Final Project: Scrolling Message Display Board

Objective: To design a **S**crolling **M**essage **D**isplay **B**oard (**SMDB**) using VHDL. Students will apply their digital design knowledge and Finite State Machine (FSM) principles to implement hardware components and circuitry required of an **A**pplication-**S**pecific **I**nstruction Set **P**rocessor (**ASIP**). The ASIP will be used to realize the SMDB design, displaying custom messages and ASCII-type art using the DE2-115 FPGA board and its peripherals. The processor will support four built-in programs which display scrolling-type content across a HEX display at various speeds (slow, medium, fast, and super fast) as selected by the user.

1) Introduction and Background:

Scrolling Message Display Boards (***SMDB***), as seen to the left, are a popular way to advertise businesses and promotions to customers. The goal of SMDBs are to advertise using words and messages scrolling across a display board, where the message may flash or remain still. Caricatures and/or ASCII-type art may also be used to further animate the messages. Depending on the intricacies and programmability of an SMDB, purchase prices may range anywhere from $200 to $600 CAD.

Although we will not be building a dedicated circuit (i.e. an ***ASIC*** - ***A****pplication* ***S****pecific* ***I****ntegrated* ***C****ircuit*) for our final project, FPGAs are often used to *prototype* chips. In contrast to FPGAs, ASICs provide high performance at the cost of design flexibility. Therefore the DE2-115 FPGA board will help us prototype and determine the strategies needed to realize a SMDB hardware system with design flexibility.

An **Application-Specific Instruction Set Processor (ASIP)** is used to provide a trade-off between a general-purpose CPU (consider the various software applications that can be executed on a CPU) and the performance benefits of an ASIC. ASIPs are similar in principle to the accelerator cores we cover in Lecture 17, however ASIPs require a custom instruction set by which we can process and achieve our desired system behaviour for specific applications, in contrast to ASICs which is a hardware system designed for one specific application.

In this project, you will encode an *instruction set*[1] used by your ASIP. The instructions will be derived based on the instructions needed by the system to execute SMDB programs. We will hard code programs into the ASIP hardware. The ASIP's control unit will decode one instruction per cycle, and generate the appropriate signals for the datapath to execute and display messages/art on the FPGA board peripherals.

2) SMDB Functional Overview:

The SMDB will be designed to support four main programs: 1) Scrolling custom message (a custom message of your choice), 2) Scrolling snake left, 3) Scrolling snake right, and 4) Fly-in-a-box GIF. Three to four switches on the dev board will be used to select a program. When the switches are all off (0..00 considering active-high logic), the ASIP does not execute a program and the HEX displays remain off. When the program switches are activated, specifying a program, the program will continuously loop until either

[1] ***Instruction Set*** *– a complete set of all instructions that are recognized and executed by your processor in machine code. EX: add = 001, sub = 010, etc*

a) the switches are flipped to the off position (in the case of programs 1-3), b) the "pause" button is pressed, where the program will pause and thereafter resume once the pause button is lifted, or c) a "stop program" KEY is pressed, only in the case of Program 4, which will terminate the program.

The SMDB will use all HEX displays on the DE2-115 dev board, HEX7-HEX0, to display its scrolling messages or ASCII-art (in our case, the GIF) according to the program selected by the user.

Another set of two switches will dictate the speed of the scrolling text/art across the HEX displays. A user may change the speed of the scrolling text at any time (before, during, or after) the program is executing.

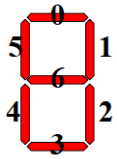
i) Program Details:

Program 0) IDLE – when the program switches are all off, no program executes and no HEX displays are active (aside from the exception of Program 4, see below).

Program 1) Scrolling custom message – select a custom message of your choice, which will display and scroll from right-to-left across the HEX(0-7) displays once then terminates. If your message contains spaces, consider that this "blank" character scrolls along with the rest of your message. The letters each take a different position on the hex display per cycle, until the last character is seen on HEX7. The program will thereafter terminate and return control to the *scheduler*. Marks will be assigned here for creativity.

