University of Pennsylvania
School of Engineering and Applied Science
Department of Electrical and Systems Engineering

ESE Senior Design

PVS: Pacemaker Verification System

Sriram Radhakrishnan
sriramr@seas.upenn.edu

Varun Sampath
vsampath@seas.upenn.edu

Shilpa Sarode
sarode@seas.upenn.edu

May 4, 2012

Advisor: Professor Rahul Mangharam
rahulm@seas.upenn.edu

University of Pennsylvania
School of Engineering and Applied Science
Department of Electrical and Systems Engineering

Authors: Sriram Radhakrishnan, Varun Sampath, Shilpa Sarode

PVS: Pacemaker Verification System

Abstract

There are currently about 3 million individuals in the world who depend on pacemakers, which are surgically implanted devices that maintain proper heart rhythm. Pacemakers, complex as they are, are programmed with tens of thousands of lines of code, and are therefore inevitably prone to bugs. It may not come as a surprise then that between 1990 and 2000, over 200,000 pacemakers were recalled due to software issues.

The purpose of the Pacemaker Verification System, then, is to test pacemakers with the realism of clinical trials but without their risks, with the goal of finding software errors before patient implantation. Our system addresses this problem with a virtual heart that can be reconfigured to exhibit different arrhythmias. The virtual heart was built by using finite state machines to model the hearts signal propagation properties for these different arrhythmias. Using code generation tools, these models were then directly translated to an FPGA hardware implementation, ensuring that the behavior of the models was retained.

This virtual heart is able to interface with pacemakers and react to them in a manner that mimics a real heart. It can both output heart beats to the pacemaker and react to paces from the pacemaker. A top-level user interface allows manufacturers, regulators, and cardiologists to view the results of this closed-loop, dynamic system. Such a system allows for interactions that no static heart model can provide, thus offering more robust testing methods and thereby saving lives.

Contents

List of Figures

List of Tables

1 Introduction

Medical devices such as pacemakers are complex by nature. Therefore, it is not surprising that about 41% of pacemakers and implantable cardioverter defibrillators were recalled between the years of 1990 and 2000 [1, 2]. A closer look reveals that these recalls were most often due to software-related issues. With over 80,000 lines of code, this again is not surprising. The number of recalls and type of malfunction is a direct indication that robust pacemaker verification is lacking somewhere between the manufacturing stage and FDA approval. Current FDA verification standards include reports from manufacturers ensuring functionality in open-loop testing scenarios. Specifically, medical devices of life-sustaining nature such as pacemakers are required to undergo the Premarket Approval (PMA) process, which involves animal testing, clinical trials, biocompatibility, and manufacturing guarantees of stress, shelf life, and wear [3]. Therefore, pacemakers are largely approved as long as they exceed some statistically relevant threshold of success for each of these criteria rather than on the guarantee of obeying the inherent laws of electrophysiology. At the end of the day, the reality is that each pacemaker implantation is still a human experiment whose success rests on the statistics of its approval [4].

The proposed hardware platform confronts the issue of pacemaker verification from a top-down approach rather than a bottom-up approach. The Pacemaker Verification System is a human heart implemented on a physical hardware device. One can configure this heart to have varying electrophysiological properties. Namely, this same hardware device can model several arrhythmias simply by changing the heart's parameters. Furthermore, the device will interface with a pacemaker through its input and output signals. By entering various electrophysiological states, the heart can therefore be used by manufacturers and regulators such as the FDA to check that the pacemaker responds appropriately to all relevant arrhythmia scenarios. Instead of basing success on statistical studies with seemingly arbitrary thresholds for what is acceptable, the pacemaker can be put to test in a biological environment well before it is even implanted in a patient. The objectives of this project are as follows:

- to select an FPGA platform
- to implement a configurable model of the human heart on the FPGA board
- to design and implement an interface between pacemaker and virtual heart, allowing for closed loop interaction
- to design and implement a user interface
- to demonstrate the success of our system in catching a known pacemaker error as a proof of concept.

Selecting an FPGA platform relied upon two constraints: resource consumption and budgeting. Since the models themselves were constantly improved and updated throughout the implementation process, only an estimate of resource consumption was before selecting the hardware device. This estimate was made by implementing smaller, simplified heart subsystems and then extrapolating the resource consumption figures to our full model. The quantification process description can be found in Section 3.3.1, with exact figures in Table 2. Secondly, we would like our product to be low cost. The major expense is the FPGA board. Given the average cost of FPGA boards in

the market and especially those that are in the vicinity of our estimated resource needs, we wanted to keep the cost of the board under $200 and successfully did so.

The second objective was to implement the configurable virtual heart. Configurable means that the parameters of the model are sufficiently generalizable so that many arrhythmias can be modeled with the same device simply by changing parameters. For implementation, one of the driving principles behind our project was the minimization of human error in the design process. As mentioned earlier, software is often the prime component of human error in complex devices. Our method to minimizing this error was to utilize model-based design, with which we manipulated design parameters at a high level of abstraction, and then used automatic code generation to acquire the desired program. By synthesizing this generated code on our hardware devices, we ensured that as designers, we had no involvement at the hardware programming layer. Any design changes occured at the model level, which was more graphical and more intuitive and therefore less prone to errors than was manual coding.

The next objective was to interface the heart with pacemakers. To make this heart model accessible to pacemakers universally, we developed the virtual heart by treating the pacemaker as a black box. That is, we only claimed knowledge of its inputs and outputs, and assume that methodology and implementation are completely unknown. As an indication of success, by simply connecting inputs and outputs, we should be able to engage the heart and the pacemaker in a closed-loop conversation. Specifically, not only must the pacemaker react to an irregular heartbeat, but the heart must respond to the pacemaker.

Lastly, a user interface was designed and implemented so that manufacturers and regulators can intuitively switch between heart models. Measurement of success is mainly qualitative in the feedback we received from our current contacts at the FDA and manufacturers such as Boston Scientific. An undeniable measure of success will be if users who are not learned in cardiac electrophysiology can still navigate the interface without difficulty. Therefore, we asked fellow students to provide feedback as well.

PVS is a robust system because of the elimination of software errors through model-based design. It is dynamic because of its configurability. Both of these aspects will be tremendously useful not only for regulators such as the FDA, but also for researchers and developers. Introducing closed-loop testing at various stages in the development process will catch errors early and potentially save the cost of completing the production of defective products. Another potential advantage is the projected reduction of product recalls. Recalls require surgical replacement of pacemakers, which is a highly taxing procedure for patients. Therefore, not only will our product provide additional savings for manufacturers from reduced recalls, but it would also greatly improve patient experience. These goals are inherently long-term in nature. Our recent phone conversations with Boston Scientific indicate that there is interest in the manufacturing industry for this kind of verification system. Our measure of success is therefore an opportunity for our product to be put to test in such manufacturing environments. With time and feedback, the system can be further improved to reach these long term goals. Any actual reduction in recalls or expenses is an added tier of success.

The major constraint of our project was the validation of the component electrophysiological models involved in the heart model. As design engineers, we must validate our model by specifying

what level of detail is sufficient. It would be too cumbersome and quite unnecessary to model the heart to the finest biomolecular detail. First, the complete biomolecular structure of cells in organs such as the heart is yet to be completely understood even in the natural sciences. Additionally, pacemakers only regulate the heart through the heart's electrical system. Therefore, any nonelectrical aspects of the heart, such as the hemodynamics, or blood flow for example, will not need to be modeled. However, timing parameters are tightly linked to the electrical system, and must therefore be taken into great consideration. Success of our product was determined by staying within an acceptable margin of error in timing parameters. We checked that the heart responded in an appropriate manner when pacemaker timing parameters were modified.

2 Discussion of Previous Work

Modeling the physiology of the human body is an important component of understanding how the body works as well as for advancing medical research and development. Our project deals with modeling the heart, specifically the electrophysiology of the heart. There has been some work on this in the past, with various results.

Our main inspiration has been work done in the mLAB at the University of Pennsylvania. They have been working on building a virtual heart model by mapping the electrophysiology of the heart to finite state machines (FSMs). We will be using these FSMs to implement the heart model on an FPGA board, using the concept of model-based development.

There have also been other attempts to model a heart. We shall begin by discussing the work at the mLAB, and then talk about these other groups.

2.1 Heart Modeling by the mLAB

The heart is a four-chambered organ whose beat is regulated by the electrical properties of its cardiac cells [5]. The heartbeat consists of the contraction of the two upper chambers (the atria) followed by the contraction of the two lower chambers (the ventricles). The electrical signals regulating these contractions have timing parameters associated with them. Modeling the heart for our purposes, then, comes down to modeling the propagation of these electrical signals and accounting for these timing parameters.

The heart beat is automatically regulated by the heart's sinoatrial node, or the SA node, located at the top of the right atrium. The SA node generates an electrical signal at regularly timed intervals, thus automatically pacing itself at a rate close to the normal human heart rate of 60-100 beats per minute. This generated signal is passed on to the rest of the tissue in a chain reaction manner, thus causing the entire heart to contract sequentially from the top to bottom. The SA node is a key component of the heart modeled through finite state machines.

The mLAB at the University of Pennsylvania has done extensive work into modeling the heart using finite state machines, and specifically, extended finite state machines (eFSMs).

2.1.1 Finite State Machines

A finite state machine (FSM), or finite state automaton, is a model of a system based on a set of inputs, outputs, and desired behavior [6]. This behavior is mapped into discrete states that are possible for the system to be in. Movement between states is handled using transition conditions. If the condition is satisfied, means that the transition will be taken. This condition can be a function of the inputs and the current state. The term FSM or automaton is used interchangeably to refer to this type of model.

An extended finite state machine, or eFSM, works very similarly to a regular FSM, but they differ in how the transition conditions are described. In a standard FSM, a transition condition can only

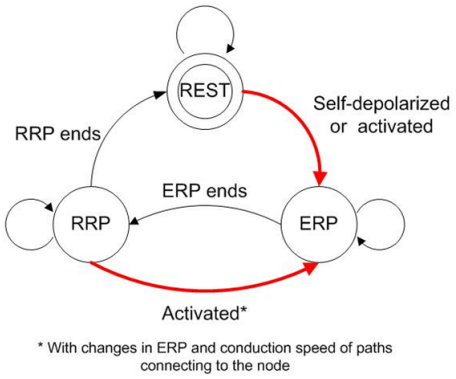


Figure 1: FSM for the Heart Node [1]

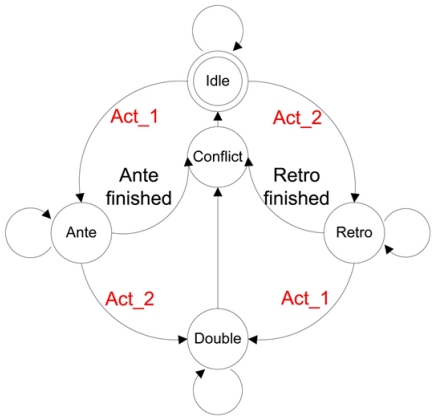


Figure 2: FSM for the Heart Path [1]

be boolean. In an eFSM, transition conditions are functions of the inputs and the current state [7]. These functions need not be boolean, and can in fact use integers or comparisons to denote the condition.

2.1.2 Nodes and Paths

Prior work by the mLAB indicates that a heart can be modeled as a collection of *nodes* and *paths* [1]. A node represents a group of heart tissue cells that share similar characteristics with each other. Paths connect the nodes together based on signal flow in the heart. A heart is similar to an electrical circuit in that signals analogous to current and voltage are propagated through pathways. A node, then, can be considered similar to a node in a circuit, while a path is analogous to the wire connecting two nodes. By connecting nodes and paths in a particular topology, one can model the signal propagation properties of the heart.

