COMPSYS 723 Assignment 01

Final Design Documentation

Cecil Symes & Nikhil Kumar

This assignment contains a FreeRTOS system which implements a Frequency Relay control system on a Nios II softcore processor on an Altera FPGA. The project is made for a DE2-115 development board.

Installation & System Information

The Nios II Soft-core and related Avalon interface should be flashed onto the appropriate Altera FPGA. This can be found in the SOPC folder.
Once the system is ready, build the system using the provided make file, which will produce an ELF file which can be flashed onto the Nios II.
The included makefile will link the Assignment01_BSP, which has been modified from the original project BSP. This BSP is slightly modified to provide a 1us timer to the timestamp function which will allow the system to use the Avalon Timestamp timer for timing of latency values.
This can be done using the Command-line using the Nios II Software Build Tools or using the Nios II Eclipse IDE.
Make sure to have the board be currently connected to your device to allow for communication between your machine and the Nios II processor, as this allows for flashing and Nios II command line outputs.

Nios II Eclipse IDE

Import the folder into the Nios II Eclipse IDE. Right click on the Assignment01 folder and select the build option. Once the project has been built, go the run tab in the ribbon, and select “run configurations” From this menu you can create a run mode which will run on the Nios II hardware. Make sure to have the right JTAG port and options selected. From here you can run the built project on the Nios II.

Command-line on Linux Systems

Open up the Terminal and change directory into the Nios II SBT installation path. Run the Nios II command line bash script, found in the Nios II SBT installation path. Running this bash script will set up the environment values to use the Nios II SBT in the open terminal. From here, CD into the project folder and navigate to Software/Assignment01/ . You can then run the make command to build the project. From here, open a new terminal and repeat the steps of running

the Nios II command line bash script. Once that has been completed, run the nios2-terminal command. This will run the Nios II terminal inside the open terminal, allowing to see any printf statements and to ensure that the Nios II processor is running.
Once the project has been built, you can now send the generated ELF file to the Nios II by running Nios2-download -g Assignment01.elf to download the built project ELF file and run the system as soon as it's downloaded.

Operation

The system will start up and begin the Frequency Relay program.
The VGA output of the DE2-115 board will be sending real-time data of the current system, presenting plots which show frequency and the rate of change for the frequency, as well as information about the system state.

Switch Control:

The switches on the DE2-115 board represent wall switches which connect/disconnect loads. The loads are represented by LEDs of the DE2-115 board. These LEDs can be turned on or off using the switches of the DE2-115 board. There are seven switches which represent seven loads of the system, from Switch 0 to switch 6.

Normal Mode:

At the beginning of the program, any switches which are set high will have the corresponding loads connected. This will show as the Red LEDs turning on. The user can connect/disconnect any loads via the switches.

As soon as the system detects an instability based on any given thresholds, the Frequency Relay will transition to Load management mode.

The given thresholds for frequency and rate of change for frequency can be changed in the maintenance mode of the system, which can be accessed from the Button 1 of the DE2-115 board.

Load Management mode:

This mode is entered as soon as the system detects that the frequency is unstable. From here it will begin to manage the loads automatically. A load will be disconnected, and the Red LED corresponding to the LED will turn off, and a green LED will turn on, signaling that the load has been turned off by the system. The system will continue to turn off loads if the frequency continues to be unstable, and will reconnect the loads one by one if the frequency is stable again.

The loads are turned off in order of ascending priority of the loads, with load 6 being the highest priority load, whilst load 0 is the lowest priority.

In this mode, the user cannot bring any new loads online, but can disconnect any loads that are currently turned on, or have already been shed and do not want to be reconnected.

The system will leave the load management mode automatically once all the loads have been reconnected and the frequency remains stable.

Maintenance mode:

Maintenance mode can be accessed via the button 1. Once this mode is entered, the system no longer will be managing the loads, and will connect any loads not currently connected.

From this mode the thresholds which determine if the system is unstable or not can be changed via the keyboard.

Keyboard Instructions:

When in maintenance mode, the user can input values using the keyboard to update the lower frequency threshold and rate of change threshold.
Only numerical digits, periods, and spacebar presses are registered. Up to 4 characters can be entered before the value is taken and used to update the current threshold. If a period is pressed, a decimal point will be placed. If the period is pressed twice, a zero will instead be registered for the second press. The spacebar will update the value immediately without waiting for 4 characters to be entered.

