COMPSYS 303 Report

Nimit Parekh
ECSE Department
University of Auckland

I. INTRODUCTION

Pacemakers are safety-critical medical devices used to regulate heart rhythms in patients with arrhythmia. Due to their critical role in keeping the heart functioning, they must behave in a deterministic way.

This report details the dual implementation of a DDD mode pacemaker, which senses and paces in both the atria and ventricles, while also supporting triggered and inhibited pacing for its corrective actions.

This was carried out in SCCharts and C, both of which are running on an Altera NIOS II board.

The first implementation was done in SCCharts, which provides a framework for ensuring determinism and concurrency in the logic system of the pacemaker. This code compiled to C, which allowed the SCCharts logic to interface with the DE2-115 board.

The second implementation was undertaken in pure C, using interrupts and timers to handle the logic of the pacemaker and the peripherals on the board. This secondary implementation helped illustrate the usefulness of a visual language like SCCharts when trying to enforce concurrency.

II. SCCHARTS IMPLEMENTATION

The SCCharts was implemented in 6 different regions, corresponding to the 6 timers used in the pacemaker diagram provided. Inputs from the heart are either AS or VS (sensed) and outputs from the pacemaker are either AP or VP (pulsed).

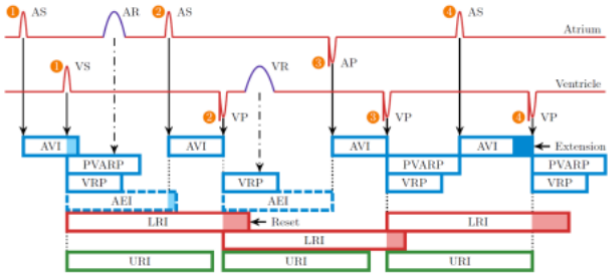

Figure 1: Timing diagram of a DDD Mode pacemaker.

- AVI – Atrioventricular Interval
  - The maximum time between an atrial event and its subsequent ventricular event.
- PVARP – Post-Ventricular Atrial Refractory Period
  - The time after a ventricular event where any atrial events are ignored as Atrial Refractory (AR) signals.
- VRP – Ventricular Refractory Period
  - The time after a ventricular event where any other ventricular events are ignored as Ventricular Refractory (VR) signals.
- AEI – Atrial Escape Interval
  - The maximum time between a ventricular event and its subsequent atrial event.
- LRI – Lower Rate Interval
  - The slowest rate at which the heart is allowed to operate. This is measured as the time between ventricular events.
- URI – Upper Rate Interval
  - The fastest rate at which the pacemaker will ever pace the heart at. This is measured as the time between ventricular events.

Figure 2: Explanation of timers.

Due to the nature of SCCharts, each region runs concurrently with every other region, which is necessary when running multiple overlapping timers.

The original SCCharts version was created by first writing the AVI and AEI timer regions as those were the most simple. This was slightly complicated by the lack of familiarity with the syntax and structure of SCCharts, which resulted in the assumption that output signals could not be read by the code, similar to how VHDL works. This caused the creation of internal signals and extra states, which complicated the logic, and were ultimately useless due to the assumption being incorrect.

AVI ended up having 2 states, idle and active. Active was triggered by an atrium event happening when idle, which would also reset the timer to 0. Once the transition was taken, if a VS was detected, the state was set back to idle. Otherwise, if the clock was equal to the value of the AVI timer, it would emit a VP and transition back to idle. AEI was implemented the exact same way but with atrium and ventricle events swapped.

After these first 2 timers were implemented and were working, PVARP, VRP and URI were easy to implement as they were all written the same way; as a 'block' where if they were active, they emitted a signal that barred AEI and AVI from outputting AP and VP signals by modifying them to check for the status of these timers before proceeding any further. These 3 timers were essentially copy-pasted, which resulted in this version of the program being written very quickly.

The LRI timer was the hardest to write and caused issues that were difficult to debug. Initially in idle, once a ventricle event was detected it would transition permanently to the active state, where the timer would reset every time a ventricle event happened. VP was emitted if the timer reached its LRI value, as that indicated that no ventricle event had happened in that time, which is the maximum acceptable time without a ventricle event happening.

While this first version did achieve the expected output graph when tested, there was a 6 tick delay in the URI and LRI executing. While this didn't result in any missed timings, it was slightly over the 5 tick accepted offset.

After many attempts to reduce this delay, the TA advised that the issue could stem from a delay when checking conditionals after taking some transitions. Guidance about the use of the pre() function was given, as was the suggestion of a code refactor.

The result of this was a recharacterisation of the PVARP and VRP timers. Instead of creating a 'block', the timers would only allow atria and ventricle events to be detected by the other timers when in their idle states. When active, these signals were ignored and so didn't trigger AEI or AVI.