Each of these nodes and paths is modeled using an eFSM, as shown in Figures 1 and 2. The node eFSM has three states, derived based on timing parameters of the heart tissue. These three states are *Rest*, *ERP*, and *RRP*. A node cycles through these three states, spending a certain amount of time in each one. This time is determined by the location of the node within the heart, when it receives an impulse, and the characteristics of the tissue that the node represents. When a node in a heart receives a signal, this input triggers a response by the node and starts a timing sequence that is manifested in the node automaton by these three states.

The path automaton represents the connection between two nodes. Signals in the heart can have either antegrade (forward) or retrograde (reverse) motion through a path, based on the directionality of the path. While signals can travel in either direction through a path, there is a preferred direction determined by the tissue that the path represents and the location of the path in the heart. This directionality is represented in the eFSM by the timing parameters of the path, namely how long it takes for signals to propagate through the path. The bidirectionality of the path is manifested in the eFSM with the two possible transitions from the idle state: one for antegrade conduction and the other for retrograde conduction. The double and conflict states serve to account for the effect of two signals simultaneously in the path.

2.1.3 MATLAB Implementation

The mLAB implemented a heart model based on these node and path automata using code in MATLAB [1]. This was purely to get a working model of the heart in the form of software. However, as mentioned in the discussion of model-based design in Section 1, this hard-coded implementation still has a layer of human error. Another drawback is that many of the automaton's transition conditions must be coded for sequential execution in the form of if-else statements rather than concurrent execution, as is inherent in FSM design and in the heart itself.

2.1.4 Simulink Models

Finally, the most recent work deals with implementing these models in Simulink [1]. Simulink is a model-building tool that is a part of MATLAB. The mLAB team has implemented a heart model in Simulink using the node and path FSMs as the building blocks. Simulink also allows inputs and outputs for the system, thus allowing for an interface with the model heart. A major benefit of this approach versus the MATLAB implementation is the ability to scale. By simply adding more nodes and more paths, one can get a more sophisticated and detailed model of the heart.

However, the problem with the existing models is that they are not feasible for hardware implementation, which is the required for our project. While Simulink has code generation utilities that we plan on using as part of model-based development, a hardware implementation has certain constraints that must be accounted for. For example, the current models employ floating-point arithmetic, something that is infeasible for the hardware platform we are considering. In addition, the current models are rather complex, and we must optimize them to better use the limited resources of a hardware platform. We will go into more detail on what we plan to do to solve these issues in Section 3.3.

2.2 Other Heart Models

Teresa Chay of the University of Pittsburgh has created a complex mathematical model that simulates the function of a heart cell under various scenarios [8]. Her current model simulates a small number of cells and their interactions with each other based on an initial impulse. This is similar to what we intend to do in that it models the electrical behavior of the heart cells. However, Chay's project thus far deals more with understanding why arrhythmias occurs and less with how to fix it. Our project is more about building a model that simulates the behavior of the heart for the purposes of testing the ability of a pacemaker to return a heart back to a normal state from an error state.

Another group has done work with modeling the heart using a 3D computer simulation [9]. Denis Noble and his group have built a program that simulates the interactions between cardiac cells. They are quite detailed and account for minute variations between the various cardiac cells and tissue. However, like the Chay project, this is an entirely software simulation. It is more difficult to interface this software simulation with a pacemaker for testing purposes. A hardware platform is more inclined towards an interface with pacemakers. In addition, both Chay's and Noble's models are entirely implemented with software, whereas our platform will be based on a hardware heart

model. This hardware implementation can better simulate the concurrency of the heart (see Section 3.3 for more information).

3 Strategic Plan and Structure

We propose the design of a Pacemaker Verification System through model-based development and code generation techniques. The following subsections will detail the theory of operation, the specifications of the product prototype, the hardware and software required to build the product prototype, the testing and demonstration procedures, and the schedule for development.

3.1 Theory of Operation

The Pacemaker Verification System is composed of several components. The principal component is the field programmable gate array (FPGA) implementing the virtual heart model. The virtual heart model itself is implemented with extended finite state machines as described in our discussion of previous research in Section 2.1. It models the heart electrophysiology as a chain of node and path automata. These models have been simulated using the Simulink and Stateflow software that is part of MATLAB.

In line with our first two goals given in Section 1, we must implement a virtual heart model on an FPGA. The constraints placed on picking an FPGA platform are outlined in Section 3.3.1. Once selected, in order to load the virtual heart model on the FPGA, hardware description language (HDL) code describing the model must be synthesized and placed on the board. A central tenet of this project is that this HDL must be created using model-based development practices with automated code generation. This assures that any properties of the original heart model (which can be simulated and verified with logic) will also apply to the HDL implementation. The tradeoff with this approach is the lack of optimization that handwritten code will have. Machine-generated code tends to be very conservative, as the generator will prefer to unambiguously maintain model semantics over any possible performance optimizations. The machine-generated code then may implement additional logic and/or registers that may be unnecessary for the desired computation. In the worst case, the machine-generated code might implement computations in a manner that is infeasible on the FPGA, since the generator does not know the exact specifications of the underlying hardware. As a result, generated HDL will place a greater burden on FPGA capacity constraints. We will not sacrifice model functionality in order to fit these constraints; instead we will optimize the models in order to generate more efficient HDL. In other words, by redesigning the models to use constructs that we know will synthesize correctly and efficiently on the FPGA, we can avoid bloat in the implementation that may cost us room for future feature additions. One example is trading the precision of computation for die area, such as substituting floating-point arithmetic for fixed-point or lookup table solutions. Section 3.3.2 describes this challenge in more detail.

To interface the FPGA heart model implementation with both pacemaker models and actual pacemakers in line with our third goal given in Section 1, I/O functionality is required. We will use the FPGA's general purpose I/O pins along with external analog conversion devices. Since the heart model will run at the speed of the heart (i.e. 60-100 beats per minute), I/O will not be a bottleneck for the design.

In line with our fourth goal given in Section 1, a user interface will also be built using these

I/O components. Whether to implement this user interface using PC software or through custom hardware will involve trading portability and cost with ease of development.

A system block diagram incorporating all of these components is shown in Figure 3, and a flowchart showing the software development process is shown in Figure 13 in the Appendix.

3.2 System Specifications

The primary goal of our system is to provide a testbed for analog pacemakers. As a result, accuracy in modeling a virtual heart and in developing an interface is of the utmost importance. The following qualitative requirements indicate how our system will achieve the needs of the end user.

In order to meet the needs of the user, the pacemaker verification system must be able to satisfy certain requirements:

- Equivalent accuracy to prior software simulations of the virtual heart model
  - Node and path propagation delay match Simulink simulations in cycle count
  - Timing delays coincide with pacemaker timing delays, such as post-ventricular atrial refractory period (PVARP)
- Connectivity with at least one analog pacemaker
- Ability to simulate at least two different arrhythmias by modulation of timing parameters

As this is a product prototype, there are some quantifiable specifications as well. Table 1 gives quantifiable specifications for the product prototype.

Table 1: Product Prototype Specifications

System total cost	\$150-\$250
Power requirements	$\leq$ 25 Watts
Number of heart nodes supported	$\geq$ 25
Number of heart paths supported	$\geq$ 25
Number of EEG probes supported	$\geq$ 2
Switchable arrhythmias supported	$\geq$ 2
Heart rate emulation	30-250 beats per minute

Specification Justification

We justify the first two quantitative specifications with our understanding of the system architecture. The FPGA board will be the bulk of the costs for the development of this system. By choosing a low-cost board, our product prototype will be more accessible to a wider range of end users. A low power board will also enable more portability. A USB-powered FPGA, for instance,

will enable us to fit within the power constraints and provide the end user with flexibility in using the device.

For the next three goals, we determined that implementing models with at least this level of complexity will provide useful results for our end users. For a prototype level, having at least two arrhythmias supported will allow us to switch models for demoing and testing. Having the flexibility of different heart rates also enables broader coverage. This also showcases the real-time nature of our system, since it is operating at the same spectrum of rates the human heart is capable of.

3.3 Hardware and Software Requirements

The pacemaker verification system will be a Simulink model physically implemented on a field programmable gate array (FPGA) with an external interface to a pacemaker. The following subsections will detail the hardware and software design approach we will take to build the product prototype.

3.3.1 Hardware Requirements and Design Approach

As discussed in Section 2.1, the virtual heart model consists of many connected node and path models. Like the hearts in our chests, these nodes and paths all have timing constraints that must be considered concurrently. Therefore, a hardware implementation will be more realistic than modeling the virtual heart in purely sequential iterative software. Since the model implementation needs to be easily configurable in order to account for various heart conditions, a custom integrated circuit solution would be too cumbersome. The best compromise then is a field programmable gate array (FPGA), which is a device that allows the prototyping of digital logic. It is composed of an array of look-up tables and registers with programmable interconnect. To program this device, the development environment running on a computer will synthesize the appropriate look-up table (LUT) and interconnect configuration for the desired digital logic, and will then implement this configuration on the FPGA [10]. To specify the digital logic, a hardware description language (HDL) will be used. We will use Verilog as our HDL because of our familiarity with the syntax.

Selecting and optimizing for a particular FPGA platform will be the bulk of our hardware design for the first half of this product's design timeline. The implementation of the virtual heart model must be physiologically realistic while not exceeding the hardware limitations of the FPGA. An FPGA has a limited number of look-up tables and registers, with more capable FPGAs having dramatically higher purchasing costs. FPGAs also do not typically have floating-point logic, which is required for the models based on prior research. Therefore, the models must be altered in order to overcome this limitation.

Table 2 shows current estimations of resource consumption for the model implementations. These estimations do not take into account the handling of floating-point arithmetic. If we also assume that these estimations scale linearly, then a virtual heart model with 19 nodes and 19 paths will consume 13319 LUTs. We should conservatively specify an FPGA with approximately 25000 LUTs, which should give a comfortable margin of error. Such an FPGA can run from $80 to $300

depending on the model [11].

Table 2: Resource consumption of singular node and path automaton for Xilinx Virtex-2P FPGA

Parameter	Node	Path	Virtex-2P FPGA Capacity
LUTs	199	502	27392
Registers	73	80	27392

Once the model is implemented on an FPGA, an interface must be designed in order to interface the model with a pacemaker. Initially, general purpose digital I/O pins available on all FPGA development boards will be used to interface with digital pacemaker models. In order to interface with actual pacemakers, however, an analog interface will be required. In order to save costs, we will not pursue an FPGA with built-in analog I/O. We will instead design an interface with external analog-to- digital converters and digital-to-analog converters, with microcontrollers and/or digital signal processors if deemed necessary.

In terms of a user interface, additional hardware might be required. FPGAs do have video graphics array (VGA) hardware for connecting to computer monitors, but the requisite HDL to drive the interface might be overly complex and exceed our FPGA capacity constraints. We can develop separate interfacing hardware using microcontrollers that displays data based on the signals the FPGA sends to the pacemaker, or send data back to a PC over an external bus such as USB or Serial.

A block diagram shown in Figure 3 shows at a high level the overall structure of the pacemaker verification system, including interfacing and user interface blocks.

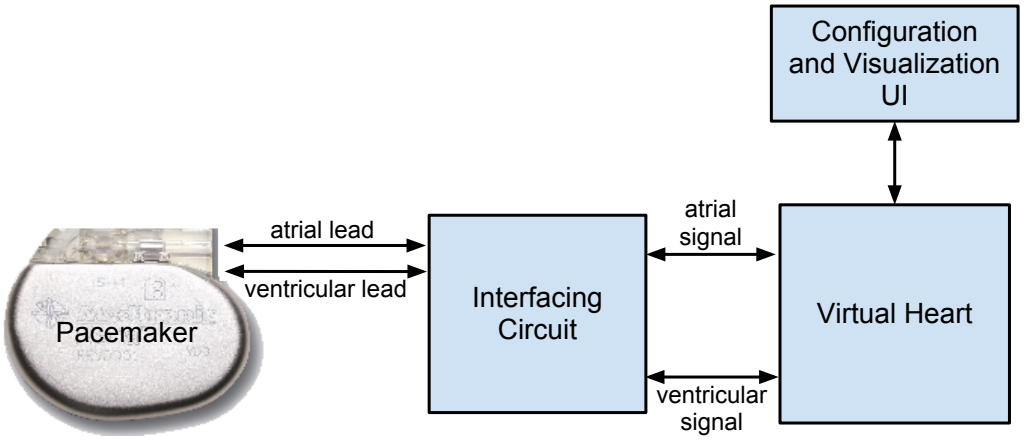


Figure 3: System Block Diagram

3.3.2 Software Requirements and Design Approach

Software development for this project is very different from that of typical software projects because of the requirement of model-based development. To ensure that the verified properties of the virtual heart model will also apply to the hardware implementation, no HDL code can be hand-written. As a result, the bulk of the software development for this project will be optimization and design with Simulink and Stateflow models. The software requirements for the HDL machine-generated from these models are that the implementation should fit within the FPGA's hardware capacity and that it also should have identical behavior to the simulation of the virtual heart model in Simulink.