For example, if the number 1,234 is desired, then the digits 1, 2, 3, and 4 can be pressed in order and the value will automatically be submitted.
If the number 1.23 is desired, then the keys 1, ., 2, and 3 can be pressed in order and the value will automatically be submitted.
If the number 1.4 is desired, then the keys 1, ., and 4 will be pressed, followed by a spacebar press in order to submit the value early.

The currently selected threshold to be edited is displayed on the LCD for 5 seconds after submission of the previous value. The first threshold that is edited by default is the lower frequency threshold. After a value is submitted, then the next threshold to be edited is the rate of change threshold. After this, the system returns to editing the lower frequency threshold, and so on and so forth until the user exits maintenance mode.
The values are updated immediately, not upon leaving maintenance mode.

System Design

StabilityControlCheck

StabilityControlCheck acts primarily as a handler for the freq_relay Interrupt Service Routine (ISR). The ISR sends the count received from the Audio Digital Converter (ADC) to StabilityControlCheck through the newFreqQ queue, and then StabilityControlCheck converts

the received integer counts to double frequencies. The conversion from an integer count to double frequency is done in StabilityControlCheck in order to minimise the computation done inside the ISR, as the ISR could retrigger during itself if too much computation was done internally. StabilityControlCheck also stores a copy of the previous ADC count, and uses the old value alongside the new value in order to calculate the system's current Rate of Change (RoC).

StabilityControlCheck checks the global variables that contain the Lower Frequency Threshold (LFT) and the absolute Rate of Change Threshold (RoCT), and determines if the system is stable based on those. StabilityControlCheck is the only task with the ability to write to InStabilityFlag, the global variable that represents that the system is unstable.

At the beginning of each loop, StabilityControlCheck checks the queue from the freq_relay ISR. Using portMAX_DELAY, StabilityControlCheck will simply enter blocking if the newFreqQ queue is empty, allowing the scheduler to work on other tasks. StabilityControlCheck is not periodic, and will run as soon as the queue receives an item and the scheduler is available.

KeyboardReader

KeyboardReader acts primarily as the handler for the ps2_isr ISR. If the system is in maintenance mode, the ISR sends keypresses into the keyboardQ queue. KeyboardReader checks the queue, and ensures that all numerical digits, spacebar presses, and periods are detected and stored. KeyboardReader takes up to 4 characters, including periods, and stores each one into an integer or character array. Once 4 characters are received, KeyboardReader converts these characters into a double. KeyboardReader can also receive a spacebar press before 4 characters are entered to convert the number earlier. KeyboardReader then stores the double into a global variable, representing either the LFT or the RoCT. KeyboardReader is the only task with the ability to write to these two global variables.

KeyboardReader decides which variables "turn" it is to get updated by another global parameter, whichBoundFlag, which acts as a binary value, with 0 meaning the current threshold to be updated is the LFT, and 1 meaning it is the RoCT. KeyboardReader swaps the binary value each time a threshold is successfully updated. The whichBoundFlag variable is global as the LCDUpdater task needs to read this variable.

At the beginning of each loop, KeyboardReader checks the keyboardQ queue from the ps2_isr ISR. Using portMAX_DELAY, KeyboardReader will simply enter blocking if the queue is empty, allowing the scheduler to work on other tasks. KeyboardReader is not periodic, and will run as soon as the queue receives an item and the scheduler is available.

LCDUpdater

LCDUpdater is a low priority task that displays the current threshold parameter being edited during maintenance mode. Its main purpose is to display to the user what the previously entered value was, whether the LFT or RoCT was updated, and which other will be updated next. LCDUpdater reads the global variable whichBoundFlag to see what variable will be changed next. LCDUpdater also reads the LFT and RoCT in order to display the new updated value. After 5 seconds of inactivity, the LCD will revert to showing both threshold values to the user.

The functionality of the task is fairly straightforward. The task does make use of a binary semaphore to synchronise with the KeyboardReader task. This ensures LCDUpdater only updates the LCD when new values are generated, reducing the amount of time the task is active.

PRVGADraw

PRVGADraw sends all the data to the VGA display. It has a very low priority as it does not write to any variables used by other tasks and is overall not critical to system functionality, with the main purpose being to convey system information to the user. It reads a lot of variables in order to display to the screen. The graphs are updated each time a new value is received from the freq_relay ISR. Timing information is also updated when new system response times are received from the LoadManagement task. If new values are not received, the display is not updated and remembers the last displayed frame.

