Australian National University

ASSIGNMENT COVER SHEET

ANU College of Engineering, Computing and Cybernetics
Australian National University
Canberra ACT 0200 Australia
www.anu.edu.au

+61 2 6125 5254

Submission and assessment is anonymous where appropriate and possible.

This coversheet must be attached to the front of your assessment when submitted in hard copy. If you have elected to submit in hard copy rather than Turnitin, you must provide copies of all references included in the assessment item.

All assessment items submitted in hard copy are due at 5pm unless otherwise specified in the course outline.

Student 1 ID	U7556893		
Student 1 Name	Gia Minh Nguyen		
Student 2 ID	U7671528		
Student 2 Name	Huy Hoang Hoang		
Course Code	ENGN4213/6213		
Course Name	Digital Systems and Microprocessors		
Assignment Item	☒ FPGA Project Report ☐ Micro-controller Project Report		
Word Count	4386	Due Date	22 April 2024
Date Submitted	22 April 2024	Extension Granted	

I declare that this work:

☒ upholds the principles of academic integrity, as defined in the University Academic Integrity Rule;

☒ is original, except where collaboration (for example group work) has been authorised in writing by the course convener in the course outline and/or Wattle site;

☒ is produced for the purposes of this assessment task and has not been submitted for assessment in any other context, except where authorised in writing by the course convener;

☒ gives appropriate acknowledgement of the ideas, scholarship and intellectual property of others insofar as these have been used;

☒ in no part involves copying, cheating, collusion, fabrication, plagiarism or recycling.

Initials

For group assignments, each student must initial.

Gia Minh Nguyen (GM)

Huy Hoang Hoang

Australian National University

CONTRIBUTION STATEMENT

We, the undersigned members of ENGN4213/6213 FPGA Project Group No: 18, hereby state our main individual contributions.

- Created initial template.
- top.v
- access_control.v
- ssds_control.v
- morseDecoder.v
- morse_processing.v
- Report

Signature 1 with name

Gia Minh Nguyen
Date: 18 April 2024

- climate_control.v
- climate_fsm.v
- temperature_controller
- Fixed a critical bug in morse_processing.v
- Write test benches for climate control's sub-modules and PIN validation
- Report

Signature 2 with name

Huy Hoang Hoang
Date: 18 April 2024

This statement may be considered to assess fair contributions among group members and, if deemed necessary, regulate individual marking.