To meet these requirements, we must utilize the different capabilities of the Simulink/Stateflow software and the Simulink HDL Coder code-generation package. Simulink HDL Coder will perform Verilog code-generation if given models in Simulink, but the efficiency of the generated code is not optimal (for instance, unnecessary shifts and conversions in an attempt to do decimal arithmetic).. To then address the first requirement, we must look into alternative models in order to express some of the calculations that require heavy optimization, e.g. division and multiple multiplications. For example, the generated code performs floating-point arithmetic in some cases in order to calculate values. This is not feasible on the FPGAs we are considering, and as a result, we will have to implement approximation routines in models in order to account for this deficiency. We are currently investigating two options to avoid floating-point arithmetic. One is using fixed-point arithmetic; this has the advantage of being able to use highly efficient integer modules. Another option is using tables, analogous to grade-school multiplication tables, that will allow the FPGA to look up the answer to the computation instead of calculating it with arithmetic routines. Other optimizations may be required in order to account for capacity constraints with block RAM and distributed RAM on the FPGA.

To satisfy the second requirement, we will have to develop a suite of test routines in order to validate the model's implementation. These test routines will be both at the FPGA synthesis level, using Verilog test benches, and also at the hardware level in order to do system integration testing.

Lastly, the Pacemaker Verification System requires a user interface for our users to both configure the system and to run their tests on pacemaker devices. Since the reconfiguration may require re-implementing the model on the FPGA, which requires the full FPGA vendor software suite, this reconfiguration interface has the potential to be extremely complex. Thus, we need to explore options for configuring FPGAs during runtime and/or using interfaces given by FPGA vendors. Additionally, as described in the hardware design section, displaying the output of the FPGA model will require additional software. If we use the FPGA to handle that as well, we will need to write Verilog HDL to configure the VGA controller. On the other hand, if we use an external device, we will have to write software for graphical I/O. Ideally such a display would include waveforms for the heart beat propagation along with the ability to switch between models.

Figure 13 in the Appendix shows a flowchart describing the software design process for developing this product prototype.

3.4 Test and Demonstration

Testing is an iterative process that must be considered throughout the design and implementation of our product prototype. Similarly, we must also keep in mind of how we want to demonstrate our prototype so that our work can be appreciated and utilized. The following subsections detail how we will approach both testing and demonstration preparation.

3.4.1 Test

Accurate testing of the pacemaker verification system is crucial for the product prototype, because this system is in turn a device to use for testing pacemakers. The critical metric for measuring the success of the system then is the accuracy of the implementation with respect to both computer models and real-world scenarios. We must therefore ensure accuracy of results within specified bounds. Since the system is built off timing principles regarding the heart, we can measure accuracy as minimizing the timing difference between signals from the system and signals from the model or from a real heart.

For the implementation of the virtual heart, there are both unit and system tests that can be applied. Our first unit tests are ensuring that the model implementation on the FPGA is accurate with respect to the Simulink simulation. We can simplify models by using oscilloscope measurement mechanisms and comparing the timing of the FPGA output to both the output of the Simulink simulation (which was validated in prior research) and hand calculations with model parameters. We have performed these unit tests with subsystems that included just one node, one node and one path, two nodes and one path, and 19 nodes and 19 paths. All of our results were accurate on the order of one millisecond, which is well within the margin of error when modeling the human heart, since heart signals are on the order of tens to hundreds of milliseconds. As we develop models with different parameterizations (in order to model different heart conditions/arrhythmia), our unit tests will be extended to confirm that the implementation still operates within specification.

While unit tests provide us perspective on the accuracy of the prototype, further integration and system testing is required. To do this, we will require third-party assistance from qualified experts like cardiologists to verify the output of the virtual heart implementation under different operating conditions. A cardiologist at the Hospital of the University of Pennsylvania previously verified the output of MATLAB-based virtual heart simulations, so their verification of the FPGA implementation will provide an extra level of comfort.

We must also consider unit and system testing with regards to the pacemaker interface. We must simulate various input and output conditions and use oscilloscopes to monitor and verify interactions. We can also use “white-box” pacemaker implementations to test this feature. A “white-box” implementation is one where all implementation details are known. Since we then know all aspects of both the virtual heart and the pacemaker implementation, it will be easier to investigate interactions and pinpoint bugs. After this testing, a final round of system testing and verification with real pacemakers and cardiologist approval will ensure accuracy as this functionality is added.

To summarize, the following tests can be and are used to check system accuracy:

- Unit testing by comparing cycle delay for signal propagation across nodes and paths against simulation
- Comparing periodic delays across activations through debugging outputs
- Verifying against pacemaker timing parameters, such as post ventricular-atrial refractory period (PVARP) and atrioventricular delay (AVI)
- Cardiologist approval

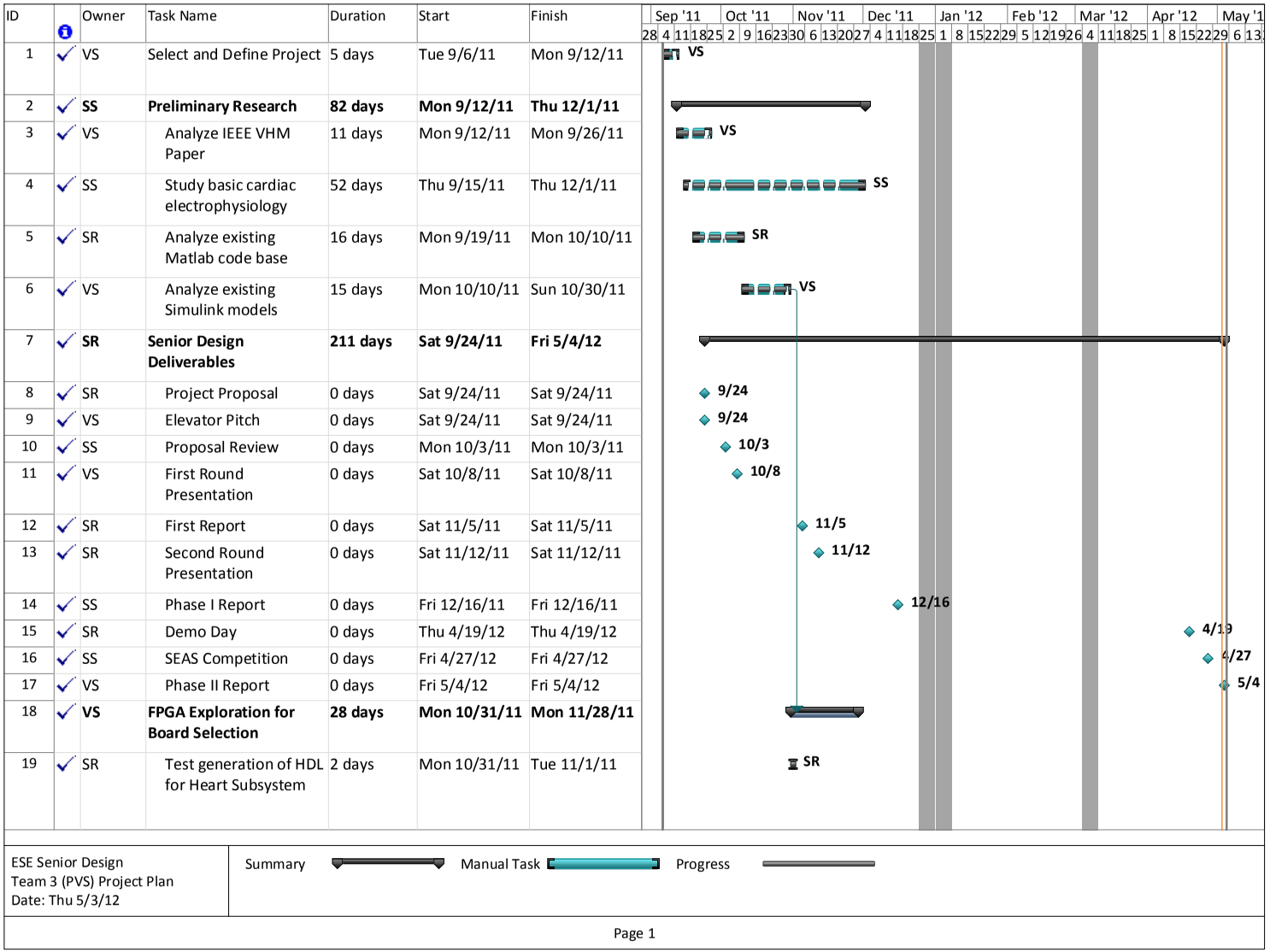
3.4.2 Demonstration

The demonstration of our product prototype will ideally be a fully functional system connected to a real analog pacemaker. Our configuration and visualization user interface will be set up on a connected laptop so that visitors can see the interactions between the two devices and reconfigure the FPGA virtual heart implementation to simulate different disorders. Since viewers who are not trained at the level of our end-users (FDA officials, pacemaker manufacturers, cardiologists) might not understand the output, we will add visual cues to illustrate how the systems interact. This demonstration will require access to a laptop running our software as well as a real pacemaker. We have access to both of these demo requirements and are confident that we can make this demonstration a reality.

3.5 Schedule

This project has several components and requirements, which can only be met through proper scheduling. The Gantt chart for this project that outlines the plan of action for each member of the team is shown on the next two pages in Figures 4 and 5. Section 3.5.2 discusses our current position in meeting this schedule after one semester of progress.