PRVGADraw does calculate average response times, as well as recording min and max response times, but these are small and rapid calculations.

WallSwitchPoll

Wall switch Poll is a polling task which periodically samples the switch state, of which it determines whether the state of the switches have changed, either been raised or lowered. If a difference in the state has been detected, the new switch state is sent over the SwitchQ queue to the Load Management task.

LEDcontrol

LED Control is a simple task which sends values over the avalon PIO to the LEDs. The task takes in an LED status struct from the LEDQ queue. This struct contains values for the Red LEDs, representing loads which have been turned on, and Green LEDs, which represent loads which have been shed. The struct is then accessed and the values are sent over PIO.

LoadManagement

Load Management is a major task with high priority, as it is the main decision making control task for the normal, load shedding and maintenance modes of the load behaviour. Internally, Load Management acts as a state machine which determines when the loads should or shouldn't be connected. The task first checks if the system is in maintenance mode, by obtaining the maintenance flag mutex and then reading the maintenance flag. If it is, the task implements the maintenance mode control loop. If the system is not in maintenance mode, then the task checks for if the system is in the load management mode or if in normal operation. By default the system starts in normal operation. This is determined by the monitorMode flag. Once the state of the system has been fully determined, then either the normal operation loop or the load management operation loop is used. The task runs periodically, as it is the highest priority task which requires the most amount of mutex access of global flags.

Maintenance Mode:

This control loop is entered and consists of maintenance mode initialisation code and the main control loop.The initialisation code, which runs when load management enters the maintenance state , first samples the switch state to make sure that the system state is updated to the appropriate state whilst in maintenance mode, which is the switch state. The load behaviour in maintenance mode is that the loads are only controlled by the switch state. This is reflected in the code as only the system state is updated by the switch state. The load management task obtains the switch state via the SwitchQ task, with the value only applied to the Red LEDs in the global system state. This system state is then sent to the LED control task via the LEDQ queue to update the values of the LEDs.

Normal Operation:
The control flow for normal operation is similar to the maintenance mode of the system. The task allows the switch state to update whether the loads are currently connected or not. This is implemented in the same manner as the maintenance mode control flow, with the use of SwitchQ to obtain the state of the switches, and the system state updated using the switch value. The system state is then sent to the led control task via LEDQ. However, the operation of the system whilst in normal operating conditions is that the system must transition from normal operating mode to a load management state if an instability in the frequency occurs. As such, the normal operation mode checks the system instability flag, which is updated by the Stability Control Check task. If the system has turned unstable whilst the system is in normal operation, then the load management task begins the transition to the load management state. This is done by first instantly shedding a load whilst in normal operation. The timing of this shedding is then recorded to be displayed by the VGA task. Once the load has been shed and the latency has been recorded, the monitoring timer used for the monitoring of the system during the load management state is started and the monitor mode flag is set high, to indicate the system is now in load management.

Load management Mode:
Once the load management task has transitioned into the load management flow, it begins to set up the other two associated tasks which run in tandem to operate the load management state, these tasks being the MonitorLogic and MonitorTimer tasks. The load management flow does this by making use of task synchronization via semaphores. The load management task gives two associated semaphores to the monitor logic and monitor timer tasks, of which the two tasks take the semaphores and then operate their given functionalities. Once they have completed their functions they wait until the load management task gives another semaphore. The flow also consists of setting up the run-once flag such that the state of the loads once leaving the load management state can be correctly ascertained. The flow finally obtains the current switch state from the switchQ queue, and then applies an AND operation between the switch state value and the system state. This was determined to be the quickest way to implement the desired behaviour of the switches during load management, as it is simply making use of a bit masking operation: When the switch state has been lowered (ie a switch which was high is now switched low), then the resulting AND operation will result in the system state matching the switch state, as any bit which represents system state which is ‘1’ whilst the corresponding bit in the switch state is ‘0’ will be driven to 0 in the AND operation. If the Switch

state was to be incremented (switch has gone from low to high), then the resulting AND operation between the system state and the switch state will leave the system state intact. The AND operation inherently preserves the lower state value. This operation is applied to both the Red and Green system state data members, as a system which has been shed, does not need to be reconnected if the switch corresponding to the load is low. The resultant system state from these operations are then sent to the LED control task via the LEDQ.