Australian National University

TABLE OF CONTENT

Australian National University

TABLE OF FIGURES

OVERALL ARCHITECTURE

1. User Interface

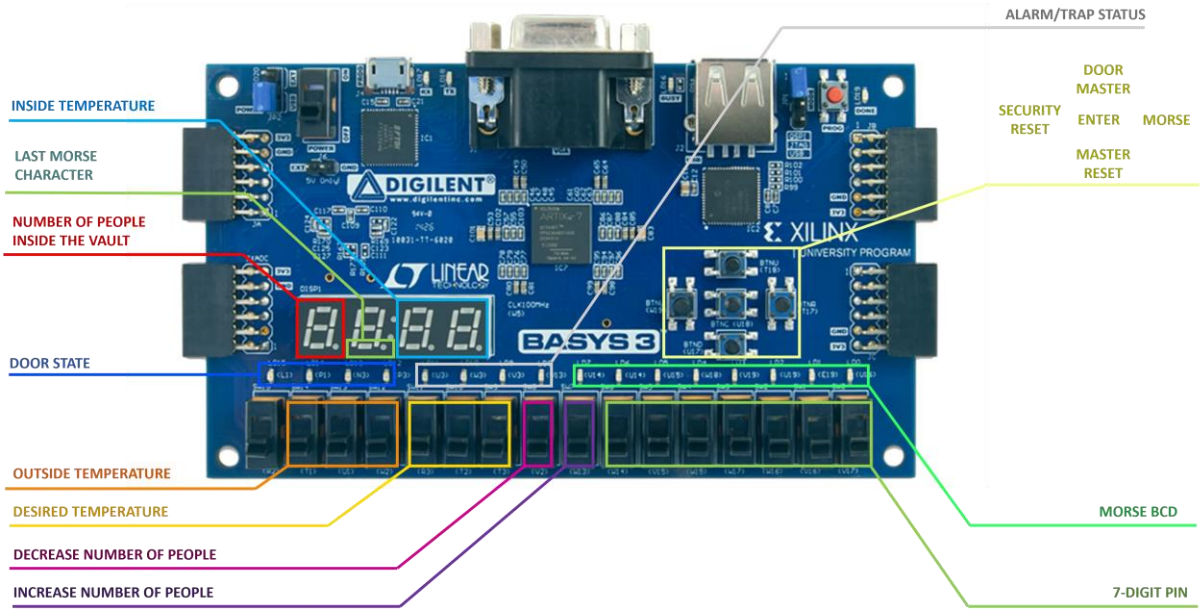

Figure 1: I/Os used in this project, and their assigned functions

1.1. Access Control

Upon powering up, the vault's door is closed (indicated by LEDs **LD15-LD12** lit up). To open the door, a valid 7-digit PIN must be entered via switches **SW0–SW6**. Confirm the entered password by pressing ENTER (**BTNC**). If the password is correct, the door opens, indicated by **LD15-LD12** turning off one by one. Otherwise, LEDs **LD8-LD11** will blink every two seconds, indicating that the ALARM has been set off. If the user fails to enter the correct password the second time or is unable to do so within 20 seconds after the ALARM set off, the door will enter the TRAP state, in which LEDs **LD8-LD11** will blink rapidly every 0.5 seconds.

Once the door has been completely opened (**LD15-LD12** are all turned off)**,** use **SW7** and **SW8** to increase and decrease the number of people inside the vault, which is reflected by the first seven-segment display (SSD). Changing the switches' state causes the number of people to change by 1. However, the number of people can never exceed 9 or fall under 0.

The door begins to close after it has been fully opened for 30 seconds. After the door has closed, the controller will check for the number of people inside the vault. If it is empty, the controller will switch to the idle state for entering. Otherwise, the controller will switch to the state that is waiting for exit, a 2-digit code must be entered in Morse code, which corresponds to a sequence of 10 Morse signals. In our implementation, the type of Morse signals ("dot" or "dash") is determined by how long the user holds the **MORSE** button (**BTNR**). If the **MORSE** button is held for 0.24 seconds or more, then the signal will be interpreted as a dash, otherwise, it is interpreted as a dot. The registered output for **MORSE** button presses is displayed on the second SSD. After receiving 10 Morse signals, the controller will immediately check for correctness and open the door if the pin is in the list of accepted sequences. If the entered sequence is not valid, there will be no behaviour and the user can retry as many times as they want.

When a digit has been entered (after every 5 Morse signals), a Binary-coded decimal (BCD) equivalent of the pin's digit will be displayed using **LD4-LD7** for the first digit, and **LD0-LD3** for the second digit, respectively. For example, if the entered pin is 72, **LD4-LD6** will light up to represent the 7, while **LD1** will light up to represent the 2.

A slight modification has been made to the extension: instead of only displaying the BCDs when the whole pin combination is entered and is correct, the LEDs to display the BCD pin digits now update immediately after enough Morse signal has been entered to form its corresponding decimal digit (updates every 5 signals). The reason is that it is more practical for the vault's staff to be informed of what digit the controller registered, so they can make adjustments to their Morse keying technique if the registered digit is not the staff's intention; meanwhile with the original specification, displaying the correctly entered pin is not useful as the staffs already know the accepted pins and should already have one in mind when they start keying in the Morse signals.

1.2. Climate Control

As the possible range of temperature is between 20 and 27 degrees, we only need 3 bits to represent them (for example, $000_2 = 0_{10}$ => 20º; $111_2 = 7_{10}$ => 27º). In our representation, we use two 3-switch sets to input temperature. Switches **SW12-SW14** are used to input the outside temperature, and switches **SW9-SW11** are used to input the desired temperature. The room temperature is regulated according to the number of people inside the vault and current state of the climate controller. Specifically, if there are more than 5 people inside the vault, the climate controller changes the vault temperature at the rate of $1^o/1s$ toward the desired temperature. Vice versa, the vault temperature is changed by 1^o every $0.5s$. However, in case the climate controller is off, the vault temperature will change toward the outside temperature instead, at the rate of $1^o/2s$.

1.3. Special Switches

SECURITY RESET (**BTNL**): Turn off the security alarm and exit from the trapped state should an invalid PIN be entered.

DOOR MASTER (BTNU): Open the vault's door at any state, no passwords are required.

MASTER CONTROL (BTND): At any time, reset the controller to the initial entering state (vault is vacated, door closed).

2. High-Level Architecture

There is a container module at the top level (**top.v**) that encapsulates other modules in our system. This top module serves several purposes: First, it pre-processes the necessary I/O inputs, thus eliminating the need for debouncing and SPOT at the individual modules level. Debouncers are needed for the **morse_key**, **people incrementing/decrementing** switches and the **ENTER** button as we don't want instantaneous repeated signals caused by the inputs' bounces. the *debounced enter* signal will then also be passed through a SPOT module which is critical, without it, the vault controller will have cycled through several states that waits for the **ENTER** input to be *HIGH*, because at the clock speed of 100 Mhz, a human's fastest press and release of a button would have taken several thousand clock cycles, while state is updated on every cycle. Preprocessing the other inputs are not necessary as **DOOR MASTER**, **SECURITY_RESET** and **MASTER CONTROL** set the next states to *IDLE_ENTER* and *ALARM_RESET* respectively, which their next state transition conditions has been carefully constrained to not be affected by those three buttons, hence it is not possible to skip a state even if the buttons bounce or are held longer than expected. The temperature switches also do not need preprocessing as they set absolute temperature values.