3.5.1 Gantt Chart



ID	Owner	Task Name	Duration	Start	Finish
1	VS	Select and Define Project	5 days	Tue 9/6/11	Mon 9/12/11
2	**SS**	**Preliminary Research**	**82 days**	**Mon 9/12/11**	**Thu 12/1/11**
3	VS	Analyze IEEE VHM Paper	11 days	Mon 9/12/11	Mon 9/26/11
4	SS	Study basic cardiac electrophysiology	52 days	Thu 9/15/11	Thu 12/1/11
5	SR	Analyze existing Matlab code base	16 days	Mon 9/19/11	Mon 10/10/11
6	VS	Analyze existing Simulink models	15 days	Mon 10/10/11	Sun 10/30/11
7	**SR**	**Senior Design Deliverables**	**211 days**	**Sat 9/24/11**	**Fri 5/4/12**
8	SR	Project Proposal	0 days	Sat 9/24/11	Sat 9/24/11
9	VS	Elevator Pitch	0 days	Sat 9/24/11	Sat 9/24/11
10	SS	Proposal Review	0 days	Mon 10/3/11	Mon 10/3/11
11	VS	First Round Presentation	0 days	Sat 10/8/11	Sat 10/8/11
12	SR	First Report	0 days	Sat 11/5/11	Sat 11/5/11
13	SR	Second Round Presentation	0 days	Sat 11/12/11	Sat 11/12/11
14	SS	Phase I Report	0 days	Fri 12/16/11	Fri 12/16/11
15	SR	Demo Day	0 days	Thu 4/19/12	Thu 4/19/12
16	SS	SEAS Competition	0 days	Fri 4/27/12	Fri 4/27/12
17	VS	Phase II Report	0 days	Fri 5/4/12	Fri 5/4/12
18	**VS**	**FPGA Exploration for Board Selection**	**28 days**	**Mon 10/31/11**	**Mon 11/28/11**
19	SR	Test generation of HDL for Heart Subsystem	2 days	Mon 10/31/11	Tue 11/1/11

Figure 4: Compressed Gantt chart showing project schedule (Page 1)

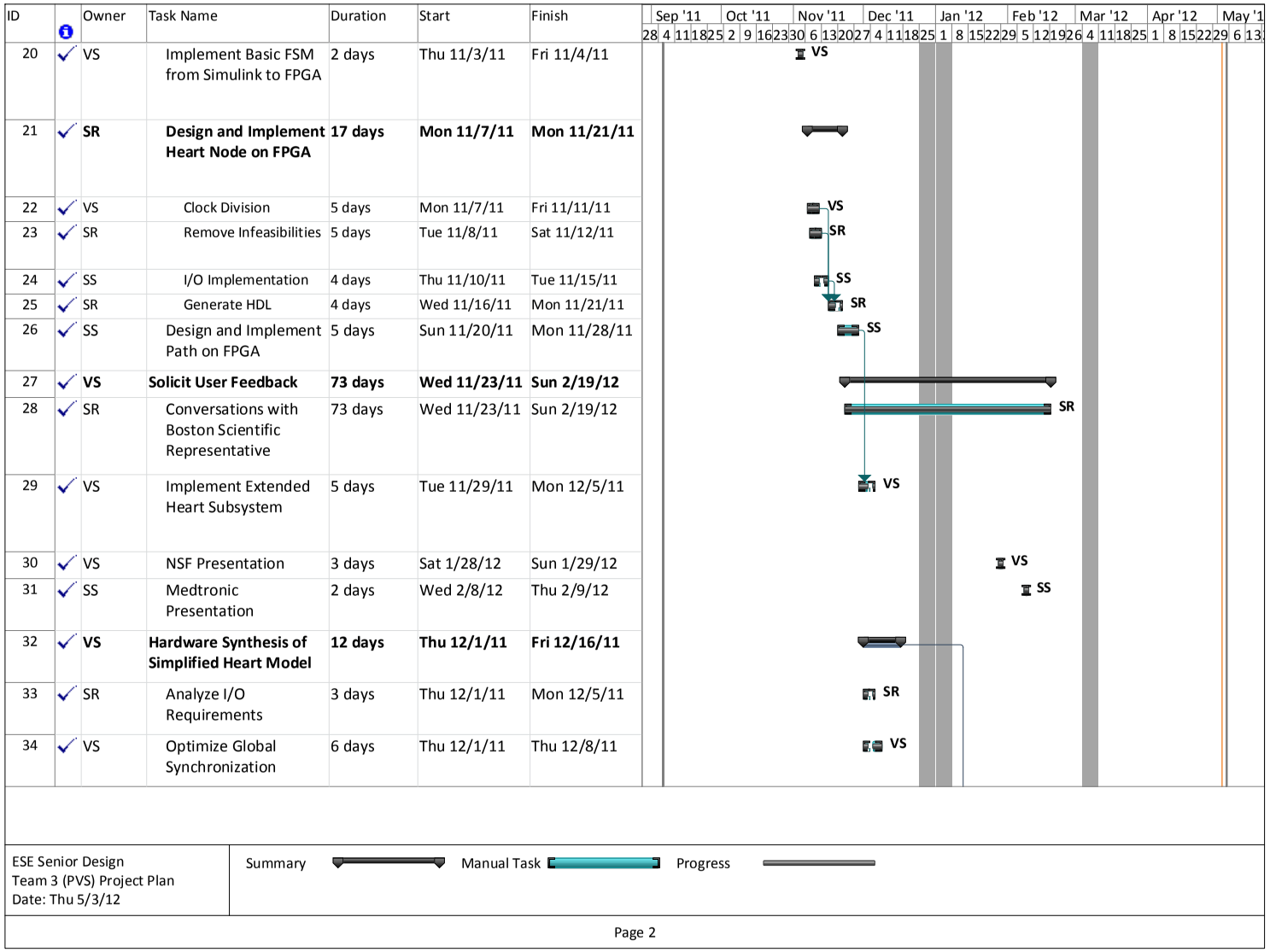
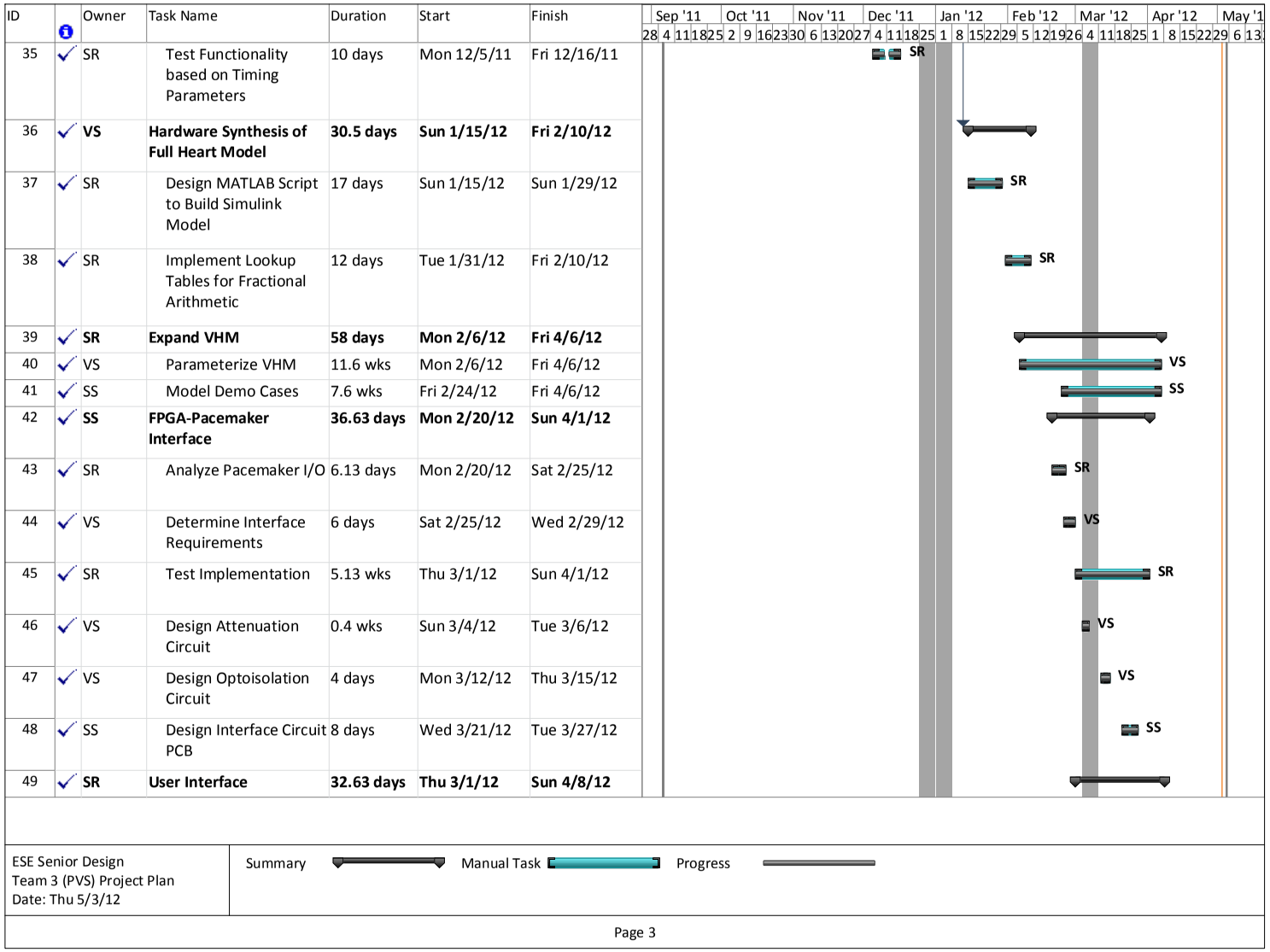


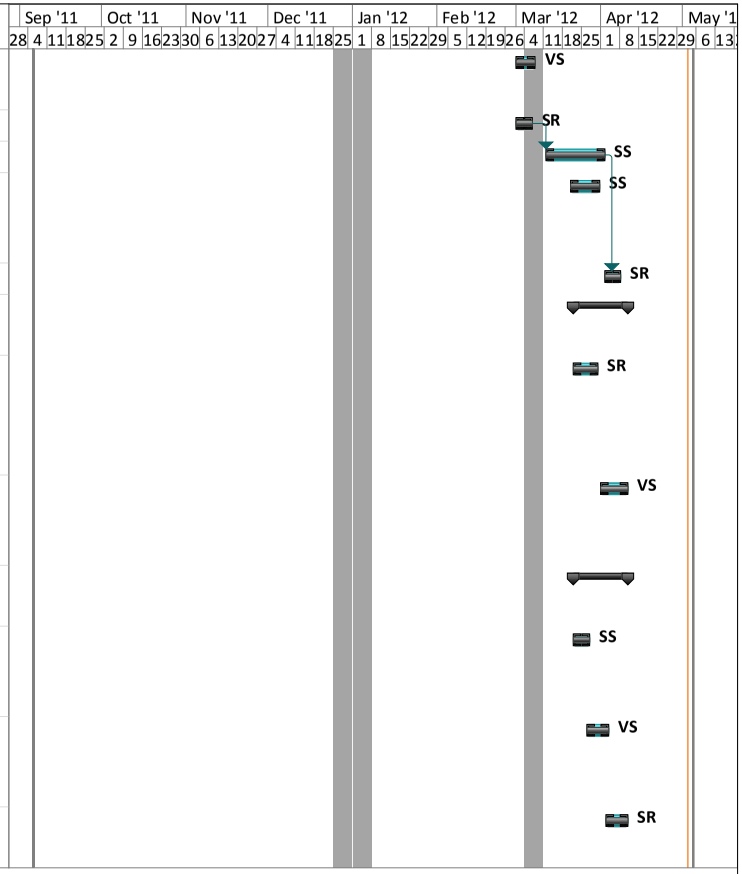
Figure 5: Compressed Gantt chart showing project schedule (Page 2)



ID	Owner	Task Name	Duration	Start	Finish
35	SR	Test Functionality based on Timing Parameters	10 days	Mon 12/5/11	Fri 12/16/11
36	**VS**	**Hardware Synthesis of Full Heart Model**	**30.5 days**	**Sun 1/15/12**	**Fri 2/10/12**
37	SR	Design MATLAB Script to Build Simulink Model	17 days	Sun 1/15/12	Sun 1/29/12
38	SR	Implement Lookup Tables for Fractional Arithmetic	12 days	Tue 1/31/12	Fri 2/10/12
39	**SR**	**Expand VHM**	**58 days**	**Mon 2/6/12**	**Fri 4/6/12**
40	VS	Parameterize VHM	11.6 wks	Mon 2/6/12	Fri 4/6/12
41	SS	Model Demo Cases	7.6 wks	Fri 2/24/12	Fri 4/6/12
42	**SS**	**FPGA-Pacemaker Interface**	**36.63 days**	**Mon 2/20/12**	**Sun 4/1/12**
43	SR	Analyze Pacemaker I/O	6.13 days	Mon 2/20/12	Sat 2/25/12
44	VS	Determine Interface Requirements	6 days	Sat 2/25/12	Wed 2/29/12
45	SR	Test Implementation	5.13 wks	Thu 3/1/12	Sun 4/1/12
46	VS	Design Attenuation Circuit	0.4 wks	Sun 3/4/12	Tue 3/6/12
47	VS	Design Optoisolation Circuit	4 days	Mon 3/12/12	Thu 3/15/12
48	SS	Design Interface Circuit PCB	8 days	Wed 3/21/12	Tue 3/27/12
49	**SR**	**User Interface**	**32.63 days**	**Thu 3/1/12**	**Sun 4/8/12**