Program 2) Scrolling snake left - An ASCII art-like snake, as shown on the left, scrolls from right-to-left across the HEX(0-7) displays. The head (7) will display on HEX0 in the first cycle. On the next cycle, the head will appear on (or "scroll left to") HEX1, with the first torso piece (П) displaying on HEX0 etc. This process continues until the tail (L) is last seen on HEX7. The program thereafter terminates and returns control to the *scheduler*.

Program 3) Scrolling snake right - An ASCII art-like snake, as shown on the left, scrolls from left-to-right across the HEX(7-0) displays. The head (backwards 7 or 'r') will display on HEX7 in the first cycle. On the next cycle the head will appear on (or "scroll right to") HEX6, with the first torso piece (П) displaying on HEX7 etc. This process continues until the tail (backwards L in this case) is seen last on HEX0. The program will thereafter terminate and return control to the *scheduler*.



Program 4) Fly-in-a-Box GIF – an imaginary two-grid box exists across two HEX displays, *HEX0 and HEX1*, that encloses a GIF-like 1-LED (1-bit) fly trying to find its way out of the box.

The 1-bit LED fly will traverse the following route on each HEX display:

Cycle	1	2	3	4	5	6	7	8	**9**
Lit LED	6	5	0	1	6	4	3	2	6

The fly will first cycle through the HEX's LEDs according to the table provided above, lighting the appropriate LED on HEX0 for **9** cycles. Once the fly has "tried to escape from" HEX0 for the 9 cycles, it will go to HEX1 and follow the same 9 cycle pattern. Once the 9 cycles are completed, the fly will go back to HEX0 and repeat the process. This process continues until a KEY is pressed by the user (your design choice), indicating that the "GIF" should be stopped. Assuming that the program switches are also set to zero or another program, control returns to the scheduler which exits Program 4 and initiates the proper action for the next scenario.

If an invalid program is selected by the user that *is not* Program 0 – 4, the ASIP will generate an error message "**ERR**" (or a similar message, ex: "eek" or "bruh") on the HEX displays **that will flash in place** while *emitting a frequency/tone from a set of headphones* connected to line-out, until the switches are set to a valid program option (0-4). *Message scrolling is not necessary*. You may select any set of HEX outputs to display the error message.

If a valid program *is* selected: a counter called *Program Execution Counter (PEC)* increments every time a **valid** program is executed. Its value is displayed in binary on the green LEDs. Since Program4 continues to loop its execution until the stop button is pressed, the PEC must increment each time the fly has completed one route pattern across HEX0 and HEX1 (cycle 1-9 x 2). The PEC is only reset back to 0 in the case of a hard reset or the counter saturates to 0.

If a "pause" button is pressed while program 1 – 4 is executing, the message scrolling across the display will be paused until the button is released. The same applies for the invalid/ERR program which will stop flashing and emitting a frequency from the headphones until the pause button is released.

ii) Designing a Custom Instruction Set Architecture (ISA) for your ASIP:

Based on the program descriptions given above, consider each program individually, and the instructions that are required to fulfil the program's behaviour on every cycle. How many instructions/clock cycles, will you need to fully execute the given program and display the required message/symbols? What behaviour should a given instruction fulfil in each clock cycle to properly execute?

Let's take an example from Program 2. The first instruction executed must display a '7' on HEX0, with the rest of the displays OFF during that one cycle. Similar to what we learned in Lecture 11 with the SOS machine, we can assign any encoding to an unassigned instruction, for instance the machine code "0001". This encoding's datawidth depends on the number of bits we require to represent **all our instructions** supported by the ASIP.