Secondly, it acts as a mediator that distributes I/O inputs to each sub-module and links the sub-modules to make one complete system. This approach localizes the major functions into their respective sub-modules, making the sources modular, and increasing readability and comprehensibility for the development or maintenance process.

There are three modules under **top**: access control (**access_control.v**), climate control (**climate_control.v**), and SSD control (**ssd_control.v**). Access control is responsible for vault entry and exit, while climate control adjusts vault temperature according to requirements. Though they work independently most of the time, there are vital signal lines that need to be shared between the sub-modules. (Refer to *Figure 2*) The number of people inside the vault needs to be passed from **access_control** to **climate_control** to determine the climate control status. All the outputs that are displayed via the SSDs will be passed to the **ssd_control** sub-module. **ssd_control** is the driver for the SSDs, hence it manages all the details such as

controlling anodes and cathodes based on the decimal/binary inputs, so the other modules do not need to worry about individual cathodes and their corresponding segment.

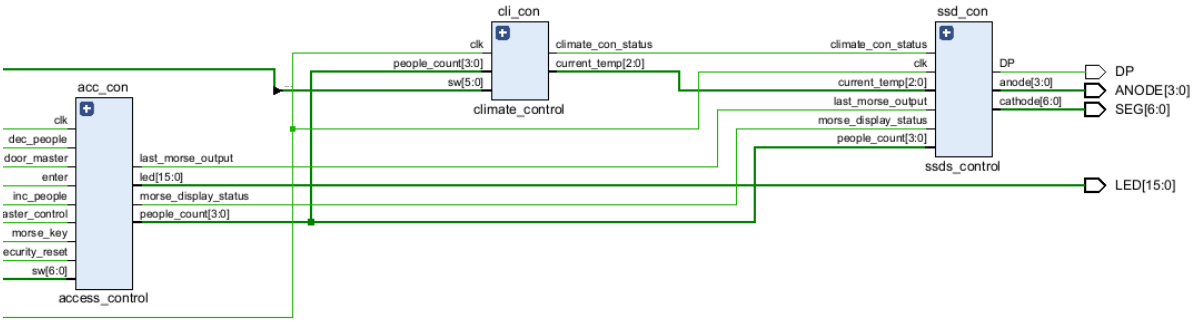

Figure 2: High-level architecture of the system

ACCESS CONTROL MODULE

The **access_control.v** module consists of several functions: *people incrementing and decrementing* (*people inc/dec*), *morse signal processing*, *morse decoder/translator*, and the *finite state machine* (*FSM*) for access control. As the functions suggested, the module control the entering, exiting operations of the vault's controller, as well as controlling the number of people inside the vault. Hence, the module has the access to all 15 LEDs, **SW0-SW8**, as well as the first two SSDs to display its output.

As the *people inc/dec* function and the *FSM* is only instantiated once in **access_control.v** and they share a lot of I/O between other functions, we decided to not abstract them into sub-modules. The decision also created a flow of reading in **access_control.v**, and eliminated the need for jumping back and forth between the parent and its sub-modules. To keep the code comprehensible and easy to navigate, the functions are neatly separated into distinct sections in the code and are indicated by header-styled comments (an example is shown in Figure 3).

```
//------------------------------------------------
//--    Morse signal processing and registering    --
//------------------------------------------------
```

Figure 3: Example of a header for a section of code used inside access_control.v

The *morse decoding* function and *morse processing* function are also only instantiated once, however they were still encapsulated into their respective sub-modules **morse_processing.v** and **morseDecoder.v**. The reason for **morse_processing** is that separating it makes testing for the morse function easier, as it is not affected by other components of the controller. **morseDecoder** was also separated as it is just a "dictionary-like" helper sub-module similar to **sevenSegmentDecoder**, and that also explains the same naming convention.