Figure 6: Compressed Gantt chart showing project schedule (Page 3)

ID		Owner	Task Name	Duration	Start	Finish
50	✓	VS	Implement Serial I/O	3 days	Thu 3/1/12	Wed 3/7/12
51	✓	SR	Select UI Device	3 days	Thu 3/1/12	Tue 3/6/12
52	✓	SS	Design UI Software	24.63 days	Mon 3/12/12	Mon 4/2/12
53	✓	SS	Design Runtime Scenario Selection Interface	12 days	Wed 3/21/12	Sat 3/31/12
54	✓	SR	Test UI Software	6 days	Mon 4/2/12	Sun 4/8/12
55	✓	**VS**	**Verifying FPGA Implementation**	**23 days**	**Thu 3/22/12**	**Tue 4/10/12**
56	✓	SR	Formulate Simulink-FPGA Comparison Methods	2 wks	Thu 3/22/12	Fri 3/30/12
57	✓	VS	Discuss Implementation with Cardiologists	11.5 days	Sun 4/1/12	Tue 4/10/12
58	✓	**SS**	**Design Demo Day Presentation Scenarios**	**23 days**	**Thu 3/22/12**	**Tue 4/10/12**
59	✓	SS	Study Pacemaker Mediated Tachycardia (PMT) case	7 days	Thu 3/22/12	Tue 3/27/12
60	✓	VS	Determine Model Settings (Timing Parameters) for PMT	9 days	Tue 3/27/12	Tue 4/3/12
61	✓	SR	Implement PMT with VHM and Pacemaker	9 days	Tue 4/3/12	Tue 4/10/12



ESE Senior Design
Team 3 (PVS) Project Plan
Date: Thu 5/3/12

Summary — Manual Task — Progress

Page 4

Figure 7: Compressed Gantt chart showing project schedule (Page 4)

3.5.2 Schedule Discussion

We were able to successfully complete our project. Our schedule presented in schedule presented in Section 3.5.1 illustrates our progress. Tuning of our schedule over the months of February and March allowed us to appropriately pace ourselves in the month of April, so that we did not have too high a burden in April. A "monumental blitz" was spent regardless, as there was an ever-present need to fine tune both the graphical user interface and the demonstration plan.

In retrospect, spending more time with the pacemaker initially would have given us more insight into next steps earlier, and saved us planning trouble in the later months of February and March, and perhaps left more room to accomplish some of the tasks we have now saved for future work. Regardless, we are proud of our product prototype built over Phases 1 and 2.

4 Results

A product prototype of the Pacemaker Verification System was successfully built and tested against a Boston Scientific Altrua S606 Pacemaker. We met our goal of inducing a form of Pacemaker-Mediated Tachycardia (PMT) in our virtual heart in order to show the efficacy of the system. We were able to reproduce Endless Loop Tachycardia (ELT), a pacemaker-induced arrhythmia that causes the heart to beat at an abnormally rapid rate. We use this demo as an example of how our closed-loop testing framework delivers results that typical static input testing systems could not reproduce.

The following subsections detail the designs achieved for the following subcomponents of the Pacemaker Verification System:

- Virtual Heart Model
- Automated Model Generation and Software Toolchain
- FPGA Synthesis and I/O Implementation
- Pacemaker Interface
- Runtime Reconfiguration
- Graphical User Interface
- Device Construction
- User Feedback

We end by discussing our demo of ELT and its implications.

4.1 Virtual Heart Model

The virtual heart model is the centerpiece of the Pacemaker Verification System. Our implementation on an FPGA enables us to model various arrhythmias and test pacemakers against those "hearts" in a closed-loop environment. This section discusses the implementation of the model in both its simplified and complex forms, and Section 4.2 discusses the implementation process.

4.1.1 Simplified Virtual Heart Model

Over the course of Phase 1, a simplified virtual heart model was implemented on our FPGA platform as the basis for future models. As discussed in Section 3.3, FPGA boards have certain limitations, and so the simplified models account for these limitations. Namely, all decimal division was removed, as this was infeasible. Additional work was performed to simplify the communication channels between the node and path automata in setting variable conduction delays.

These simplified node and path automata were connected together to create a full virtual heart model. The topology of this model was based on an existing model created by the mLAB consisting of 19 nodes and 19 paths connecting the nodes. Each node had an output showing activation. Activation refers to the point at which the electrical signal initiated at the SA node reaches the corresponding part of the heart that a node represents. By including activation outputs, the nodes could be monitored to see when they get activated as a measure of signal propagation through the heart model.

As shown in Figure 8, there is a clear, regular rhythm in the virtual heart model. Output 1 (yellow) shows the activation of the sinoatrial (SA) node, which maintains natural rhythm in the heart. Outputs 2, 3, and 4 represent three other nodes in the heart model. One can see that the signal from the first node propagates to the other nodes after some time delay. More importantly, this delay is consistent between heartbeats, and the heartbeats themselves occur at regular intervals. By comparing delay and period measurements from the oscilloscope to the timing parameters set in the Simulink models, proper functionality was confirmed. We also confirmed that the simplified heart implementation fit well within our capacity constraints. Table 3 shows the utilization of the simplified heart implementation on the DE0-Nano FPGA board.

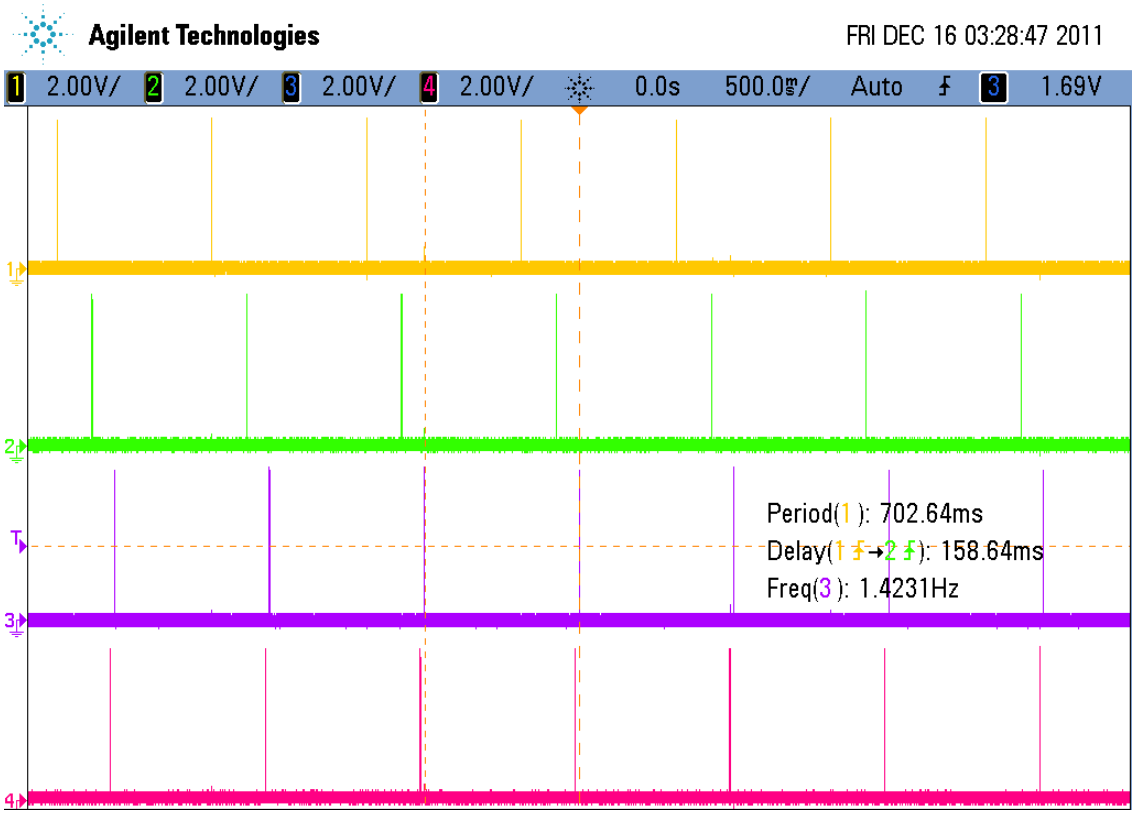


Figure 8: Oscilloscope Reading of Simplified Heart Model

Table 3: Resource consumption of simplified heart model on DE0-Nano FPGA

Parameter	Quantity	DE0-Nano FPGA	Percentage
LUTs	4485	22320	20.1%
Registers	1925	22320	8.6%

4.1.2 Complex Virtual Heart Model

The beginning of Phase 2 focused on reimplementing some of the complexities of the virtual heart model that were removed for the implementation of the simplified model. One such feature was a varying time period for the duration of the ERP state in the node automata. This was a function of the current state at the time of activation, and the resulting value is based off decimal division. Since the denominator is constant at runtime (only variable depending on the node itself), one-dimensional lookup tables were generated for each node automata to calculate this ERP duration. Additionally, two-dimensional lookup tables were constructed for calculating conduction delay ratios in the paths, but this functionality offered marginal benefits and was subsequently removed. Table 4 shows resource utilization of the complex virtual heart model with node automata lookup tables using the same topology as the simplified model presented in the previous section. Resource consumption increased substantially to 34%, due to the use of LUTs as multiplexers for the 1D lookup table implementation.

Table 4: Resource consumption of complex heart model on DE0-Nano FPGA

Parameter	Quantity	DE0-Nano FPGA	Percentage
LUTs	7575	22320	34%
Registers	2288	22320	10%

In summary, virtual heart models with varying levels of complexity were constructed. Both models allow us to emulate hearts with different arrhythmias and provide our base for testing pacemakers.

4.2 Automated Model Generation and Software Toolchain

We have developed an automated system for generating virtual heart models with MATLAB. We can specify a base modeling library for node and path automata along with a topology and delay configurations for each automata instance, and have as a result generated hardware description language (HDL) for the entire heart. We can simply copy this generated HDL into our FPGA I/O harness (described in Section 4.3 and synthesize an FPGA implementation that will run on hardware in minutes.

Figure 9 shows the progression of the model in our software toolchain. MATLAB is used by our automated build tools to generate the HDL, and Altera's Quartus FPGA tools are used to deliver

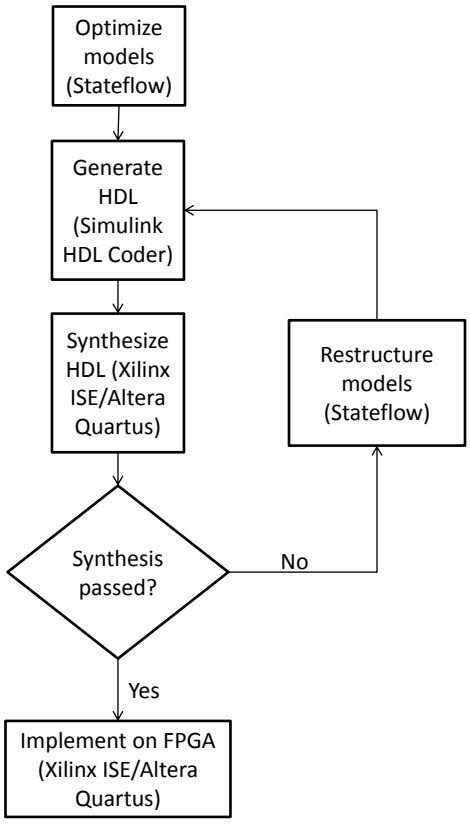


Figure 9: Software Toolchain Flow

our hardware implementation. With minimal future work the Quartus toolchain segment can also be scripted to run automatically. The end result is that the expert user who wishes to design their own models can be given the more intuitive interface of MATLAB to lay out and simulate their models, and our software can transfer this model to the FPGA.

