is 8 bits wide.

Memory arrays shall be used only for storage of pixels, they shall not be used to mimic computation. In the past, a few groups have tried *unsuccessfuly* to reduce the number of lookup tables in their design by using memory arrays to mimic computation. While it is theoretically possible to use memory arrays for some types of computation, this approach is not relevant to Kirsch edge detection.

3 Design and Optimization Procedure

Follow the suggested schedule in Figure 1.

Begin by copying the files to your local directory. The procedure below shows how this can be done if the desired target directory is `/home/your_userid/ece327/proj`.

```
% cd ~/ece327
% cp -rL /home/ece327/proj  proj
```

3.1 Reference Model Specification

Use the reference model code of Kirsch with the provided testbench to run the four test cases and produce the results. To switch between test cases, set the value of the generic when running a simulation by passing a $-Gtestnum=i$ flag to the simulation program.

The suffix of the output `.ted` file is by default "sim" for "simulation. You may change this suffix using the `result_suffix` generic. When simulating the reference model, you should set the suffix to "spec", because `run-ece327-proj` will evaluate the correctness of your results by comparing them against the "spec" file.

To run the command line version of `uw-sim` without opening the graphical interface, use the `--nogui` flag.

```
% uw-sim --nogui -Gtest_num=1 -Gresult_suffix=spec kirsch_spec.uwp
```

You will use the resulting images to verify the functionality of your VHDL code. The reference model specification uses the same entity as your implementation. The reference model is not synthesizable.

3.2   Equations and Pseudocode

The Kirsch edge-detection algorithm was designed to be implemented simply and efficiently in hardware. You can do significant area optimizations very early in the design cycle by finding efficient ways to calculate the derivative equations. For example, identify algebraic optimizations and common subexpressions in the equations.

Several useful equations to apply in the optimizations are:

$$
\begin{aligned}
5a - 3b &= 8a - 3(a + b) \\
\max(a - c, b - c) &= \max(a, b) - c \\
\max(a + b, b + c) &= b + \max(a, c)
\end{aligned}
$$

3.3   Dataflow Diagram

The dataflow diagram shall show the calculation for the presence of an edge. The input to the dataflow diagram shall be the 3×3 window of pixels for the current position. The output shall be `o_edge`.

The pixels in the 3×3 window shall be labeled as:

a	b	c
h	i	d
g	f	e

The dataflow diagram does not need to include the calculation of `o_dir`. The dataflow diagram does not need to include writing to or reading from memory, updating the row and column counters, or the state machine.

In addition to the standard VHDL arithmetic operations, the dataflow diagram may use a MAX datapath component. For your dataflow diagram, you do not need to show the implementation of your MAX component.

In your VHDL code, you will need to implement your own MAX component, function, or procedure, because VHDL does not include a MAX operator. One complication to take into account is that your edge detector must compute both `o_edge` and `o_dir`, which means that your MAX will need to propagate *two* different values (a derivative and a direction). A function is probably the most concise option. But, a function may have only one result, so you will need to put the derivative and direction together into a record. A procedure may have multiple outputs and is slightly more concise than a component. A component does not require learning any new VHDL, but is the most verbose option.

The suggested process to go from your pseudocode to a dataflow diagram is:

1. Draw a data dependency graph.

2. Predict the slowest operation (e.g. memory access, 8-bit subtract, etc).

3. Estimate the maximum clock speed that can be achieved if a clock cycle contains just the slowest operation. This gives you an upper bound on the clock speed.

4. Based upon your maximum clock speed dataflow diagram, estimate the total latency of your system, including updating the memory arrays and convolution table. Compare your latency against the latency goal in Section 2.2. If your latency is excessive, reschedule the operations in your dataflow diagram, and potentially increase your clock period.

5. Choose your optimality goal. From your maximum clock speed and optimality target, calculate the maximum area that you can use.