Continuing, if the first instruction of Program1 reads "0001", when executed by the ASIP, a '7' will display on HEX0. The second instruction of Program2, when executed, needs to display a '7' on HEX1 and Π simultaneously on HEX0 with the rest of the HEX displays OFF. We can encode this as "0010" for instance, or any other binary number that we have not yet assigned to our ISA, given that the functionality is not redundant with other instructions. Complete this encoding exercise for all programs and their respective instructions using the following exercise.

Exercise: You have been provided with an ***ISA Table*** in the Appendix that will need to be filled out and submitted with this project. Design a custom instruction set for your ASIP and use this table to guide you and complete your ISA design. Decide how many instructions you require per program, specify what each instruction must accomplish considering a given HEX_i display, and assign the instruction with an encoding. You may find that some instructions overlap throughout different programs, for instance the "L" used in the snake Program 2 and perhaps your custom message that begins with the letter "L"**. In this case, ensure your ISA does not possess redundant instructions.**

Once you have completed this exercise, your ISA will be fully encoded with a finite instruction count. Use this count to determine the number of bits required to represent your instructions and update your table accordingly. It is recommended that you use generic VHDL parameters throughout the project so that your design is flexible and should you require extra bits while adjusting your programs.

3) ASIP/SMDB Structural Overview: The structural diagram is presented in Fig. 1.

1) Control Unit - when a program has been selected, the control unit accepts instructions from the scheduler and generates data to be displayed on the HEX.

2) Datapath – interconnects the scheduler and Program Execution Counter (PEC).

3) Scheduler – i) monitors the program switch inputs and schedules a program for execution non-pre-emptively unless a reset is asserted. ii) controls instruction execution for the specified program, which is sent to the control unit. iii) Handles program termination and returns to the idle state until the next program is selected. If a reset is asserted, then the schedule returns to the idle state.

4) Prescale – varies the clock signal speed, and in turn the scrolling speed seen across the HEX displays.

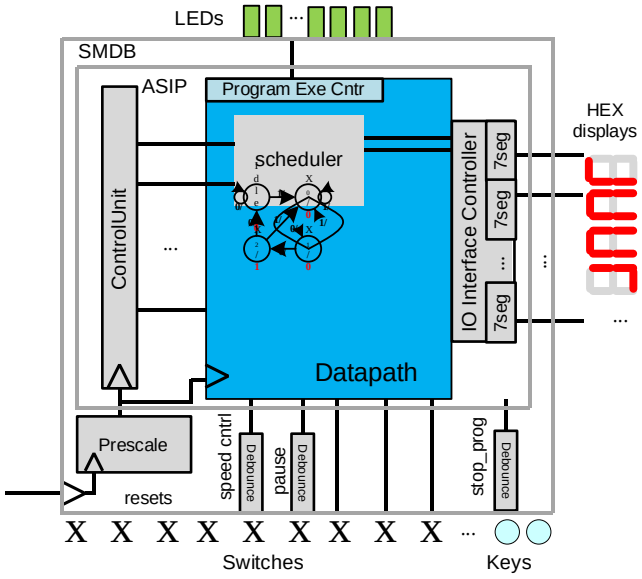


Fig. 1: Structural Diagram (partial) of Scrolling Message Display Board (SMDB)

5) Debouncer – used to debounce the speed-select switches, pause and stop program KEY. **The debouncer circuit will be provided to you**. Understand the circuit and instantiate it into your ASIP design.

6) Program Execution Counter (PEC) – counts the number of valid programs executed by the ASIP, and displays the binary equivalent on the Green LEDs, until a ***hard*** reset is asserted.

7) IO Controller – Manages and instantiates the 8 Custom7Segment converters required by the SMDB.

8) Custom7Seg (7Seg) – A custom 7Segment converter that can display all the characters required of your programs.

9) Application Specific Instruction set Processor (ASIP) –structural definition that connects the Control + Datapath + IO Controller

10) Scrolling Message Display Board (SMDB) system is the ***top-level*** design that i) instantiates and connects the ASIP, debounce and prescaler circuits, and ii) interfaces the components to their appropriate FPGA pins and peripherals.