4.3 FPGA Synthesis and I/O Implementation

The virtual heart was implemented on an Altera Cyclone IV FPGA with the Terasic DE0-Nano development board. The DE0-Nano board was chosen for reasons of cost capacity, and I/O. At $60 to $100 depending on distributor, the board is remarkably inexpensive. The capacity was also more than sufficient for our needs. Space tests conducted during Phase 1 (and detailed in Section 3.3.1 indicated that the board would have sufficient capacity for modeling the virtual heart. In practice, the board more than met our requirements for both virtual heart implementation and harness (for facilitating I/O) implementation. In terms of the I/O itself, the development has plenty of general purpose, bidirectional I/O pins that we could use for both facilitating debugging and practical I/O considerations.

FPGA synthesis involved using the Altera Quartus II FPGA development software. Synthesizing a new update to the Pacemaker Verification System is extremely straightforward. Because of the automatic HDL code generation, the build involves simply copying the new HDL to the build harness and invoking synthesis, all of which can be scripted.

The build harness allows the generated virtual heart to be seamlessly interfaced with the board I/O. The harness has the following components:

- Phased-locked Loop (PLL) - needed for configuring the heart clock rate at 1.5kHz
- LED I/O - for debugging and visualization
- Serial transmitter - for transmitting heart data to a computer for interfacing
- Warning light - indicator for heart bradycardia or tachycardia
- Pacemaker input capture
- Mode setter - for heart runtime reconfiguration

Of particular note are the serial transmitter, pacemaker input capture, and mode setter. The serial transmitter is designed to send heart model data to a computer for visualization on a graphical user interface. Since the virtual heart model is entirely based on timing, timing values were captured at various triggers and then sent across the serial I/O line. Timing values were captured by maintaining a "clock" whose value could be read and interpreted by the PC at the other end of the serial connection. This timing value was represented as a 4-byte unsigned integer, and a one-byte header was added in order to distinguish different timing triggers. The triggers were set to sinoatrial node (i.e. atrial) activation, right ventricular node (i.e. ventricular) activation, and atrial and ventricular pacing signals. The transmitter is able to distinguish between pacing and sensing events or the combined pace and sense event so that all possibilities of results can be represented. The transmitter itself is built using a finite-state machine to push the total 5 bytes of data whenever necessary at 115200 baud. The resulting transmitter is able to cope with fast tachycardia scenarios thanks to its high baud rate, and has been reliable in providing data for visualization.

The pacemaker input capture hardware is necessary because of pulse widths coming from the opto-isolation interface. Due to noise and attenuation, the pacemaker input signal had an amplitude of greater than 2 volts and a width of only 70 microseconds. The pulse width needed to be increased since the heart only samples at a rate of 1.5kHz. A finite-state machine was designed to latch pacemaker pace signals appropriately so that the heart model could receive their input.

The mode setter enables the Pacemaker Verification System to be reconfigured at runtime to model different arrhythmias. It consists of a serial receiver that interprets data sent from a PC across an additional serial line. Additionally, signals for premature atrial and ventricular contractions are encoded into the reconfiguration bytes. See Section 4.5 for further discussion on runtime reconfiguration. This mode setter interface is scalable to support a variety of other inputs, so in future revisions many more parameters can be reconfigured at runtime for the virtual heart model.

In summary, the FPGA I/O harness combined with the machine-generated virtual heart model allow us to seamlessly revise and configure the Pacemaker Verification System.

4.4 Pacemaker Interface

A circuit for interfacing the pacemaker with the FPGA-based virtual heart was designed and fabricated. The circuit accomplished its two goals of correcting voltage levels between the FPGA and the pacemaker and isolating the signals between the two devices.

As a precursor to discussing the internals of the circuit, it is necessary to discuss the design of the pacemaker input and output systems. The pacemaker interfaces with the heart through two leads, one for the atrium and one for the ventricle. Each lead has two conductors, one called the ring and the other called the tip. To pace, the pacemaker applies a potential difference from the ring to the tip with a programmable width and amplitude (width ranges in tens of milliseconds and amplitude has a maximum of 7.5V). To sense, the pacemaker simply detects a voltage difference between the ring and the tip above a certain threshold (which has a maximum level of 10mV). Boston Scientific technicians advised that sensed voltages should stay close to 15mV in amplitude.

Since the FPGA board used has digital pins that accept a high input from 3-5V, and output a high signal at approximately 4.2V, it was necessary to attenuate the output "heart beat" from the FPGA to the millivolt range and to program the pacemaker to output a pace signal with an amplitude of 3.5V. Since the impedances for the pacemaker leads was unknown and crosstalk and supply noise was a concern with low amplitude signals, isolation was also incorporated in the circuit through opto-isolation. Figure 10 provides a block diagram for the interface circuit between the FPGA heart and the pacemaker.

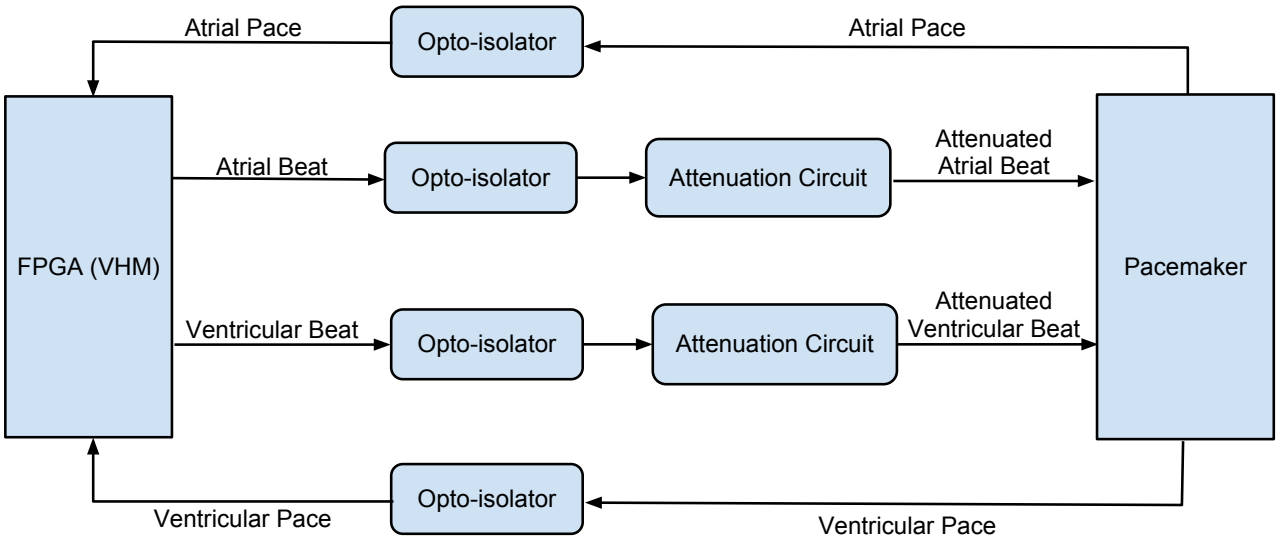


Figure 10: Pacemaker Interface Circuit Block Diagram

The final version of the attenuation circuit was built using the LF347N JFET operational amplifier (op-amp). This op-amp was used to build two inverting amplifiers in series. The first amplifier

had a fractional gain of 0.0015, and the second amplifier had a gain of 1 (simply used to invert the signal). The isolator was used to bring the input "heart beat" to a 9V peak, so that the attenuation circuit output had a theoretical pulse amplitude of $9 \times 0.0015 = 13.5$mV. This allowed the pacemaker to correctly sense the heart beat given its sensitivity range. The output pacing signals from the pacemaker were not attenuated, but were isolated so that the FPGA only received signals powered through its own supply. To maintain a clean supply source for the isolation and attenuation circuits, two 9V DC batteries were used. The resulting circuit allowed the system to have full pace and sense functionality with a current pacemaker. Figure 14 in the Appendix has the full schematic for the interface circuit.

This circuit was additionally fabricated. Layout was performed using the Eagle CAD software package and was designed to fit in a printed circuit board that was approximately the same size as the FPGA for design. An image showing the Eagle layout of the board is given in Figure 15 in the Appendix.

4.5 Runtime Reconfiguration

An important aspect of the Pacemaker Verification System is to be reconfigurable so that pacemakers can be tested against hearts with different arrhythmias. While new models can be designed in MATLAB and synthesized on the FPGA using our software toolchain, it is important that the model already implemented on the board is reconfigurable at runtime so that users can instantly change their test configuration. The current implementation has two parameters exposed that already allow substantial freedom in changing the emulated heart condition. The first parameter is the sinoatrial node rest period, which dictates the amount of time the heart "rests" before its next activation. In other words, changing this parameter directly changes the heart rate of the model. The second parameter exposed is the atrioventricular path forward conduction delay, which controls the delay in signal propagation from the atria to the ventricles of the heart.

The mechanisms to allow this reconfiguration are built in to the model and into the FPGA I/O harness. The MATLAB models themselves expose these two parameters for reconfiguration, and the mode setter module (discussed in Section 4.3), allow these values to be set via serial I/O. The graphical user interface then exposes these values for user manipulation.

Additionally, the models and the FPGA harness allow for premature atrial and ventricular contraction events to occur. These one-time events allow the virtual heart to emulate extra beats in the atria or the ventricles, which are natural events that can lead to scenarios like endless-loop tachycardia. Emulation of these events allow us to deterministically evaluate their impact, which is an advantage that testing via clinical trials cannot provide due to the inherent nondeterminism of the human heart. We provide one example of their utility in the discussion of endless-loop tachycardia in Section 4.9. In future work, the runtime reconfiguration system can be expanded to include far more parameters (approximately 50 in the currently used model). This will allow expert users to be able to fully reconfigure the virtual heart to expose any arrhythmia/test condition they wish to evaluate a pacemaker against.

4.6 Graphical User Interface

A graphical user interface (GUI) was designed in MATLAB to both configure and visualize the output of the Pacemaker Verification System. The GUI communicates via a bidirectional serial interface to the FPGA virtual heart, so it can both configure and visualize the heart state in real-time. On the visualization side, the GUI displays pulses for any event data transmitted by the virtual heart. This event data is currently the occurrence of an atrial or ventricular sense or an atrial or ventricular pace. The GUI collects this data in real-time and intuitively separates it into different plots. It also calculates the heart rate and displays it in a meter on the bottom-right corner for straightforward recognition. When the heart rate exceeds the tachycardia limit or dips below the bradycardia limit, the GUI flashes and plays an alarm signal to indicate trouble. In normal operation, a beep occurs for every ventricular beat in order to mimic hospital heart monitors. Additionally, to aid intuitive visualization, the GUI can overlay a surface electrocardiogram (ECG) waveform over the atrial and ventricular events. The ECG scales appropriately in regard to changing atrioventricular delay.

In terms of configuration capabilities, the GUI enables the runtime reconfiguration of the sinoatrial rest period and the atrioventricular forward conduction delay, and the emulation of premature atrial/ventricular contractions as described in Section 4.5. It can also dictate that the virtual heart disconnect itself from the pacemaker, so that we can see how the heart behaves without pacemaker interaction.

With its dual roles of visualization and reconfiguration, the GUI has become an essential component of the Pacemaker Verification System. Future work can expand it further to expose more parameters to expert users and even more data for visualization. A screenshot of the current GUI is shown in Figure 11.

4.7 Device Construction

A physical enclosure was built to accommodate the FPGA virtual heart, the interface circuit, the battery supplies, and the connection to the pacemaker. The enclosure was designed in SolidWorks. Figure 12 shows the final device design.

4.8 User Feedback

We have conducted conversations with various potential end users for feedback on our design and feature suggestions. One such conversation was with Jonathan Krueger of Boston Scientific, a medical devices manufacturer. Jonathan is an engineer in the Cardiology, Rhythm, and Vascular Group, and is very experienced in pacemaker design. We learned from him how pacemakers are currently tested in his company, namely with custom simulators. He is intrigued with our work with FPGAs and offered a feature suggestion of simulating “ectopic” arrhythmia, where the cardiac rhythm arises from outside the normal sinoatrial (SA) node.