6. Pick your optimality strategy: high-performance or low-area, and then pick your area and clock speed targets.

7. Based upon the throughput requirement and your dataflow diagram, decompose your design into pipeline stages.

8. Estimate the datapath resources and registers that your dataflow diagram uses.

9. Estimate the total area of your system by including allowances for the circuitry to update memory, the convolution table, and the control circuitry. (For most datapaths, an efficient implementation of the circuitry for control and memory can be done in about 160 FPGA cells.)

10. Estimate optimality of entire system.

> **Note:** *A week of all-night panicked debugging can save a day of thoughtful data-flow-diagram designing.*

3.4   Implementation

Begin with a thin-line implementation that simply loads data into memory, reads data from memory, and loads the data into the convolution table.

Do not begin coding the main part of your design until you are satisfied with your dataflow diagram and have the thin-line implementation working on the FPGA board.

Your VHDL code should be based on a dataflow diagram: either the one that you submitted for the first deliverable, or an improved diagram.

Use the memory component provided in `mem.vhd`.

If you create additional files for your source code, you will need to add the names of these files to `kirsch.uwp` and `kirsch_top.uwp`. For more information, see the web documentation on the format of the UWP project file.

3.5   Functional Simulation

To simulate your design, use the following command, with the number of the test that you want to run:

```
% uw-sim  -Gtest_num=3 kirsch.uwp
```

Use `diff_ted` or `diff_ted_to_bmp` to compare your resulting `.ted` file to the file produced by the reference model. For a line that is different in the two files, `diff_ted` uses `*` to show which fields are different.

```
% diff_ted tests/test3_spec.ted tests/test3_sim.ted
```

Use `ted_to_bmp` to convert a `.ted` file to a `.bmp` file and visually compare two `.bmp` files. The visual comparison can be very useful in detecting patterns in bugs (e.g. images are shifted or skewed).

When debugging your design, fast turnaround time on simulations is very important. You should develop some small test cases (e.g. 8×8 arrays) and modify the testbench and your design to work with these smaller arrays. To make it easy to change the size of the image that the circuit is using, your design should use the constants `image_height` and `image_width`, which are defined in `kirsch_synth_pkg.vhd`, rather than hardcoded values for the image size.

- Some groups made good use of Python and other scripting languages to generate tests and evaluate the correctness of results.

- The spreadsheet `kirsch_spec.xls` is designed for you to modify so that you can check the internal calculations of your design.

- The testbench contains a constant named "**bubbles**" that determines the number of clock cycles of invalid data (bubbles) between valid data. Make sure that your design works with any value of bubbles that is 3 or greater.

3.6 High-level Optimizations

- Many groups kept logs of their design changes and the effect on optimality scores. The vast majority felt that these logs were very helpful in achieving good optimality scores.

- Do area optimizations to reduce your area to your target before worrying much about clock speed. Area optimizations involve larger changes to a design than timing optimizations.

- Too much reuse of components can increase the number of multiplexers and amount of control circuitry.

- Estimate if your design has more flip-flops or lookup tables. If your area is dominated by lookup tables, there is no point in trying to save area by reducing the number of flip-flops. Conversely, if you area is dominated by flip-flops, then do not optimize the lookup tables.

- After you are within 5-10% of your area target, or are decreasing your area by less than 10% per day, change your focus to clock speed optimizations.

- For clock speed optimizations look at the 3–5 slowest paths listed in the timing report. Your slowest paths might be different from what you predicted from your dataflow diagram. Try to understand the cause of the difference (e.g., control circuitry), then look for simplifications and optimizations that will decrease the delay. Retiming and state-encoding are two techniques that can often be used to decrease delay.

- Once your optimizations begin to change your area and clock speed by less than 10%, optimizations often have unpredictable effects on area and clock speed. For example, combining two separate optimizations, each of which helps your design, might hurt your design. At this point, it is usually wise to transition to peephole optimizations.