4) SMDB Component and ASIP Specifications:

A) Scheduler – *Outputs a series of instruction based on the program selected by the user*

Inputs: **control** clk, rst, hard_rst, stop_prog (1bit) **data** program *(SW-bit input, sent from dev board switches)*
Outputs: **data** inst_out (n-bit output, where n= ISA datawidth) **control** pce (1-bit)
Main Scheduler States: idle, **running** (any other state if required)

The scheduler is a Finite State Machine (FSM) that manages user input from the switches, specifying the desired *program* for execution. Assuming no other program is currently executing, the scheduler passes control to another second FSM that manages program execution. Specifically, the second FSM is responsible for outputting one instruction per cycle from the selected program. The exact states required are up to you the designer. Once the program has completed, control is returned to the main scheduler FSM. Note that a program's execution is non-preemptive: a program's execution can not be interrupted by another program input by the user (the FSM will simply ignore the request. The new request does not need to be stored). If the switches are left to select the same program, the program will execute once again.

To output 1 instruction/cycle (*inst_out*) of the selected program, the second scheduler must have access to a library package (definitions_package.vhd) containing the hard coded programs. See subsection F) below for more information.

The *stop_prog* input is used to terminate Program 4's Fly-in-a-box execution. The output *pce* is pulsed (asserted only for 1 cycle) when a valid program has been executed, so that the datapath's Program Execution Counter may be incremented.

B) Custom7Seg (C7S) – *A custom Seven-Segment converter used to display content on the HEX displays*

Inputs: **data** D (**c7s**-bits, based on your Custom7Seg Table) **Outputs:** **data** Y (7-bits)

Exercise: Fill in the ***Custom7Seg (C7S) Table*** provided in the Appendix after you have completed the ISA table exercise. Use the C7S Table to assist you in to determining the number of bits required to encode all the characters required by your programs for display on the HEX. In turn, use this to determine the datawidth of C7S's input *D*.

C7S is a purely combinational circuit that accepts an n-bit input D (as derived in the exercise above), and outputs a 7bit string Y to illuminate the appropriate LEDs on a HEX display. All input entries are encoded by you the designer, except for input "00..0" which should not illuminate any LEDs. Recall that the DE2-115 HEX displays are active-low logic.

C) ControlUnit – *generates (control) data for the datapath based on the instruction input*

Inputs: **control** clk, rst, hard_rst **data** inst *(n-bit input, n = ISA datawidth)*
Outputs: **data** toSeg (bits = c7s-bits*8 (custom data type representing the concatenated data for 8 HEX displays)

The Control Unit accepts one instruction (*inst*) per cycle, sent by the scheduler, and generates an output *toSeg* sent to the C7S units. toSeg is the concatenation of 8 C7S converter (input D) values based on the instruction (inst) input, to be converted and displayed on the HEX. Therefore toSeg must possess a

datawidth of c7s*8 bits. This must be implemented as a custom 2D array data type in the definitions_package. See point F) below for details on implementation specifics.

D) IO_controller – *Input/Output (IO) Controller that manages output to the HEX displays*

Inputs: **data** toSeg (c7s*8 bits custom data type)
Outputs: **data** to_hex (custom 7bit*8 datatype, to be redirected to each HEX display by SMDB top-level)
Components: custom7Seg

The IO Interface Controller (IO_controller) is a purely combinational circuit that instantiates eight custom7Segment (C7S) converters, one per HEX display of the DE2-115 board. The *toSeg* input, sent by the control unit, is spliced and redirected to the appropriate converter component. The outputs of the C7S converters are aggregated (concatenated) to form the *to_hex* output of the IO_controller. Again, a custom 2D array datatype must be used to form the to_hex output using the definitions_package.

Although this controller may seem redundant, the design is used to keep the datapath design simple, clean, and uncluttered from the 8 instantiations required by the converters.