We have also demonstrated our project to Dr. Sanjay Dixit, who is Director of the Cardiac Electro-

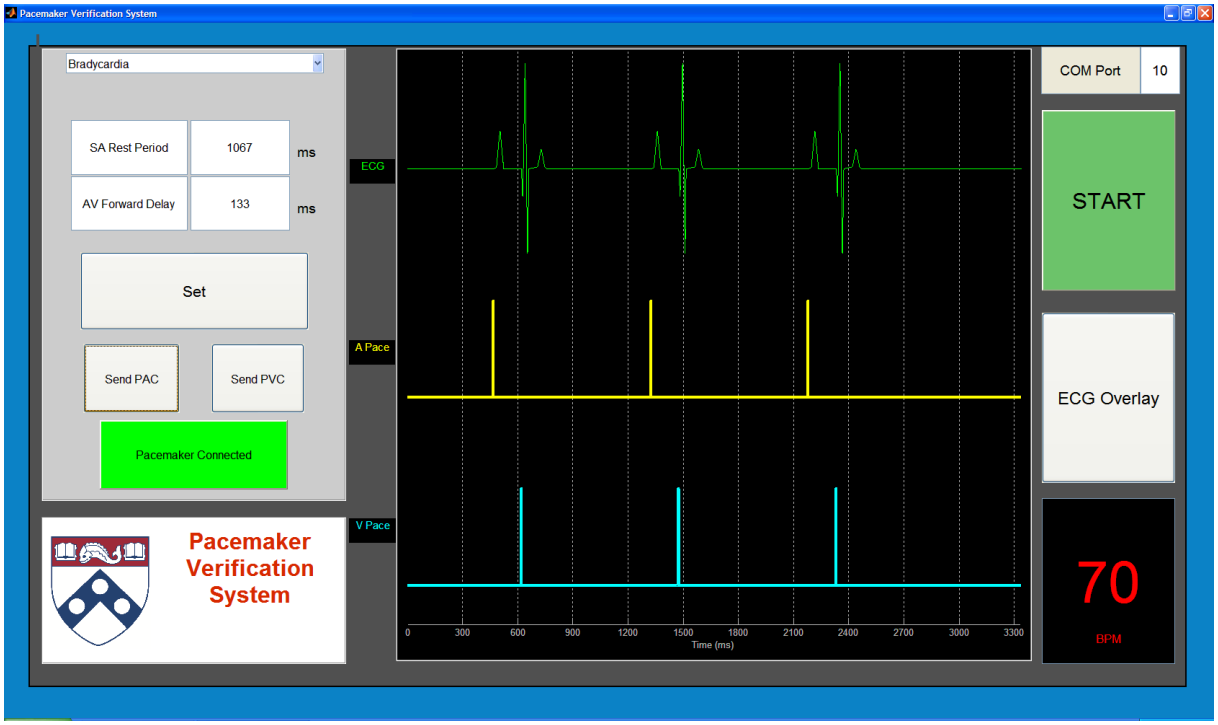


Figure 11: Graphical User Interface

physiology Laboratories at the Philadelphia VA Medical Center. He is appreciative of our efforts to model the electrophysiology of the heart in hardware, and suggested several newer pacemaker algorithms to test against next with newer heart models, such as bi-ventricular pacing and post-PVC (premature ventricular contraction) response. He also suggests the use case of education, where the Pacemaker Verification System can be used to teach medical students about heart electrophysiology and also the impact of a pacemaker. For this use, he suggests making a more technical user interface that exposes more parameters and runtime details.

The conclusion we received from our feedback conversations is that the Pacemaker Verification System has a solid foundation in its product prototype form, and that there are plenty of avenues to pursue further work.

4.9 Endless-Loop Tachycardia Demonstration

We successfully demonstrated a real pacemaker putting our virtual heart into endless-loop tachycardia (ELT) in our Pacemaker Verification System. ELT is a pacemaker-induced arrhythmia that causes the heart to beat at an abnormally rapid rate. For it to occur, the heart must have certain properties that make the pacemaker's various timing delays destructive. One property is the retrograde conduction of a signal from the ventricles to the atrium, a property that is built into our path automata and has reconfigurable delay. Another property is the possibility of a premature ventricular contraction (PVC). This premature contraction retrogradely conducts to the atria, where it is detected as an atrial sense. The pacemaker follows with a ventricular pace, which retrogradely

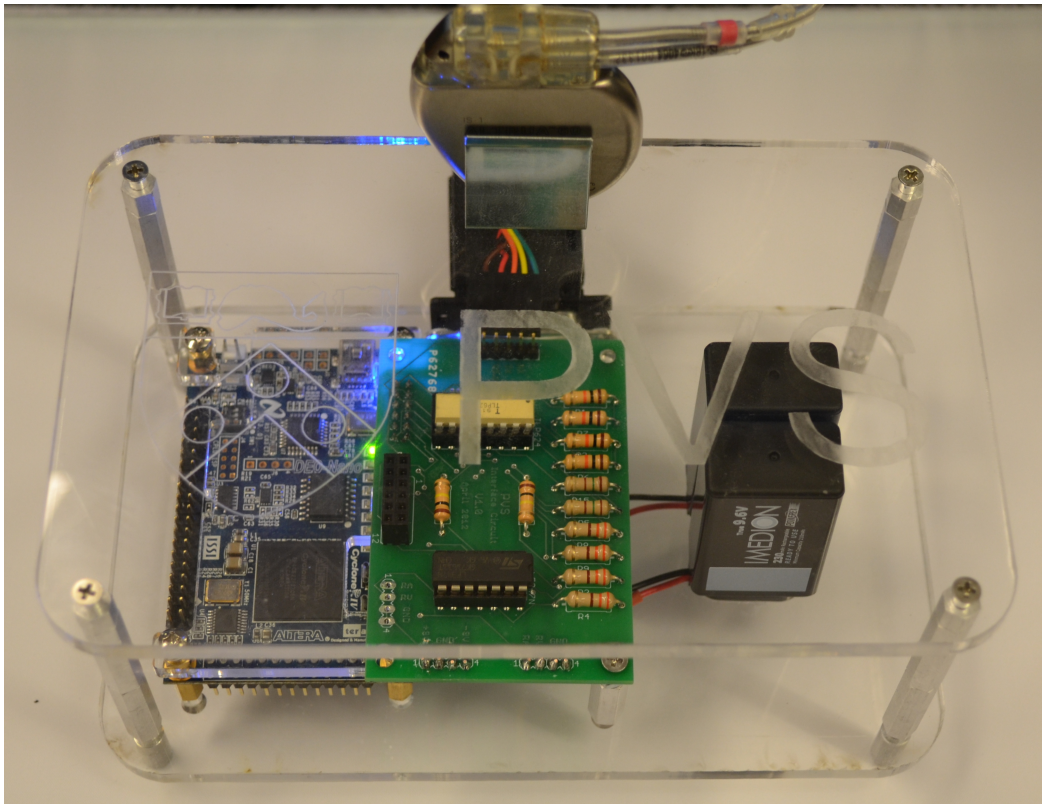


Figure 12: Final Product Picture

conducts as before, starting an endless feedback loop.

We must note that a scenario like this can only be tested with a closed-loop system like the Pacemaker Verification System. The feedback loop can only occur if the pacemaker responds to the heart's output and the heart's retrograde conduction responds to the pacemaker's output. By demonstrating this scenario, we validate the necessity of a closed-loop testing environment.

Demonstration of this scenario also verifies our model. For this scenario to work, the model had to have retrograde conduction delays greater than the post-ventricular atrial refractory period (PVARP), which we were able to systematically tune to be the case. The cyclic period of the heart was also validated with the pacemaker's lower rate limit (LRL), and the maximum tracking rate also validated the model's pacing ability at the correct rate. By measuring the system's operation against a black-box pacemaker, we were able to verify its accuracy.

Endless-loop tachycardia is not a new phenomenon in dual-chambered pacemakers [12], but the fact that the virtual heart was able to reproduce this error with a real pacemaker provides another level of validation to our design. If we provide the appropriate level of reconfigurability, then a specialist will be able to apply different timing parameters to the virtual heart such that they can test corner cases for other algorithms, and perhaps even new pacemaker algorithms on the horizon. This is a key strength of the platform; the generic and flexible design of the virtual heart model enables it to be applied to a variety of scenarios. The scenarios outlined by our user feedback in Section 4.8 are all possible given the right timing parameters for the model. This demonstration

of our product prototype gives us a glimpse into its capabilities, and suggests a bright future with further work.

5 Lessons Learned

There are many lessons learned during the course of a year-long project that one wishes they knew at the beginning. One of the most important tools when working with a team on an extended project is having a strong, clear schedule. This schedule should clearly outline the various components and sub-components of the project, and which team member will be working on each part. Such a schedule allows anyone, whether it be the team or an advisor, to know at any time how the project is proceeding and whether every team member is contributing equally to the project. Having a schedule also allows the team to plan out the various parts of their project. While it is inevitable that the schedule will change, it is nonetheless helpful to have a plan instead of just doing work as necessary.

Another important lesson is to pick a good team. Working with trustworthy and dependent people is crucial to having a successful and enjoyable project. In addition, having a good team dynamic makes it easier to adapt to changing project requirements and unexpected surprises.

Finally, it is absolutely important to start early. Initially, much research had to be done into modeling a heart, and work went slowly. However, because this initial research started in September, there was sufficient time to complete the project by April.

6 Equipment, Fabrication, and Software Needs

Apart from software available within Penn Engineering, we will need the following items for this project:

- Simulink HDL Coder
- Altera/Terasic DE0-Nano FPGA Board
- Sparkfun FTDI Basic Breakout Board (3.3V)
- Toshiba TLP-624 Opto-isolation IC
- LF347 Quad JFET Operational Amplifier IC
- Boston Scientific Altrua S606 Pacemaker

Simulink HDL Coder will enable us to machine generate HDL and conform to our model-based development principles. See Section 3.3.2 for more information on this subject. We have also selected an FPGA board for this project, the Altera/Terasic DE0-Nano. This board satisfies our capacity constraints while within a low cost and low power envelope. See Section 4.1.1 for more information on this selection.

7 Conclusion

1. A virtual heart model was implemented on an the Altera/Terasic DE0-Nano FPGA board with minimal resource consumption and approval from the input of a cardiologist. These results meet our propsed goals of achieving an accurate heart model that is extensible. With the resource consumption at around an average of 22%, the heart model can be extended to four or five times the complexity without moving to a new board.

2. The implementation of the heart model was designed to be an automatic process, thus ensuring our desired goal of error-reduction. Theoretical models at a high level of abstraction were created and HDL code for hardware synthesis was automatically generated. Therefore, any changes made at the model level prograte all the way down to hardware. This model can be expanded to include increasingly greater levels of complexity in the heart. This is a portion of the project that can be expanded upon in the future.

3. The heart model was successfully interfaced with a real pacemaker. This result demonstrates the desired goal of building a testing platform by treating the pacemaker as a blackbox. Simply by configuring heart I/O of the pacemaker I/O, the two are able to talk with each other. Furthermore, an important principle of opto-isolation was implemented so that the two systems are electrically isolated, thus reducing risk associated with cross-talk. Future work would include interfacing with more than one pacemaker to demonstrate universality.

4. A user interface was implemented to meet our goal of making our system accessible. The interface allows the user to configure the heart model intuitively simply by changing parameters. As a proof of concept, three preset heart models were implemented. In addition, controls were added on the interface to demonstrate that future custom models are intuitive to implement. Furthermore, the user is able to visualize the heart and pacemaker outputs graphically to ensure correctness of pacing. Ideally, all timing parameters should be exposed to the user interface. Right now, only two parameters are exposed for our demonstration. Future work could involve including an advanced pop-up menu with all the parameters. The number of these paramters would needed to be extended as the heart model as is extended.