MonitorTimer

The Monitor Timer function is tasked with controlling the monitoring timer. The monitoring timer is run as soon as the system goes into an unstable state during normal operation. If the system goes from unstable to stable or vice versa, this means that the observation period must be reset. The task is responsible for implementing this behaviour. The task is first synchronized via the load management task, then reads the instability flag to obtain the current stability of the system. If the stability of the system is not that of the previously stability recorded within the task, this means that the stability of the system has changed, and thus the observation period for the load management state must be reset. The task resets the monitoring timer to match this behaviour. If the task does not detect a change in the stability of the system and the timer overflows, thus calling its callback function, the 500ms period is completed and thus the system can then manage a load.

MonitorLogic

The monitor logic task is responsible for determining what action the system should take once the 500ms observation period is up. The task is synchronized via the MonitorTimer semaphore, of which the monitor logic blocks until the semaphore is given by the monitoring timer callback function. This means that the monitoring timer has gone 500ms without being reset, and thus has overflowed.Thus the observation period has been completed. Once the MonitorTimer semaphore is given the system either is required to shed or reconnect a load depending on the system stability. The task reads the Instability flag and then either runs the load shed function or the load reconnect function depending on if the system is unstable or not.

Loadshed Function

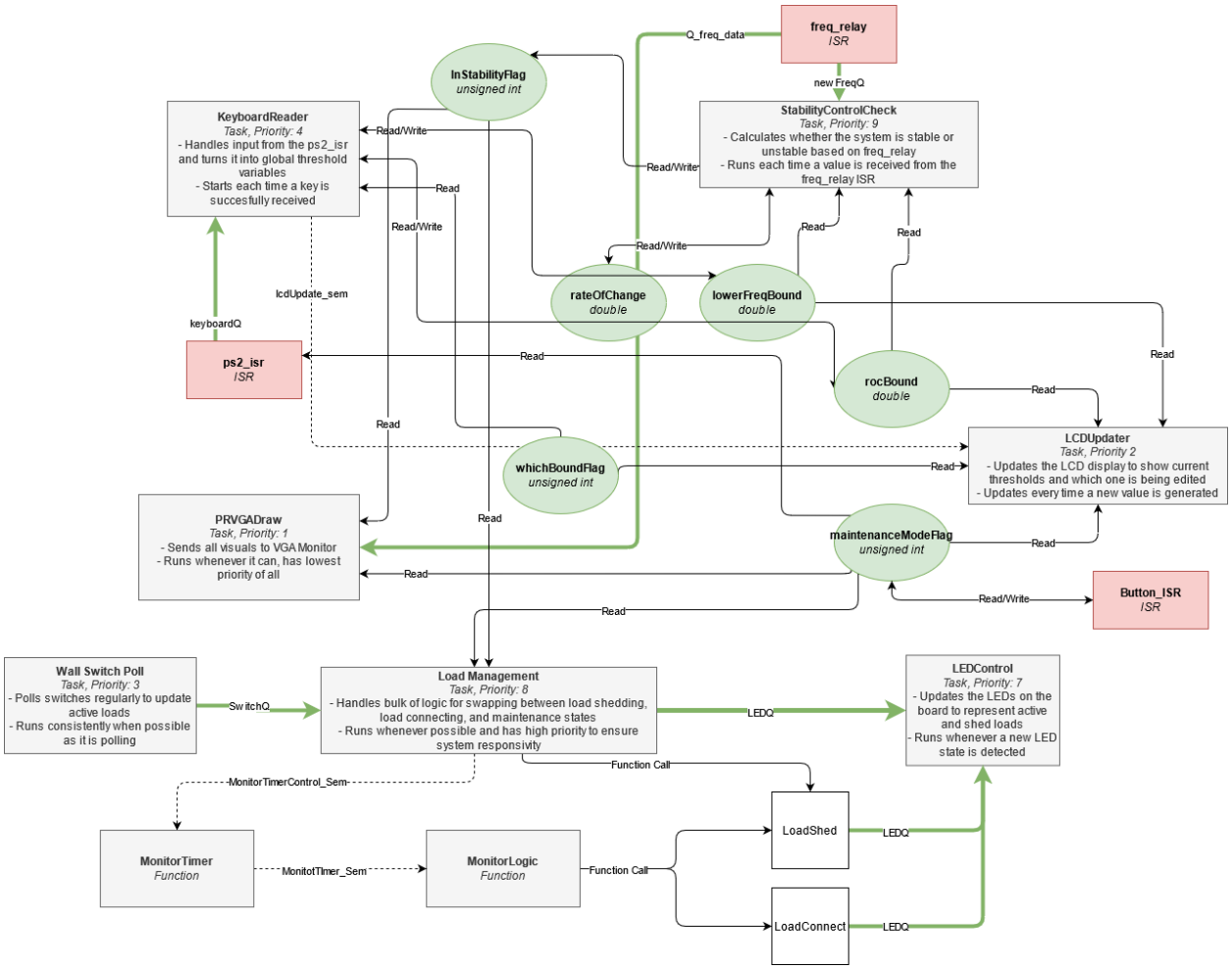
The load shed function works by obtaining the system state and then determining which is the lowest significant bit value that is currently high on the red state value. This is required to be set low on the red state value and the corresponding bit which is required to be set high on the green state value to conduct a shed operation. The lowest significant bit of the red value corresponds to the lowest priority load currently connected, and thus by setting this bit low, the load has been disconnected. This bit is then set high in the green state value to indicate that the load has been shed by the load management system. The function runs a loop of which checks if the AND bit mask operation between the system red state and a variable does not equal zero. If the AND operation is resulting in zero, then the LSB has not been found in that bit position, and thus the variable is left shifted by 1. This continues until the LSB has been found, which is when the result of the AND is not zero. This value is then used to drop the system Red bit low,