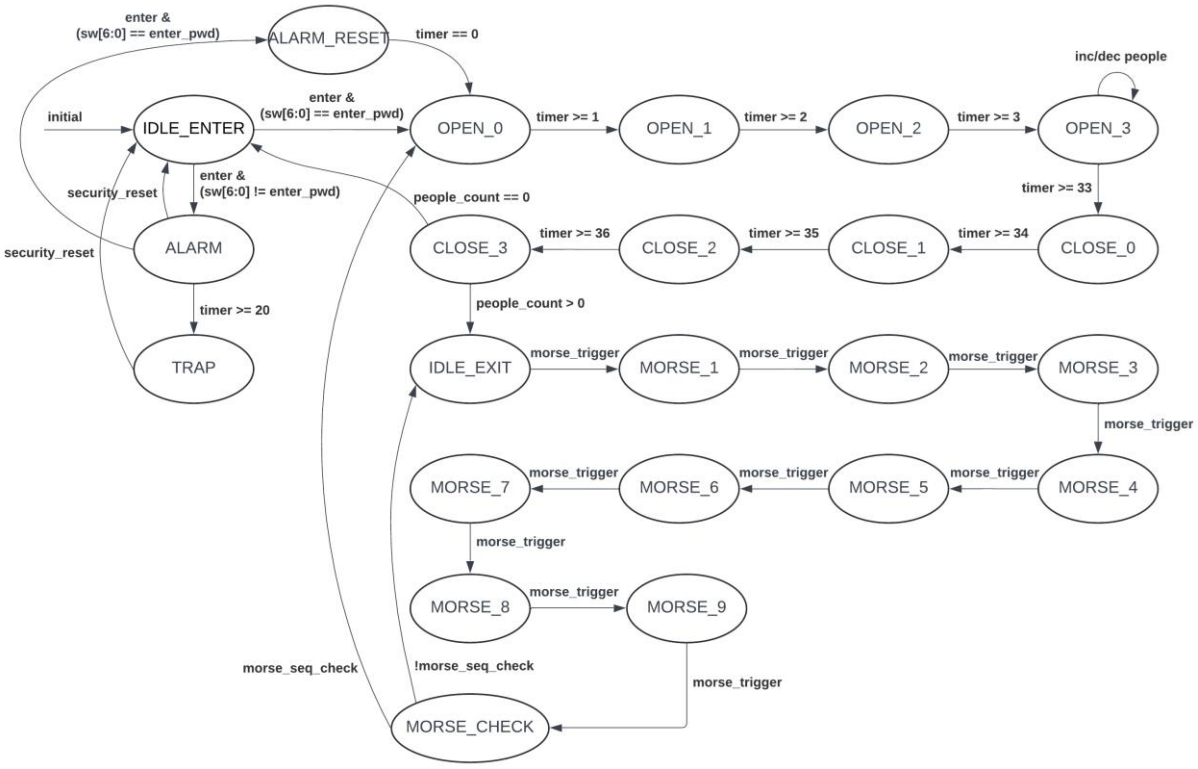

Figure 4: Diagram of access control's finite state machine.

1. Morse code processing

```
assign morse_seq_check = (input_morse_seq == {m0, m7}) |  // 07 ----- --...
                         (input_morse_seq == {m1, m0}) |  // 10 .---- -----
                         (input_morse_seq == {m2, m3}) |  // 23 ..--- ...--
                         (input_morse_seq == {m3, m4}) |  // 34 ...-- ....-
                         (input_morse_seq == {m5, m6}) |  // 56 ..... -....
                         (input_morse_seq == {m6, m7}) |  // 67 -.... --...
                         (input_morse_seq == {m7, m2}) |  // 72 --... ..---
                         (input_morse_seq == {m7, m8}) |  // 78 --... ---..
                         (input_morse_seq == {m3, m7}) |  // 37 ...-- --...
                         (input_morse_seq == {m4, m5}) |  // 45 ....- .....
                         (input_morse_seq == {m8, m9}) |  // 89 ---.. ----.
                         (input_morse_seq == {m9, m2});   // 92 ----. ..---
```

Figure 5: code to check the morse input sequence against accepted passwords

The section is denoted as “Morse signal processing and registering” in **access_control.v**. Since both morse and binary have two possible digits, we matched Morse’s *dash* to *b1* and Morse’s *dot* to *b0*. We decided that definition would be intuitive since *dot* is shorter than *dash*, which is mirrored to the fact that *b0* is less than *b1*. Based on that, we created a list of parameters “*m0*” (morse-0) to “*m9*” (morse-9) as binary representation of the morse code that represent the decimal number, to easily reuse them in the later codes. Using the same idea, the module *morseDecoder* is created to convert binaries that represent morse to binaries that represent decimal. It should also be noticed that all number digits are represented by exactly 5 morse signals, hence we know that the morse sequence for exiting should be expecting 10 values/morse signals. From that information, we can use a sequence with 10 values (*input_morse_seq*) to hold the current morse sequence that is being keyed in, and compare it to all of the accepted passwords which are encoded into their corresponding sequence. The comparisons are then linked together using “*OR*” operation (Figure 5) so the wire output HIGH whenever there is a pair of matching sequences. This wire can then be directly used as the condition for *MORSE_CHECK* to determine whether the input is correct. (Figure 6)

```
MORSE_CHECK: begin
    if(morse_seq_check) nextstate = OPEN_0;
    else                nextstate = IDLE_EXIT;
end
```