The use of the pre() function ensured determinism by using the value from the previous tick, while the 'immediate' keyword, as well as new syntax such as 'entry' and 'during' helped avoid timing conflicts in a way that the previous version didn't. This resulted in the 6 tick delay being minimised to 1, which could be attributed to other aspects of the programming environment.

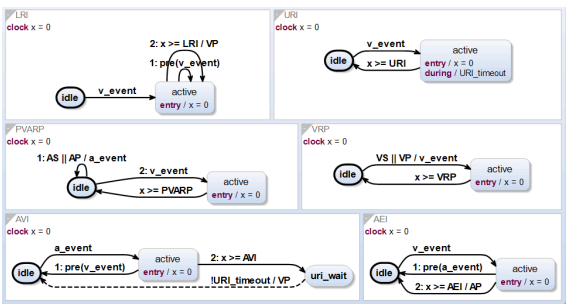

Figure 3: SCCharts state diagram

III. IMPLEMENTATION OF MODES

Both implementations had two modes of interfacing with the board. The first mode is by simulating atrial and ventricular sense events by pressing KEY1 and KEY0 respectively, while the second is communicating with a virtual heart program via UART.

The button mode was more challenging than expected, as the assumption was made that it would be similar in concept to that of Lab 2, which already had code written that performed that task. However this was not the case, and required the assistance of the TA.

While the buttons were being registered as inputs, no output was being produced, and the green LEDs meant to represent AP and VP were not illuminating either. The issue stemmed from the tick not incrementing, so the deltaT function and a timer ISR were used to move the tick along.

This was done by starting a timer alarm, creating a systemTime variable and passing the address of it into the timer ISR, which would then timeout and increment systemTime every 1ms. Another variable called prevTime was created and deltaT was defined as the difference between these two variables. prevTime was then assigned the value of systemTime for the next tick. A function called resetHeartEvents was also used, which set all inputs and outputs to 0 every tick. It was called together with tick() and data() to initialise every tick.

This solved the issue, and the buttons and LEDs started working instantly. A reset button (KEY2) was also added to pause the LEDs and console prints to aid in debugging if necessary and to conserve power.

The UART implementation was more straightforward, although it did require a bit of searching on the internet and asking previous students of the course about the ALTERA UART functions in system.h. 3 functions were created; an init function, an ISR function, and a send function.

The init function initialises the UART module to enable receiving data via interrupts. It first disables all UART interrupts, then enables the Receiver Ready interrupt which generates an interrupt whenever a character is received. It then registers the uart_isr function as the interrupt handler for UART, which tells the processor to call that function when an UART interrupt happens.

The ISR function reads received data from the data register and sets AS_temp and VS_temp variables to 1 based on whether the received character was an ‘A’ or a ‘V’. The use of an interrupt allows for non-blocking UART reads, as it doesn’t block the processor from running the code while it waits for the characters as it is not known when they will be sent.

The uart_send function writes a character into the transmit register and polls until the Transmit Ready bit is set, indicating that the UART is ready to send data. Polling can be used here instead of interrupts as we know when to send characters.

Changing between the two modes was implemented by flipping a switch (SW0), where down is button mode and up is UART mode. The same thing was done to switch between the two implementations, where SW1 being down was SCCharts and up was C. Red LEDs were assigned to each switch and were activated when the switch was up, creating an obvious

indicator of the mode. Pressing the reset button also triggered a red LED, visually showing that the code was paused.

It was tricky to manage keeping the two modes separate so that they wouldn't overlap. The buttons shouldn't accept inputs when in UART mode and UART shouldn't accept inputs when in button mode. This was achieved by using an if/else statement, where the buttons were only read as input if the switch was in mode 0, and AS/VS_temp, the signals set by the UART input reads, were only assigned to AS/VS_input if the mode was 1. This also means, however, that to pause the function of the code, the board must be in button mode as the reset button press would not be detected in UART mode.

In both cases, C and SCCharts, when AP and VP output equalled to 1, green LEDs 2 and 0 respectively were lit up, then reset. The send_uart function was also used to write either 'A' or 'V'. While the UART write only worked in UART mode due to the inputs not being read when in button mode, the green LED indicators were used in both modes as a way to easily view that the pacemaker was working as intended. A for loop was used to extend the amount of time the LEDs were illuminated to see them better, and printf statements were used to distinguish between the implementations, and to debug the input and output signals.

IV. C IMPLEMENTATION

The C implementation was quite simple. However it was written based on the original SCCharts code, so differs in methodology. An effort was made to match the C code to the new SCCharts version, although this was unsuccessful. The timings were off and it was pacing even when the heart was beating completely normally. This could be due to the SCCharts refactor being written to take advantage of the synchronous nature of the language, which doesn't exist in C. This could also have been the result of incorrect implementation of the SCCharts logic. Either way. This led to reverting back to the previous method.