3.7 Logic Simulation

Synthesize your design and then perform logic simulation using the following commands:

```
% uw-synth --logic kirsch.uwp
% uw-sim --logic kirsch.uwp
```

> **Note:** *For logic simulation, your kirsch core needs to drive all of the output signals in the* `kirsch` *entity. If you do not drive an output, Precision RTL will optimize it away and the output port will not appear in* `uw_tmp/kirsch_logic.vhd`*. The missing port will cause logic simulation to fail.*

- Logic simulation can be very time consuming due to the large number of pixels to be processed. Modify the image size in the package, so that instead of sending $256{\times}256$ pixels to your code, it sends a small portion of data (e.g. $16{\times}16$)

- If you have bugs in logic simulation, read the handout on debugging in timing simulation.

3.8 Peephole Optimizations

- Minimize the number of flip-flops that are reset. For example, usually there is no need to reset datapath registers.

- Use the minimum width of data that is necessary to prevent overflow. Related to this, avoid using the type `signed` unless you need negative numbers.

- When comparing a signal to a value, it may not be necessary to look at the full width of the signal. For example, if the value being compared to was 64 and the signal width was 8 bits, only bits 6 and 7 need to be examined to determine if the signal is greater than or equal to 64.

- You are probably wise to stop your optimizations at the point where your optimizations are improving the optimality by 2–3%.

3.9 Implement on FPGA

1. To create a design for the FPGA board, you need to include the UART and seven-segment display, which are in `kirsch_top`:

   ```
   % uw-synth --chip kirsch_top.uwp
   ```

   The `uw-synth` script should generate a file called `kirsch_top.sof`. **On a Nexus computer that is connected to an FPGA board:** Download this `.sof` file to the board:

   ```
   % uw-dnld kirsch_top.uwp
   ```

2. Run `ece327-run-proj` on your PC that is connected to the FPGA board, select a test image. The program will send the data to the board. The result that your circuit generates will be sent back to PC and be displayed on the monitor. The program `ece327-run-proj` will also display the percentage of errors that your design result contains. During the communication between PC and FPGA:

   - The seven segment will display the row count of image being sent to FPGA that will be displayed on numeric digits.
   - LEDG7 and LEDG6 will display the mode.

3. Note: `ece327-run-proj` has several flags that can be used to speed up the iterative process of download-run-debug. Use the `-h` flag to see the list of flags.

4 Deliverables

4.1 Group Registration

Register your group on Coursebook.

4.2 Dataflow Diagram

See Section 3.3 for information on the input and output, design process, and analysis for your dataflow diagram.

It is not expected that your design will be completely optimized at this point. In your final report, you will have an opportunity to discuss how you improved and optimized the dataflow diagram to arrive at your final design.

The dataflow diagram shall list the resources used (adders, subtracters, comparators, and registers). The dataflow diagram shall state the latency and throughput. The dataflow diagram shall include estimates for the total area, clock period, and optimality, with *brief* explanations of and/or equations for how the calculations were done.

No explanatory text is needed to show how you arrived at your final dataflow diagrams and no extra marks will be given for this. No fancy cover page is needed. The report *must* be a PDF file. Hand-drawn images are fine. Marks will be deducted for images that are poor quality, blurry, too-dark, *etc.*.

Submit with `ece327-submit-dfd` *dfd-name.pdf*

4.3 Design Report

Maximum: 3 pages, no cover page (just put project members and UWuserids), minimum 11pt font.

> **Note:** *We will only mark the first 3 pages!*

Your audience is familiar with the project description document and the content of ECE-327. Don't waste space repeating requirements from the project description or describing standard concepts from the course (e.g. pipelining).

Good diagrams are very helpful. Pick the right level of detail, layout the diagram carefully, describe the diagram in the text.

Design reports are to be included as a PDF file named `report.pdf` in the project directory.