Figure 6: *MORSE_CHECK* section inside the case statement of next state logic block

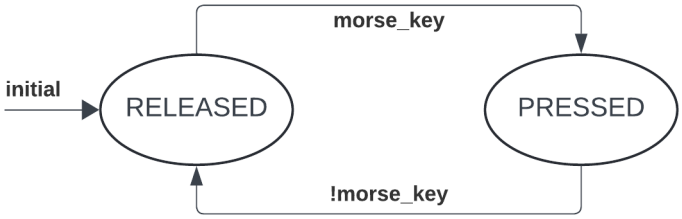

Figure 7: FSM of the morse processing module

To process the morse key requires a simple FSM itself (Figure 7) as it needs to be able to record the time pressed. Making this into a sub-module eliminate the need to create an extra 10 states in the main *FSM* to wait for the *morse_key* to be released. The way it works is that whenever the *morse_key* is pressed, a heartbeat clock with the rate of a beat every 0.24 seconds will be enabled, and a counter is incremented on the beat. Upon released, the counter will be compared, if it is larger than or equal to 1 (meaning the button has been held for more than or equal to 0.24 seconds), the *last_morse_output* will be updated to be dash (*1'b1*), else, it is assigned to be a dot (*1'b0*). After that, a SPOT signal *morse_trigger* is sent as a output to notify the parent modules that a new morse signal has just been entered. The counter and timer will then be reset and disabled.

The heartbeat of 0.24 seconds, when added with a 0.1 seconds debounce makes the morse dot's time limit ~0.34 seconds, which is not any particular values but it is definitely more than necessary for a concise button press (which to our testing average at ~0.18 seconds), and it is definitely less than a long button press (which according to the 1:3 timing convention in Morse would make a dash ~1.02 seconds).

2. Access control's FSM

The *FSM* is composed of 4 main components: state memory, next state logic, timer procedural, and output logic. For an explicit and good-style implementation, the components are separately presented in their respective always blocks.

There are 23 states in total, which can be encoded using 5 bits. The door opening and closing process is split into 8 states in total so that each state is responsible for turning on/off an LED, therefore displaying the LEDs rolling effect. There are 10 morse entering states corresponding to the sequence of 10 morse inputs, so that we can identify which index of the morse input sequence we are in at any point in time. Since we already listen for morse signal from *IDLE_EXIT*, we can stop at *MORSE_9* and *MORSE_CHECK* is used for checking the input sequence.

2.1. Timer procedural block

A lot of the states needed to be timed in seconds. Hence, the timer procedural block implements a *timer* which is essentially a counter that increments by 1 every second according to the beat generated by the heartbeat divided clock *CDHB*. Then, the next state logic block will be "listening" to this *timer* to decide on when to change to the next state. Several approaches have been considered for the timer like countdown vs elapse. It was decided that the timer should be counting up as there were difficulties in implementing countdown timer, including but not limited to efficiently setting the time duration between the existing states without adding extra states just to set the timer. An elapse timer will be enabled in every state that requires a timer to move to the next state as seen in *Figure 4*. In the other states, it will be disabled and reset to be ready for the next use. Hence an if-else block is implemented to independently reset the timer in the appropriate states which is determined by *timer_states*, a list of states that needs timer to be enabled, concatenated

together using "*OR*" operation. (Figure 8) This method eliminate the need to redundantly set and reset timer at individual states.

```
// timer procedural (timer counting at 1s interval)
always @ (posedge clk) begin
    if (timer_states) begin
        timer_en    <= 1'b1;
        timer_reset <= 1'b0;
    end else begin
        timer_en    <= 1'b0;
        timer_reset <= 1'b1;
    end

    if(timer_reset) begin
        timer <= 8'd0;
    end else if (timer_states & beat) begin
        timer <= timer + 8'd1;
    end
end
```

Figure 8: procedural block for second division timer

That means, as the *OPEN_X* and *CLOSE_X* states are all consecutive timer states, the time will be accumulated across these states, it will be started at *OPEN_0* and accumulates up to *CLOSE_3*. In other words, "*timer >= 36*" condition from *CLOSE_2* to *CLOSE_3* means that it takes 36 seconds from when the door started to open until it has closed (3 seconds for opening or closing, 30 seconds holding open). We purposely adapted to this accumulating timer design as it eliminates the need for intermediate states to reset the timer, therefore minimizes the number of states needed for the door opening/closing process, and therefore simplifies the implementation. However, consecutive timer states become a problem when we want to open the door when the alarm has been triggered. Intuitively, it would have been a transition from *ALARM* to *OPEN_0*, but since the timer has been accumulating since *ALARM* state is entered, multiple states of the door opening states may be skipped when the transition happens. Hence, the solution is to add an immediate state *ALARM_RESET* in between, which waits for the timer to reset before proceeding. With that being said, *DOOR_MASTER* should also switch to *ALARM_RESET* instead of straight to *OPEN_0* to handle the cases where current state has enabled the timer.