and the corresponding green bit high. This new system state is then sent to the LEDQ to be displayed on the LEDs.

LoadConnect Function

The loadConnection function works in a similar manner to the loadshed function. The function obtains the system state and then determines the most significant bit that is currently set on the green state value. This corresponds to the highest priority load that has been shed. The task will then drive this bit value low on the green value, and raise this corresponding bit high in the red value, thus completing a reconnect operation. The function determines the MSB similarly to the loadshed function, a variable is set to the highest possible bit, and then an AND operation is conducted to determine if the bit position is high or low. If the resultant is zero, the variable is shifted down to the next bit position, to be checked again. This continues until the MSB has been found, and thus the bit value is driven low in the green value and raised high in the red value. The resulting system state is then sent on the LEDQ to be displayed on the LEDs.

Diagram of Final System Design



Justification of Design Decisions

Each ISR has a dedicated handler task. These tasks capture any data sent from the respective ISR, and also perform computations in order to shift computation load away from the ISR.

System functionality was split the way it was to ensure that each task was as self-contained as possible. Communication between tasks was aimed to be kept at a minimum, ensuring the design was loosely coupled. This allowed for easier development, as due to the separation of concerns inherent in this approach, we were able to split responsibility for each aspect of the system, and thus work on separate tasks in parallel, speeding up development of the system. With the use of Git, we were able to effectively develop the system in tandem to each other and then merge our changes to complete the system.

For each global variable, only one task has the ability to write. Other tasks may read, but ensuring only one task can write minimises unpredictable behaviour. All tasks must also request a mutex before reading or writing to global variables, ensuring once again that there is no unpredictable data change halfway through a critical section.

Tasks that read global variables make local copies of global variables just before using them in a critical section, or immediately give a This allows the mutex to be taken and given back very quickly, and as the task is only reading the mutex, having a local copy that is updated every time before it is used is functionally the same, but with less mutex possession time.

Global variables have been chosen when data staleness is a factor in decision making. If the data was to be stale then the likelihood of the system making a wrong decision is very high. Thus by implementing global variables which are mutex protected allow for atomic read and writes as well as making sure that reads and writes are sequential to each other, meaning that a read can be done knowing that it is the most recent value possible in the system. If a queue was to be used instead, then the issue of getting values that are far older and different to the current system state means that the would be acting on information that has already been passed due to the transient nature of the data, and thus make a miscalculation.

Semaphores are used for task synchronisation. For example, the LCDUpdater task only needs to run when a new value is detected from the KeyboardReader task. In order to prevent needless updating of the LCD and usage of system resources, the LCDUpdater task checks for the Semaphore at the beginning of each loop. If the semaphore is not ready to be taken, then the task simply blocks, allowing the scheduler to work on other tasks. The Semaphore is given by the KeyboardReader when enough characters have been received and a global variable gets updated.