E) Datapath – *interconnects components required for processing instructions (as directed by control unit)*

Inputs: **control** clk, rst, hard_rst, stop_prog **data** program *(signal sent from switches (SW) input)*
Outputs: **data** inst (n-bit from ISA Table derivation, sent to control-unit) pce (4-bit sent to ASIP)
Components: scheduler **Integrated Logic or Component (optional):** PCE

The datapath instantiates the scheduler and implements the Program Execution Counter (PCE) logic according to the descriptions provided in previous sections.

The PCE may be implemented as a component or integrated into the datapath as logic (i.e. a process). As discussed, the PCE keeps count of the valid programs executed by the ASIP, **with a maximum 15 program count**, which rolls back to zero once the maximum count has been reached; note that this is not equivalent to a reset. The output count, pce, is output, and redirected for display on the DE2-115's green LEDs.

Soft and Hard Resets

There are two types of resets supported by the ASIP: 1) (**soft**) reset and 2) **hard** reset. The soft reset will force all **FSMs to return to the idle** state and terminate any running program. In the case of a **soft reset**, the **Program Execution Counter** does not erase its contents.

In the case of a **hard** reset, all **memory will be wiped from the system**, and all FSMs return to their idle states. Therefore, a hard reset is equivalent to a factory reset.

The ASIP and SMDB will require two reset inputs, reset and hard_rst, both implemented as synchronous resets and interfaced to two of the SW[x] FPGA pins of your choice.

F) Package – *included in VHDL files requiring the package content. A VHDL "package" contains custom type declarations, implementations (if required) and the 4 program's instructions. The programs are a hard coded set of instructions in order of desired execution.*

Included in the **Appendix** of this project manual is a VHDL package file template, called *definitions_package.vhd*. This file will contain declarations and/or definition for custom data types, functions, constants (such as your programs) that are required by your logic. Use this file to define any other items necessary for your design, such as GENERIC type values for instance.

Consider a VHDL *package* to be very similar to a header file in C++: this file contains definitions, declarations of custom types, helper functions etc that may be frequently used in your project. Like header files, you must include the package at the top of your VHDL file where applicable using the syntax:

```
USE work.definitions_package.all;
```

In general, the VHDL files you create and include as components in your project are considered user defined and part of a default library called **work.** You have seen the "work" folder created multiple times in Modelsim. On the contrary, built-in VHDL library definitions we have been using are part of the **ieee *standard*** library (ieee.std_logic_1164.all, ieee.numeric_std.all which we use for "+" etc).

There is no need to **import** the work library as it is already contained in your "working" directory. You must solely include the "USE work.xxxxxx" syntax above in your VHDL file to include your definitions package. *Consider this package synonymous to the user-defined header files you create in C++ (ex: #include "helper_functions.h" =* ***USE work****.xxxxx) vs the compiler-defined headers included in your code (ex: #include <iostream> =* ***USE ieee****.xxxxx).*

This "package" file must be included in your project folder and declared in the appropriate VHDL files as hinted above, or any other component that encompasses your data types, constants etc.

Important Note: When **compiling** a design in **Modelsim that includes a package**, ensure you compile **package(s)** *first*, and then proceed to compile the other VHDL files. Change the compile order to match this requirement. If you do not follow this precedence, the design will not compile. **The package must be compiled first** so that the environment can link the definitions to dependent VHDL files.

You may also include the **ieee.logic_1164** and **ieee.numeric_std** libraries in the package definition to avoid inclusion redundancy at the top of your VHDL files. In this case, you must remember to include the definitions_package in every VHDL file you create in the project.