2.2. State memory block

State memory is a block that triggers on *posedge clk,* mainly used to update the current state of the *FSM* with the next state. Due to its triggering condition, it can also be used to implement the **MASTER_CONTROL** functionality using an if block, so that we can reset the controller to the initial state at any point. Because the block is being triggered at every clock cycle, we can also implement the **DOOR_MASTER** functionality within this block by adding an *else if* block which listens for the DOOR_MASTER button being pressed with an extra condition that is the current state must be suitable for the action (which includes every states except OPEN_X states and CLOSE_X states, where X is an integer from 0 to 4) These two functions are not represented in the FSM diagram *Figure 4* as their base states are pretty much every states which would cause chaos to the diagram, hence are described here.

2.3. Next state logic block

```
MORSE_9: begin
    if(morse_trigger) begin
        nextstate = MORSE_CHECK;
    end else begin
        nextstate = MORSE_9;
    end
end
```

Figure 9: MORSE_9 case section, inside the case statement of the nex state logic procedural block

The “next state” block contains a switch structure to determine the next state following the current state, when conditions are met. As the name suggests, the block only check if the conditions for next states are met and assign the appropriate value to *nextstate*. In each case of the switch structure, there exist an if-else statement to check if condition to switch to the next state is met, the condition for the if statement are reflected in the transitions shown in *Figure 4*, where *nextstate* can also be deduced at any stages. The else statement is used to assign *nextstate* to the current state, to keep the *FSM* in the same state as before as no condition for switching to next state is met. (Figure 9)

2.4. Output logic block

The block is used to mainly control the behaviour of the 15 LEDs, with the addition of recording the *last_morse_output* into the corresponding index of the *input_morse_seq*. It is implemented as a case structure, with each case being a state. The IDLE_ENTER state is the only state that controls all 16 LEDs, to initialize or reset them. The other states will only operate on the LED indexes corresponding to its function i.e. door *OPEN* and *CLOSE* states only operate on *LED[15:12]*, ALARM only operates on *LED[11:8]*. Hence, the LEDs group are independent from each other.

To determine when to blink the alarm LEDs, if-else statement is used for the states ALARM and TRAP. Two divided clock with customizable heartbeat length is added to **access_control.v**, one for *ALARM* which produce a beat *alarm_beat* every 2 seconds, and one for *TRAP* which produce a beat *trap_beat* every 0.5 seconds. Both beats are 0.15 seconds long, which correspond to the duration that the LEDS will light up for, just enough for human eyes to recognize while it is fast enough to produce the blinking illusion. The LEDs will be turned on if the beats are *HIGH*, and are turned off otherwise. As the alarm LEDs blink in a recurring manner, it is not necessary to reset these two clocks when we enter the *ALARM* or *TRAP* states, hence the two clocks can be left running throughout the circuit.

In terms of recording the morse input sequence, we can keep assigning the *last_morse_output* to the appropriate *input_morse_seq* starting from the *IDLE_EXIT* state. As we update the *last_morse_output* value before sending out *morse_trigger* signal, the block should have the last few clock cycles to write the updated value to the current sequence’s index before the state moves to the next. That also explains why we need *MORSE_CHECK* to check for the sequence validity and not nest the check into *MORSE_9*. As the check should be autonomous, *MORSE_CHECK* does not rely on any I/O input, and moves on right after it has computed the validity check. Also, in the states *MORSE_4* and *MORSE_9*, since we have gathered enough morse signals to form a number digit, we can pass that section of the sequence into the *morseDecoder* to get a binary representation of the decimal number digit, so that we can immediately display it on the corresponding LEDs in the subsequent states (*MORSE_5* and *MORSE_CHECK* respectively). As the block use blocking assignments, we can be assured that the indexes of the sequence that are passed into the *morseDecoder* are all updated beforehand, hence could be carry out in the same state.

3. People incrementing and decrementing

people_count is created as an output to be passed into ssd_control. The last states of the inc/dec switches are recorded every clock cycle in a procedural block, and if the current state is different compared to the last state, the block will either increase or decrease people depending on which switch was flicked, also given that the number of people is within the range of 0 to 9 inclusively. In order to do so, we can use an if-else if statement, a branch for each switch. The statement is encapsulated by another if statement to check if the current state is *OPEN_3*, to allow people to enter exit only when door is completely opened. An else if branch is then added to the outer if statement to overwrite *people_count* to 0 in the case of **MASTER_CONTROL**.