A lot of tasks have similar block and synchronise functionality, where at the beginning of each loop, if the task needs information from another task to execute, then the task will check its

Queue for data, and will use portMAX_DELAY to wait. If no data is present, the scheduler will put the task into blocking, allowing the scheduler to do other things in the meantime. Once data is eventually received, the task can proceed when the scheduler is ready. This allows tasks to only execute when necessary, saving system resources and allowing other tasks to run in the meantime. Using portMAX_DELAY does have the downside that the task cannot do anything until the Queue receives data, or the Semaphore is received. However, most tasks require the data to arrive before execution can occur, meaning that this is not an issue.

MonitorLogic and MonitorTimer are also two functions which use semaphores for task synchronisation, and both wait on the semaphores by implementing a portMAX_DELAY. The deliberate choice of having the load management operation become split into three tasks and a timer was a deliberate design decision. This choice was made to simplify the load management task, as well as provide concurrency to the major function of the load management state, as the observation period, and management of the observation period should be happening in parallel to updating the loads based on the switch state. This also applies to when the system is shedding a load, as the switch state must be taken into account whilst shedding or connecting.

Priority was assigned so that tasks that are critical to system functionality have the highest priority, and tasks that only convey information to the user are low. Tasks like StabilityControlCheck and LoadManagement are very high priority as they make up a bulk of the system's logic, and also need to respond quickly to ISRs. Tasks like LCDUpdater and PRVGADraw are low priority as while they still serve an overall purpose, they are not critical to the system as a whole and largely serve just to convey information to the user.

The use of the Avalon Timer core to implement the Altera Timestamp function was used in this design. The Avalon Timer core uses a hardware based timer, making for more accurate measurements. The FreeRTOS tick based system was used for the overall system uptime, as it was decided that millisecond or even microsecond accuracy was not important given that the system would be running for seconds and even minutes.

Utilisation of FreeRTOS Features

Many features of FreeRTOS were used in our design, including Binary Semaphores, Queues, Mutexes, and Queues.
If communication was required between two tasks, queues were used. Usage of queues allowed data to be temporarily stacked up, and the receiving task would be able to consume the data when available.
Some variables needed to be read by numerous tasks, such as the LFT and RoCT. In order to prevent creating a large number of queues, global variables were used instead. The benefits are a simpler implementation, but mutexes needed to be used as well to prevent unpredictable data access. Each global variable has a corresponding mutex and all tasks must obtain the mutex before reading or writing to the variable. The mutex must also be given back.

Binary Semaphores were used as a synchronisation mechanism between tasks. If a task only needed to start after another task finished something, then the task could enter a blocking wait state, allowing the scheduler to proceed with other tasks.

Task interaction is through mutex protected globals, queues and semaphores. Semaphores as described earlier are used mainly for synchronization between tasks. Mutexes are used to protect the globals which are used to pass state of the overall system which needs to be accessed by multiple tasks. Queues are used to send data between two tasks, where one to one or many to one relationships between tasks are required for communication.

Limitations/issues in your design

Potential delays due to waiting on mutexes and semaphores
Many global variables are used. This reduces using queues and their associated issues, however using mutexes across multiple tasks introduces the possibility of deadlocking and race conditions.
Using queues has the benefit of buffered communication between tasks, meaning no need to wait for a mutex, and being able to read data from the queue whenever needed. However, queues present the issue of data staleness, meaning that if a single variable is queued sequentially between multiple tasks, it may not be as relevant by the time it gets to the last task. Global variables fix this issue, as all tasks will have access to an up to date value at all times, however this means mutexing must be taken into consideration.

Higher number of tasks means more context switching than maybe required
Having a large number of tasks makes communication between tasks

Use of Unsigned Ints for flags:
Unsigned ints can be 2-4 bytes in size depending on the system. Using 8-bit integers would have been slightly more efficient. While it may not have impacted our design on the DE2-115, a much more resource limited system may benefit from this.

Time & Task Allocation

Task	**Implemented By**	**Time Taken**	**Notes**
Setting up Git and Project	Both	2 hours	Initial project setup
PRVGADraw	Cecil	12 hours	PRVGADraw was updated consistently throughout the project.
KeyboardReader	Cecil	6 hours	

LCDUpdater	Cecil	4 hours	
StabilityCheckControl	Cecil	12 hours	
Switch Polling	Nikhil	1 hours	
LED Control	Nikhil	1 hour	
Load Management	Nikhil	16 hours	
LM: MonitorLogic	Nikhil		Subtask of Load Manage
LM: MonitorTimer	Nikhil		Subtask of Load Manage
Report, Documentation, Final Design Diagram	Both	4 hours	