G) ASIP – *Integrates and makes the appropriate connections between the control unit and datapath*

Inputs: **control** clk, rst, hard_rst, stop_prog **data** program (n-bit redirected from switches)
Outputs: **data** to_hex (custom data type 7bits*8 hex displays), pec (4-bit)
Components: datapath, controlUnit, io_controller

The components above (A-F) all comprise the Application Specific Instruction-set Processor (ASIP) design. Once all components have been designed above, the datapath, ControlUnit and io_controller may be instantiated and connected in the ASIP's structural definition. Thereafter, the ASIP will be used as a component in the top-level design, SMDB. Accordingly, **this file should not contain any FPGA pins**. Use

the descriptions provided above to make the appropriate connections. Create a structural block diagram to help you instantiate and connect the components.

H) Prescale – *A clock divider used to slow down the FPGA's 50MHz clock, with a variety of speeds available as selected by the user using "mode". The resulting clock is used as the SMDB system's clock source*

Inputs: control clk, mode (2 bits)	**Outputs:** control clk_out (1bit)

As discussed in Lab08, your DE2-115 FPGA board possesses a 50MHz clock, generated by an onboard crystal oscillator. The output of the oscillator is connected to pin **CLOCK_50** (pin Y_2) which generates a clean clock signal. The problem with using this clock for displaying messages is that the clock is **too** fast. Consider that a 50MHz frequency translates to a **20ns** period. The *human eye requires at least a few milliseconds (ms)* to recognize an image. For this reason, we require a Prescale circuit, however this time we must implement varying speeds that can be selected by the user.

The prescaler's clk input will be mapped to the CLOCK_50 pin in the top-level file. The input "mode" will dictate the clock speed of the clock output "clk_out". Mode will be specified with two switches on the FPGA dev-board, selected by you the designer, and controlled by the user. The clk_out signal will act as the clock source delivered to all synchronous components in the ASIP.

The speed of the clock may change at any time: before, after, or during a program's execution, and the speed must also be instantaneously reflected on your scrolling display.

When mode is "00", this will be considered "normal" operating mode as determined by you, the designer. Thereafter three other speeds are required for implementation: slow, fast, very_fast. Use your best judgement to determine what is considered a slow, fast and very fast speed. To realize these operating modes, it is recommended that you create the prescaler circuit independently: use a temporary VHDL wrapper that is interfaced to LEDs as output, and the CLOCK_50 pin as input. Use the switches for mode input, and LEDs to gauge the clk_out speed from the prescaler.

I) Debouncer – a circuit that provides a clean output signal from the mechanical push buttons and switch inputs as they are pressed. At minimum, the debouncer circuit must be used for the prescaler's "mode" input, the pause and prog_stop KEY inputs. **This VHDL code is provided to you**. Ensure you read the code, understand the logic and instantiate the component in the top-level SMDB VHDL file.

J) SMDB – *top-level file integrating the debouncer circuits, prescaler, and ASIP, while interfacing all IOs to their appropriate FPGA pins.*

Inputs: control CLOCK_50, SW, KEY
Outputs: data HEX0 – HEX7, LEDG, LEDR
Components: asip, debouncer, prescale

The SMDB must:

i. instantiate the ASIP component and appropriately connect all its ports
ii. route the FPGA clock pin input, CLOCK_50, to the prescaler circuit, connecting the output clock source to the required synchronous components

iii. At minimum, debounce the reset, hard reset, pause, and stop program (prog_stop) to the switch and key pin inputs. The other inputs are optional for debouncing, but nevertheless it is always good practice to debounce all mechanical components in your digital logic circuits.
    - Assign a KEY to pause the program execution. In general try to avoid using KEY0 as this button has been destroyed by previous years always mapping designs to KEY0.

iv. Interface the rest of the FPGA pins to their appropriate ports. Recall that LEDG displays the program counter (pce). You may also use LEDR for debugging purposes.

v. **PROGRAM Switch Assignments** – add the last digit of your student number + last digit of your partner's student number. If the sum is:
    - ODD – use a **one-hot coding** to represent program input on the switches
    - EVEN – use the **log2** method to assign the minimum no. of switches for program input.
    - Consider that the switches in the 0 state (no program running) is encoded

You will be required to determine any specifications that have been purposely left out, such as the pause functionality, port/signal datawidths, exact data types. The rest of the implementation is up to you. Ensure you have proper justification for your design choices.