CLIMATE CONTROL MODULE

1. Overall Architecture

This module is composed of 2 sub-modules: **climate_fsm** and **temperature_controller**. The **climate_fsm** module plays a vital role in transitioning between states, while the **temperature_controller**

module regulates the vault temperature by design requirements.

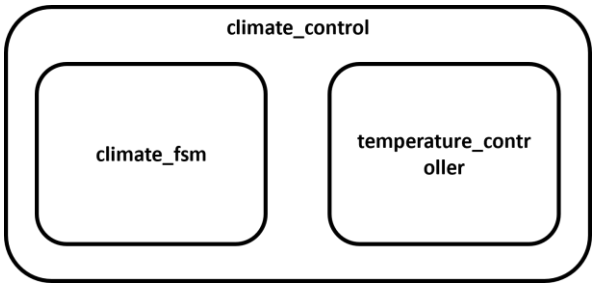

Figure 10: Architecture of climate_control

This module accepts number of people from **access_control** and temperature data from slide switches, while it outputs the vault's temperature and the current climate control status to **top** module, which in turn relays this information to **ssd_control** for displaying the current temperature.

The module's behaviours are mainly determined by its current state and inputs to the module. It can be summed up as follows (For state diagram, please refer to *Figure 5*):

ON: This state is maintained as long as there is people inside the vault. This is the normal operation state of the climate controller module. In this state, the module will cool or heat the vault temperature toward the desired temperature.

OFF: This state is activated in 10 seconds after the last person leaves the vault. The vault temperature is not regulated in this state. Instead, the vault temperature will gradually change toward the outside temperature. Additionally, the temperature display will blink during this state to indicate that climate control is off.

DISPLAY OFF: Upon power up, the system automatically goes to this state. Apart from that, this state is also activated 20 seconds after the climate control is off. Its behaviours are similar to those in the **OFF** state, but with the temperature display turned off.

2. Climate FSM

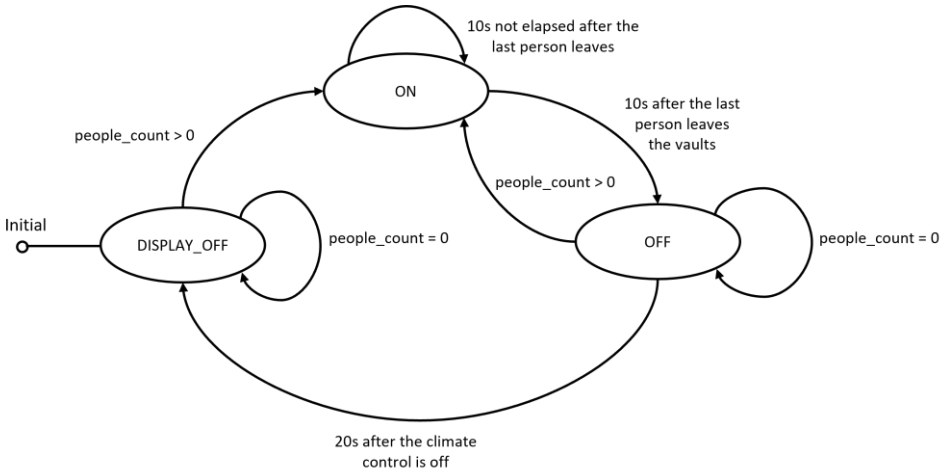

Figure 11: State Diagram of Climate Control

climate_fsm facilitates the logic for transitioning state, mostly based on the number of people inside the vault. It outputs the current state so that the temperature controller can adjust the temperature accordingly. *Figure 5* shows the state transitioning diagram.

3. Temperature Controller

3.1. Initial temperature

As per the project requirements, the initial temperature of the vault should be equivalent to the outside temperature. As such, it is necessary to read the outside temperature only once upon system power up. Thus, we used a 1-bit register as a "flag" named **initialized**: The idea is to move the initialization code to the conditional block **if(!initialized)** so that they are executed only once. Specifically, in the first clock rising edge, the vault temperature is set to the outside temperature, and **initialized** is also updated to 1. In this way, the vault temperature is set to the outside temperature once during initialization.

```
if(!initialized) begin
    next_temp <= outside_temp;
    current_temp <= outside_temp;
    initialized <= 1'b1;
```

Figure 12: Logic for initializing vault temperature

3.2. Temperature Adjustment

The vault temperature changes at different rates and toward multiple target temperatures (based on the number of people inside the vault and the current climate control state: When there are more than 5 people inside the vault, the climate controller adjusts the vault temperature toward the desired level at the rate of $1^o/1s$. Vice versa, the vault temperature is changed by 1^o every $0.5s$. In case the climate controller is off, the vault temperature will change toward the outside temperature instead, and at the rate of $1^o/2s$). As such, we need three timers ($2s$ period, $1s$ period, and $0.5s$ period) and two target temperatures (outside temperature and desired temperature). This makes it challenging to implement and maintain the logic for adjusting the vault temperature. For this reason, our implementation utilises multiplexer-like structures to control the temperature adjustment function.