> **Note:** *Your report* must *be in PDF and **must** be named* `report.pdf`

> **Note:** *Most of the report can be written before beginning the final optimizations. Most groups have the report done, except for their final optimality calculation, before beginning their final optimizations. This allows them to submit their report as soon as they are satisfied with their optimality score.*

The report should address design goals and strategy, performance and optimization, and validation.

Design Goals and Strategy Discuss the design goals for your design, and the strategies and techniques used to achieve them. Discuss any major revisions to your initial design and why such revisions were necessary. Present a high-level description of your optimized design using one or more dataflow diagrams, block diagrams, or state machines.

Performance Results versus Projections Include performance estimates made at the outset of your design and compare them to the results achieved. Describe the optimizations that you used to reduce the area and increase the performance of your design. Describe design changes that you thought would optimize your design, but in fact hurt either area or performance. Explain why you think that these "optimizations" were unsuccessful.

Include a summary of the area of your major components (look in the area report files), the maximum clock speed of the design, the critical path (look in the timing report files), latency, and throughput of your design.

Verification Plan and Test Cases Summarize the verification plan and test cases used to verify the behaviour of your design and comment on their effectiveness as well as any *interesting* bugs found.

4.4 Implementation

You may submit your design as many times as you want without penalty. Each submission will replace the previous design and **only the final submission will be marked**. To submit your design, enter the following command **from the directory containing your design**:

```
% ece327-submit-proj
```

Unless a message is displayed indicating that the submission has been aborted, your submission has been sent and you will receive an email containing your submission results. It may take up to 30 minutes for this email to be sent as the submission first needs to be processed and tested.

When your submission is sent, a simple "Dead or Alive" test will be run to ensure that basic functionality is present. An email will be sent to the submitter indicating whether or not the submission was successful. If there are warnings or errors, you should attempt to fix them and resubmit to ensure that your design can be properly processed for full functionality testing.

Do not rely on the "Dead or Alive" test as the only testing mechanism for your design. The "Dead or Alive" test simply runs the testbench for 1 ms and checks whether `o_valid='1'` happens.

4.5 Demo

Your group will demonstrate the design on the FPGA board to a TA or an instructor and will answer questions about the design and verification process. The demonstration will last approximately 30 minutes. It is important for everyone in the group to be present and to participate.

At the demo, we will give you instructions for how to copy the .sof file that we synthesized from your project submission. Then, you will demo several example images and answer questions about your project.

You'll be marked on quality, clarity, and conciseness of answers; and on group participation. Don't memorize prepared answers! If it appears that a group is just regurgitating prepared responses, we will change the questions in the middle of the demo.

5 Marking

5.1 Functional Testing

Designs will be tested both in a demo on the FPGA boards and using an automated testbench hooked up to the entity description of your design.

The design is expected to correctly detect all the edges of input images and exhibit correct output behaviour on both functional simulation, timing simulation and on the FPGA board. For full marks, designs shall work correctly on the FPGA board as well as with `uw-sim --logic --timing`.

A *Functionality Score* will be used to quantify the correctness of your design. This score will range from 0 (no functionality) to 1000 (perfect functionality). For calculating *Functionality Score*, we will run a series of tests to evaluate common functionality and corner cases, then scale the mark as shown below:

51

Scaling Factor	Type of simulation
1000	we will first try timing simulation (`uw-sim --logic --timing`) of the synthe-sized design.
800	if that fails, we'll try the synthesized design with zero-delay simulation (`uw-sim --logic`).
600	if that fails, we'll try the original source code (`uw-sim`).

5.2 Optimality Testing

Optimality will be evaluated based on *logic synthesis* results for a Stratix FPGA for area and clock speed, and behavioural simulation for latency.