5) Suggestions for Component Integration

DO NOT design and build the SMDB and ASIP cores as one large circuit. Implement one component at a time, and thoroughly test each unit to ensure proper behaviour. Ensure you understand the protocol inherit of each component. You will need to use this timing behaviour to integrate the components and interface them appropriately in your design. Once the integrated components work, proceed to integrate the next component until you reach the top-level.

Here is a list of suggested steps and hints to ease SMDB system integration. Ensure you fill in the ISA and C7S tables in the Appendix before attempting any of the VHDL designs. It is important that you understand the datawidths and functionality required of your design before proceeding to code your VHDL.

A) Designing and debugging the custom7Seg (C7S)

i) Fill-in and complete the C7S Table in the Appendix of this document, according to the instructions you have determined necessary for your ISA to execute the four programs + error program. This will require you to also complete the ISA Table.

ii) Create a new project and complete the VHDL design for your custom7Seg converter using the C7S table.

iii) Create a VHDL wrapper that connects your input, D, to a set of switches (SW) on the DE2-115 board, and the converter's output Y to one of the HEX displays. Go through the table entries and ensure that each input works as intended, displaying the appropriate symbol or character to the HEX display.

B) Designing and debugging the Control Unit

i) Using your completed ISA table, create a new VHDL file, controlUnit.vhd and code your ARCHITECTURE as a large switch-type construct which lists each possible instruction scenario, and generates the output required for all HEX displays according to your ISA table. Consider the number of bits you require to represent all the instructions in your ISA.

ii) Create a VHDL wrapper which connects the control unit's output to the C7S inputs. Map the inputs and outputs of these two components to their appropriate FPGA pins. For instance, the control unit's instruction input, reset and hard_reset may be mapped to the DE2-115's switches. The control unit's toSeg output should be spliced and input to the appropriate C7S component instantiation, and the C7S outputs mapped to the necessary HEX pins. At this point, you may also decide not to create the custom data type toSeg, and simply have 8 outputs from the control unit, one for each HEX display.

For the clock's input you can either: i) Use a KEY as input, or ii) (better idea in the long run) use a clock divider/prescaler to slow down the FPGA pin input CLOCK_50, and use the output as the clock source. Feel free to use your prescaler from Lab06 or Lab08 as a clock source for this debugging exercise (you may need to increase the length of the counter).

Once your wrapper is created, synthesize, and program the FPGA, and use the switches to input instructions into the control unit. Verify that the output on the HEX displays matches your ISA table specifications.

C) Scheduler design and debugging

Create your programs either in the scheduler.vhd or the definitions_package.vhd. You will likely require a custom type(s) to realize each of your programs. Consider the programs as a 2D array of constants (Program1 is X elements (instructions) each of Y bits). Create your custom type(s) and use this to create your 4 programs.

The scheduler's design will require an incremental coding technique. Start with a basic template for a VHDL FSM design that implements two or more states: IDLE, RUNNING:

The IDLE state will transition to the RUNNING state when a program input is specified by the switches.

The RUNNING state will transition to the IDLE state when there are no remaining instructions left for execution.

The RUNNING state will trigger a second FSM process. The states implemented by the second FSM are up to you, however the FSM's main purpose is to execute/output all instructions of the selected program, and return control to the main FSM scheduler.

Once you believe you have the basic template for the scheduler, use a testbench to verify the behaviour of your scheduler. Incrementally add functionality and program support to verify that the scheduler functions as intended.

Thereafter, you may create a top-level VHDL wrapper that instantiates your C7S, control unit and scheduler components (with a prescaler) to verify that your programs execute on the HEX displays as intended.