There are two multiplexer-like structures in our implementation. The first one is used to select a beat signal from one of three timers to update the temperature, and the other one is used to select the target temperature from either the desired temperature or the outside temperature (When the climate control is off, the vault temperature changes toward the outside temperature). This sub-module uses the current state from **climate_fsm** and number of people from **access_control** to choose appropriate signals (Refer to *Figure 7*).

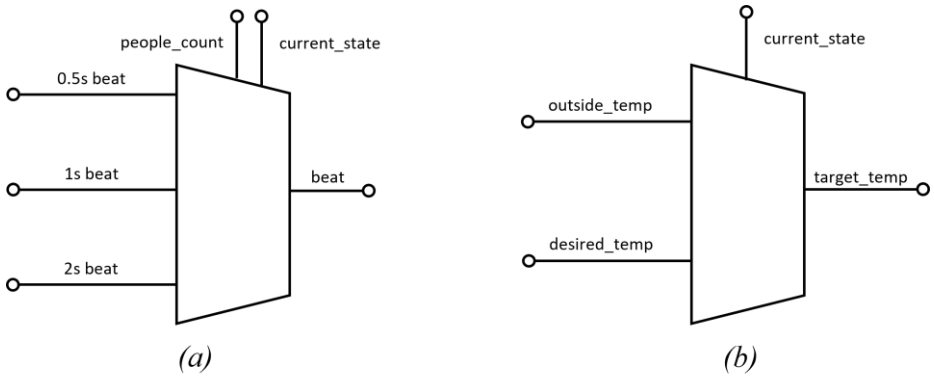

Figure 13: Multiplexer-like structures for selecting appropriate signal. (a) Select timer beat (b) Select target temperature

Every time a beat signal is received, the vault temperature is compared with the target temperature, and be updated if they are not equal. The temperature is only changed by 1 degree for every beat signal received (*Figure 8*).

```
if(target_temp != current_temp) begin
    next_temp <= (current_temp > target_temp) ? current_temp - 3'd1 : current_temp + 3'd1;
end else begin
    next_temp <= current_temp;
end

// Update the temperature periodically
if(beat)
    current_temp <= next_temp;
else
    current_temp <= current_temp;
```

Figure 14: Logic for updating temperature

SSD CONTROL MODULE

This module encapsulates **sevenSegmentDecoder**. It is dedicated to controlling the 4 SSDs to display the values of its inputs. This module accepts people count, last morse key, climate control state, and vault temperature from the modules **climate_control** and **access_control** accumulatively.

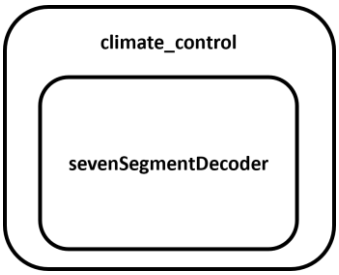

Figure 15: Architecture of ssd_control

This module serves 3 main purposes:

- Uses received people count to display the number of people inside the vault using the 1st SSD.
- Uses last morse key signal to indicate the last entered Morse character using the 2nd SSD. In our implementation, we modified the decoder module to be able to display the Morse dash by adding a case for number 10, in which only the **D** segment is lit (*Figure 9*). For displaying the Morse dot, we just need to enable the DP (decimal point) port.
- Uses climate control status and vault temperature signals to display the current vault temperature. As the temperature is constraint to be in the range 20 – 27, it will use both the 3rd and 4th SSDs for displaying the vault temperature. That also explains why the 3rd SSD is hard-coded to display number 2.

The module also relies on several control lines:

- Climate control status signal determines how the temperature should be displayed (whether to turn on 3rd and 4th SSDs): When the climate controller is active, the temperature is displayed normally. When the climate control is off, these SSDs blink to alert users that the vault temperature is not being regulated.
- **morse_display_status** signal is used to determine if the 2nd SSD should be enabled. The signal originated from **access_control** module and is triggered when it is in a state that is listening for morse input.

```
4'd10: ssd = 7'b1110111;  // underscore aka morse dash
```

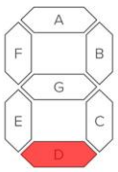

Figure 16: Displaying the Morse dash