A function called c_implementation was written that works exactly the same way as the previously explained original SCCharts code. This function is called when the implementation switch is up/equals 1. Volatile ints were created to represent the clock timers, which were added to the timer ISR and incremented with each tick. Volatile uint8_ts were used as state and timeout variables, which in hindsight should have been boolean types instead. State 0 corresponded to idle and 1 to active. PVARP, VRP, and URI_timeout variables worked the same way as the original SCCharts logic - as signals that block the output when equal to 1.

Each timer was written as a separate if block, as regions aren't a structure in C. This of course meant that the timers did not run concurrently, and thus the timing delays created would make the 6 tick delay of the SCCharts logic not really matter.

V. TESTING AND VALIDATION

The SCCharts code was tested via the provided online tool. It included a graphical representation that displayed the timings generated by the code for 4 different test cases that simulated different heart conditions.

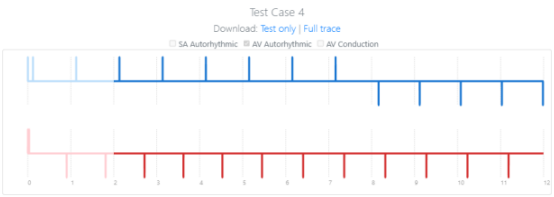

Figure 4: Timing diagram output for Test Case 4

For further, deeper verification of the output, .csv tiles were made available to investigate timings. However due to the amount of unnecessary information present, regex had to be used to reduce it down to the useful data. This analysis showed the 6 tick delay that caused the SCCharts code to be refactored, as visible in Figure 5.

Time	A	V		LRI/URI
132	1	0		
716	0	-1		
1138	1	0		906
1622	0	-1		
2144	1	0		906
2528	0	-1		
3150	1	0		923
3451	0	-1		
4156	1	0		956
4407	0	-1		
5162	1	0		956
5363	0	-1		
6163	-1	0		956
6319	0	-1		
7119	-1	0		956
7275	0	-1		
8075	-1	0		956
8231	0	-1		
9031	-1	0		956
9187	0	-1		
9987	-1	0		

Figure 5: Timing csv output of Test Case 4 with 6 tick delay.

Testing was also done via manually pressing the buttons to simulate heartbeats. Simple, easily verifiable tests were carried out, such as only pressing KEY0 to see if only the AP LED would flash, and vice versa with KEY1 for the VP LED. When pressing both buttons, no LEDs should flash, and when not pressing any buttons, both should pace.

A Heart.exe program was also provided that would communicate with the board over UART and would supply inputs for AS and VS, and would receive AP and VP outputs and display it on a simulated heart monitor. The COM port and baud rate would need to be selected, and the program also allowed for changes to the heart rhythm via toggleable boxes that simulated different arrhythmias. An issue was present however, in that when the boxes were all deselected, while the LEDs and console statements reacted as expected, the program still detected inputs, even though the simulated heart was not doing anything. The TA advised to ignore this issue as the program does not accept AS/VS inputs from the pacemaker so was not an issue with the code.

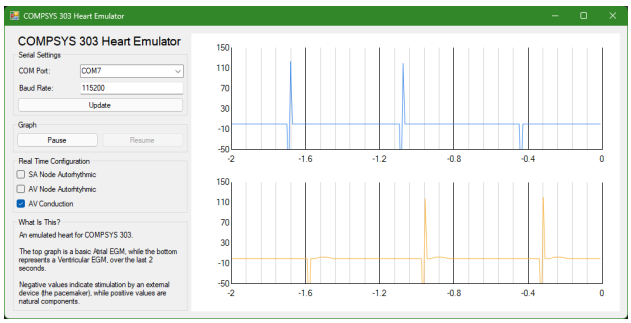

Figure 6: Heart.exe program.

The UART could also be tested by opening a PuTTY terminal and checking that the output received was “AVAVAVAV…”.

VI. WORKLOAD

As this project was conducted solo over a 3 week period, there was no workload split. The SCCharts took the most time, approximately 10-12 hours, which included looking at examples, getting familiar with the language, and the 2 different versions. The buttons also took a while due to the issue with the ticks, and mainly involved debugging. The UART and C implementation took a total of 4 hours as it was quite simple.

Originally files were sent to myself on Discord, but realised that when it was necessary to go

into the labs it would be difficult to manage files. This led to the use of GitHub and git commits to keep track of version history.

VII. FUTURE WORK

A potentially interesting route to take this pacemaker project is by being able to connect it to the internet, which could allow signals and data to be stored in databases for reference. It could be useful to be able to run it remotely, and test different heart conditions. It could also be useful information for doctors, if they could access a record of a patient's heartbeats, and when the pacemaker was needed in comparison to their heart. Having remote access to the outputs could allow the pacemaker to sync with smartwatches or fitness trackers to provide users with real-time heart health insights.

Another avenue worth considering could be incorporating AI/ML into the pacemaker. Machine learning algorithms could predict arrhythmias and adapt pacing strategies dynamically, while AI-based diagnostics could help optimize heart rhythm regulation based on real-time health data.