5.3 Performance and Optimality Calculation

The optimality of each design will be evaluated based on functionality score, performance, and area (FPGA cell count). Performance will be measured by clock frequency in megahertz as reported by `uw-synth --opt`. For area, we count the number of FPGA cells that are used. This is the greater of either the number of flip-flops that are used or the number of 4:1 combinational lookup tables (or PLAs) that are used. Systems with an excessive latency (more than 8 clock cycles) will be penalized as shown below:

If $Latency \leq 8$ then

$$Optimality = Functionality \times \frac{ClockSpeed}{Area} \qquad (9.1)$$

If $Latency > 8$ then

$$Optimality = Functionality \times \frac{ClockSpeed}{Area} \times e^{\frac{-(Latency-8)}{20}} \qquad (9.2)$$

Note: *Assuming perfect functionality, if you achieve an optimality score of 600, you are guaranteed to obtain at least 80% of the optimality mark.*

5.4   Marking Scheme

Dataflow diagram
    •Clarity of drawing
    •Optimality of design
	5%

Submission
    •Correct signal, entity, file, and directory names
    •Correct I/O protocol
	5%

Optimality (see Equation 9.1) •High speed, small area, functionality	45%
Design Report •Clearness, completeness, conciseness, analysis	15%

Demo
  Test image functionality   15%
  Discussion about your design and design process   15%
	30%

5.5   Late Penalties

For the dataflow diagram, late submissions will receive a mark of 0 for the dataflow diagram.

To mimic the effects of tradeoffs between time-to-market and optimality, there is a regular deadline and an extended deadline that is one week after the regular deadline.

Projects submitted between the *regular deadline* and *extended deadline* will receive a multiplicative penalty of 0.1% per hour. Projects submitted at the extended deadline will receive a multiplicative penalty of 16.8%. Projects submitted after the extended deadline will be penalized with a multiplicative penalty of 16.8% plus 1% per hour for each hour after the extended deadline. Penalties will be calculated at hourly increments.

Example: a group that submits their design 2 days and 14 hours after the *regular* deadline will receive a penalty of: $(2 \times 24 + 14) \times 0.1 = 6.2\%$. Thus, if they earn a pre-penalty mark of 85, their actual mark will be $85 \times (100\% - 6.2\%) = 80$.

Example: a group that submits their design 1 day 4 hours after the *extended* deadline will receive a penalty of: $16.8\% + (1 \times 24 + 4) \times 1\% = 44.8\%$. Thus, if they earn a pre-penalty mark of 85, their actual mark will be $85 \times (100\% - 44.8\%) = 47$.

A   Statistics for Optimality Measurements

A.1   Altera Stratix IV

The data below was generated in March 2017 using Precision RTL 2016.1.1.28 for an Altera Stratix IV EP4SGX70HF35M at speed grade 3.

The given delay is for a circuit that contains registered inputs and outputs with a combinational functional unit. The delays given below can be summed to estimate the delay through multiple functional units, because the wiring delay encountered with multiple functional units will compensate for the the clock-to-Q and setup times of the registers incorporated into the delay value for a single functional unit.

Memory has a negligible delay on its inputs and a delay of approximately 1 ns on its outputs. (Functionally, memory has combinational inputs and registered outputs. But, from a timing perspective, memory appears to have registered inputs and combinational outputs.)

a + b

| width | no mux | | | mux on 1 input | | | mux on 2 inputs | | |
	LUTs	delay total	incr	LUTs	delay total	incr	LUTs	delay total	incr
1	1	0.62	0.62	1	0.76	0.76	1	0.86	0.86
2	2	1.05	0.43	2	1.15	0.39	4	1.45	0.59
3	3	1.22	0.17	6	1.45	0.30	7	1.76	0.31
4	5	1.57	0.35	7	1.82	0.37	10	1.97	0.21
5	5	1.51	-0.06	5	2.18	0.36	10	2.18	0.21
6	6	1.60	0.09	6	2.28	0.10	12	2.28	0.10
7	7	1.69	0.09	7	2.37	0.09	14	2.37	0.09
8	8	1.77	0.08	8	2.46	0.09	16	2.46	0.09
9	9	1.86	0.09	9	2.55	0.09	18	2.55	0.09
10	10	1.95	0.09	10	2.63	0.08	20	2.63	0.08
11	11	2.03	0.08	11	2.72	0.09	22	2.72	0.09
12	12	2.12	0.09	12	2.81	0.09	24	2.81	0.09
13	13	2.21	0.09	13	2.90	0.09	26	2.90	0.09
14	14	2.30	0.09	14	2.99	0.09	28	2.99	0.09
15	15	2.38	0.08	15	3.08	0.09	30	3.08	0.09
16	16	2.47	0.09	16	3.17	0.09	32	3.17	0.09