D) Remaining Design

Once you have completed steps A)-C), aside from the IO controller and definitions_package.vhd, the remaining steps are mostly component instantiations, connections, and fulfilling all system features. It is a good idea to create a checklist of all the features that must be fulfilled, and use this to guarantee that your final design has met all the specified requirements. A testbench should be used for each step, besides

the top-level SMDB, to verify correct functionality. Ensure your VHDL is properly formatted, concise and well-written. Consider using loops, for generates whenever possible.

6) Presentation/Deliverables/Group Work **Due Monday, Dec 6th @ 11:59pm**

Demo: Please sign up with your partner for this project using **Canvas > People > Project**. Give your group a name. *The best group name will receive a +1 on their project*. If you do not wish to work with a partner, this will be considered during project grading. We will use your group name to book a demo time on **Tuesday, December 7th, 2021**. Both members MUST be present for the demo, else a mark of zero will be assigned to the demo portion of this project.

You and your partner must demo your SMDB design to the TA during the scheduled time and answer a series of questions. The demo should take 5-7 minutes max. Your preparation will also be considered in this demo mark. *If something in your design does not work properly, clearly state the blunder else you will be heavily penalized for the omission.*

Deliverables uploaded to Canvas as one Zip:

- **For those working in partners**: Include an **individual report**. Clearly outline in bullet points your contributions to this project with a brief description. Marks will be assigned accordingly to workload distribution. You and your partner will not be assigned the same marks for uneven work distribution.\
- Cover page signed by you and your lab partner (if applicable)
- ISA table
- C7S table
- A detailed top-level, structural diagram labelling all signals and component instantiations in your design (a detailed Fig. 1). You may also submit several diagrams of different components to provide the necessary details.
    - Ensure all diagrams are professional looking. The use of Visio or similar software is recommended.
- Quartus project. Ensure all your files and project settings, when extracted and opened, are fully functional and ready for the TA or instructor to simulate. Use “Clean Project” to remove unnecessary files.
    - You will be marked on the quality of your code: concise with proper formatting. Use comments so the TAs can follow your work. Any vague code will be marked as interpreted by your TA.
- A **“debug” folder** must be included, testbenches, waveforms etc used to verify your component and top-level designs. Ensure subfolders included are organized and well-labelled.
- A README file commenting on your submission organization, folders, and files so that the TA may navigate and find everything required of this project.

Mark Breakdown: Total [/90]
[/15] Demo + questions
[/50] VHDL – content, behaviour, structure, specifications met
[/20] Design, Development, and Verification – tables,
[/5] individual report

Appendix:

The next two pages contain the ISA and C7S tables. If you require more table entries, use a 2nd, 3rd copy of the table until you have completed the exercise. See last page for an example of how to fill in the table.

ISA Table after filling out the table, determine the no. of bits you require to encode all the instructions of your ISA

Instruction	HEX7	HEX6	HEX5	HEX4	HEX3	HEX2	HEX1	HEX0

Custom7Seg Display Table – after filling out the table, determine the no. of bits you require for input D

D	HEX Output	Comment(s)
0	1111111	LEDs all off 
1	1111000	left direction snake head aka 7 

VHDL Packages: definitions_package.vhd

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;


package definitions_package is
 -- Package Declaration Section (examples below)

CONSTANT N : INTEGER := 8;
 TYPE bus_width IS ARRAY (0 to 2) OF UNSIGNED(N-1 DOWNTO 0);

end package definitions_package;


package body definitions_package is
 --blank, include any implementations here, if necessary

end package body definitions_package;
```

To include this package in your design/component, write the following at the top of your vhd file:

```
USE work.definitions_package.all;
```

Thereafter, you may use any definition included in this work package within your code.

ISA Table Example:

Ex from pg. 3, Program2: Instruction 0 = all HEX off.
Instruction 1 = HEX1-7: all off and HEX0: 7

Instruction	HEX7	HEX6	HEX5	HEX4	HEX3	HEX2	HEX1	HEX0
0	--	--	--	--	--	--	--	--
1	7	--	--	--	--	--	--	--