a - b

width	no mux			mux on 1 input			mux on 2 inputs		
	LUTs	total	incr	LUTs	total	incr	LUTs	total	incr
1	1	0.62	0.62	1	0.76	0.76	1	0.86	0.86
2	2	1.05	0.43	2	1.15	0.39	4	1.45	0.59
3	3	1.22	0.17	6	1.45	0.30	7	1.76	0.31
4	5	1.57	0.35	7	1.82	0.37	10	1.97	0.21
5	6	1.98	0.41	9	2.02	0.20	14	2.57	0.60
6	8	2.05	0.07	12	2.52	0.50	17	2.78	0.21
7	9	2.79	0.74	15	2.95	0.43	21	3.39	0.61
8	11	2.86	0.07	18	3.63	0.68	25	3.69	0.30
9	9	1.97	-0.89	9	2.54	-1.09	18	2.57	-1.12
10	10	2.06	0.09	10	2.62	0.08	20	2.66	0.09
11	11	2.15	0.09	11	2.71	0.09	22	2.75	0.09
12	12	2.23	0.08	12	2.80	0.09	24	2.84	0.09
13	13	2.32	0.09	13	2.89	0.09	26	2.93	0.09
14	14	2.41	0.09	14	2.98	0.09	28	3.02	0.09
15	15	2.49	0.08	15	3.07	0.09	30	3.11	0.09
16	16	2.58	0.09	16	3.16	0.09	32	3.20	0.09

a > b, a ≥ b

width	no mux			mux on 1 input			mux on 2 inputs		
	LUTs	total	incr	LUTs	total	incr	LUTs	total	incr
1	1	0.62	0.62	1	0.76	0.76	1	0.86	0.86
2	1	0.76	0.14	1	0.86	0.10	3	1.34	0.48
3	1	0.86	0.10	4	1.50	0.64	6	1.61	0.27
4	3	1.60	0.74	4	1.52	0.02	8	1.70	0.09
5	2	1.30	-0.30	5	1.60	0.08	11	2.17	0.47
6	3	1.11	-0.19	7	2.08	0.48	13	2.10	-0.07
7	3	1.75	0.64	8	2.01	-0.07	15	2.15	0.05
8	4	1.50	-0.25	10	2.15	0.14	18	2.30	0.15
9	5	1.79	0.29	10	1.80	-0.35	22	2.30	0.00
10	7	1.60	-0.19	11	1.90	0.10	23	2.45	0.15
11	9	1.80	0.20	14	2.35	0.45	26	2.45	0.00
12	7	1.51	-0.29	15	3.00	0.65	27	3.00	0.55
13	7	2.10	0.59	16	2.91	-0.09	29	3.05	0.05
14	8	1.90	-0.20	17	2.95	0.04	31	3.29	0.24
15	9	1.71	-0.19	18	3.15	0.20	34	3.45	0.16
16	11	2.54	0.83	20	3.36	0.21	36	3.50	0.05

reg

no mux		1 2:1 mux		3:1 mux (2 2:1 muxes)		4:1 mux (3 2:1 muxes)	
LUTs	delay	LUTs	delay	LUTS	delay	LUTs	delay
0	0.31	1	0.76	1	0.85	1	0.86