5. Finally, to demonstrate the usability of our system, a pacemaker error was induced and caught. An algorithm implemented on the pacemaker to correct for a known error was turned off and the pacemaker was then subjected to testing. By showing our system catches this error, we demonstrated that our system can be confidently used for future pacemaker testing.

8 Nomenclature

Bradycardia	an abnormally slow heart rate.
BRAM	Block RAM. An FPGA component used for data storage.
CLB	Combinational Logic Block. An FPGA component used for constructing logic.
EFSM	Extended Finite State Machine. Finite state machines with arithmetic trigger and state operations.
EGM	Electrogram. A display of electric potentials of heart tissue from internal probes.
Electrophysiology	The study of electrical phenomena related to tissue.
ELT	Endless-Loop Tachycardia, a form of pacemaker-mediated tachycardia that is the result of continuous pacing
FPGA	Field Programmable Gate Array. A device that allows the prototyping of digital logic through 'programmable' hardware.
FSM	Finite State Machine. A mathematical abstraction of a system using the concepts of states and transitions.
HDL	Hardware Description Language. A grammar for describing combinational and sequential logic.
IOB	Input/Output Block. An FPGA component used for interfacing.
I/O	Input/Output
LUT	Look-up Table. A truth table implementation FPGAs use for implementing arbitrary combinational logic.
Node	A cluster of cardiac muscle cells with similar conduction properties.
Op-amp	Operational amplifier
Path	An abstraction of signal timing delays in the heart.
PMT	Pacemaker-Mediated Tachycardia, a form of tachycardia that is caused by the pacemaker overpacing.
PVC	Premature Ventricular Contraction, an "extra" ventricular beat that is natural in some patients.
RAM	Random Access Memory
Simulink	Software included in the MATLAB suite used for the modeling and simulation of dynamic systems.
Stateflow	Software included in the MATLAB suite used for the modeling and simulation of extended finite state machines.
Tachycardia	an abnormally fast heart rate.

9 References

[1] Z. Jiang, M. Pajic, and R. Mangharam. Cyber-physical modeling of implantable cardiac medical devices. *Proceedings of the IEEE*, PP(99):1 –16, 2011.

[2] Chloe Taft. CDRH Software Forensics Lab: Applying Rocket Science To Device Analysis. `http://www.medicaldevicestoday.com/2007/10/cdrh-software-f.html`, October 2007.

[3] Food and Drug Administration. Premarket Approval (PMA). `http://www.fda.gov/medicaldevices/deviceregulationandguidance/howtomarketyourdevice/premarketsubmissions/premarketapprovalpma/default.htm`, September 2010.

[4] Goldman, B.S., E.J. Noble, J.G. Heller, and D. Covvey. The pacemaker challenge. *Canadian Medical Association Journal*, 110:28–31, January 1974.

[5] Widmaier, Eric P., Hershel Raff, Kevin T. Strang, and Arthur J. Vander. *Vander's Human Physiology: the Mechanisms of Body Function*. McGraw-Hill Science Engineering, 2007.

[6] Robert M. Keller. Finite-State machines. `http://www.cs.hmc.edu/~keller/cs60book/12%20Finite-State%20Machines.pdf`, September 2001.

[7] Kwang-Ting Cheng and A.S. Krishnakumar. Automatic functional test generation using the extended finite state machine model. In *Design Automation, 1993. 30th Conference on*, pages 86 – 91, june 1993.

[8] Teresa Chay. Your Cheatin' Heart. `http://www.psc.edu/science/Chay/Chay.html`.

[9] Denis Noble. Modeling the heart–from genes to cells to the whole organ. *Science*, 295(5560):1678–1682, 2002.

[10] Xilinx. *Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet*, 2011.

[11] Digilent FPGA boards. `http://www.digilentinc.com/Products/Catalog.cfm?NavPath=2,400&Cat=10&FPGA`, November 2011.

[12] Seymour Furman and John D. Fisher. Endless loop tachycardia in an AV universal (DDD) pacemaker. *Pacing and clinical electrophysiology*, 5(4):486–489, July 1982.

10 Bibliography

Chow, Anthony W.C. and Buxton, Alfred E., Implantable Cardiac Pacemakers and Defibrillators: All You Wanted to Know. Malden, MA: Blackwell, 2006.

Dahms, D.F., "Implantable Pacemaker Testing Guidance," 1997,
http://www.fda.gov/downloads/MedicalDevices/
DeviceRegulationandGuidance/GuidanceDocuments/UCM081382.pdf

Fletcher, C., "Verilog: always@ Blocks," 2008,
http://inst.eecs.berkeley.edu/~cs150/Documents/Always.pdf.

Fogoros, R.N., Electrophysiologic Testing. Malden, MA: Blackwell, 2006.

11 Financial Information

11.1 Budget Rationale

We require an FPGA board to implement the virtual heart model. We selected the Terasic Altera DE0-Nano Board. Initially, we thought we would require analog-to-digital converters and digital-to-analog converters in order to interface with the pacemakers, but in the end, we built an interfacing circuit instead. This interface circuit was etched onto a printed circuit board for the final product. Finally, we needed opto-isolator and operational-amplifier integrated circuit chips, readily available from the Detkin Laboratory at Penn.

11.2 Itemized Budget

Table 5: Itemized budget

Item	Vendor	Quantity	Unit Price	Subtotal
FPGA Board	Terasic/Altera	1	$100	$100
FTDI Chip	Sparkfun	1	$15	$15
Printed Circuit Boards	4PCB	1	$33	$33
			Total	$148

This project is funded by ESE Senior Design.

12 Ethical Issues

The goal of our project is to reduce occurrances of dangerous pacemaker failures, which is a noble goal. However, as with any system, our project has ethical implications for the world. An inevitable result of widespread use of our system is the shift of liability from manufacturers to our testing platform. In today's world, manufacturers pay the hefty monetary price of recovering from pacemaker recalls and the hefty reputation price of having produced a faulty device. These serious consequences of recalls serve as an incentive for manufacturers to put extra effort into producing a robust system. If our system assumes the authoritative role in pacemaker testing, the first-line incentive to produce robust systems diminishes since liability of pacemaker failures would fall under the responsibility of the testing system. Such a consequence of our system can be avoided with firm and graded protocol for testing. Regulators such as the FDA must ensure that standards are enforced at mulitple levels of device manufacturing. For example, there should be evidence of implemented safety measures before pacemakers are even subjected to testing with our platform.

Another ethical issue that crops up when shifting the responsibility of safety from manufacturing to a testing platform is the idea of testing the tester. Our system revolves around ensuring the safety of pacemakers, but this implies that the system itself is infallable. One step taken to mitigate this concern is the use of model based development (See Section 3.3.2) to ensure that the desired behavior of the models is automatically implented from theory all the way down to hardware. Furthermore, this system is built such that a cardiologist must configure it and then hand these models to manufactuers. In this way, our system is actually ensuring that pacemakers do not go out into the market without the input of an expert such as a cardiologist. It is a tool that is inteded to unify experts from various fields to ensure maximum safety of the produced devices.

One potential concern is that the product may be used with complete disregard to its intention – that is, it may be used as a stand-alone testing platform without provided heart models from cardiologists. Explicit guidelines and requirements must be provided with our system to ensure that manufacturers and regulators are deterred from falsly claiming safety of devices using custom heart models without input from cardiologists and design engineers. The successful passing of testing should be accompanied with reports of the specific heart parameters used to test the desired pacemaker features.

PVS could potentially also have economical implications. The price of pacemakers could increase if manufacturers must purchase an additional testing system since this would add to the overhead manufacturing costs. However, as reported in Section 11, our system has a total cost of only around $150. Pacemakers on the market today cost tens of thousands of dollars. Therefore the additional cost from our system is comparatively inconsequential. Furthermore, with the potential of saving millions of dollars in recalls, the savings may be well worth the small additional cost of this system.

Appendix 1: Additional Figures

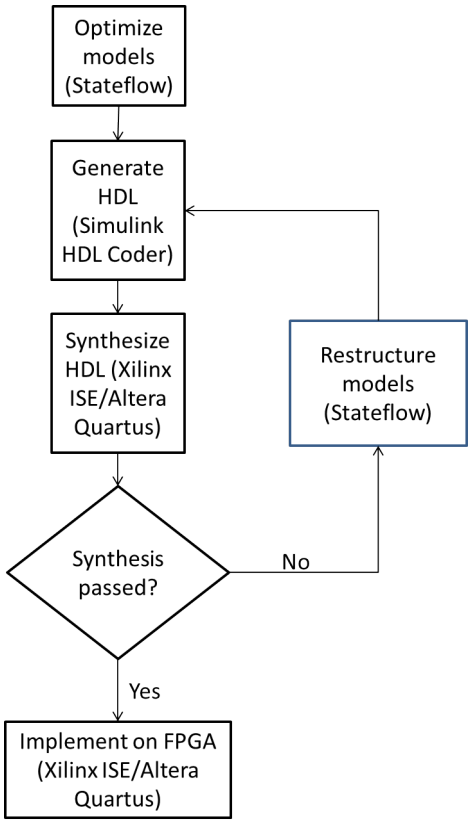


Figure 13: Software Development Process Flowchart

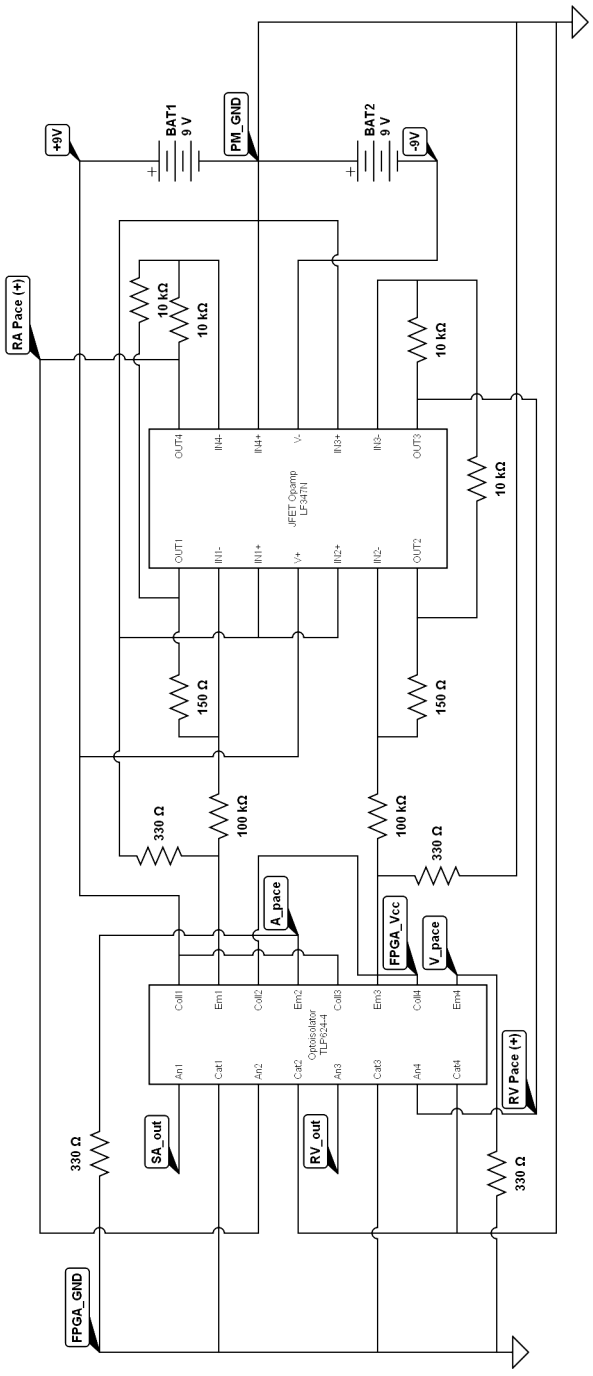


Figure 14: Pacemaker Interface Circuit Schematic

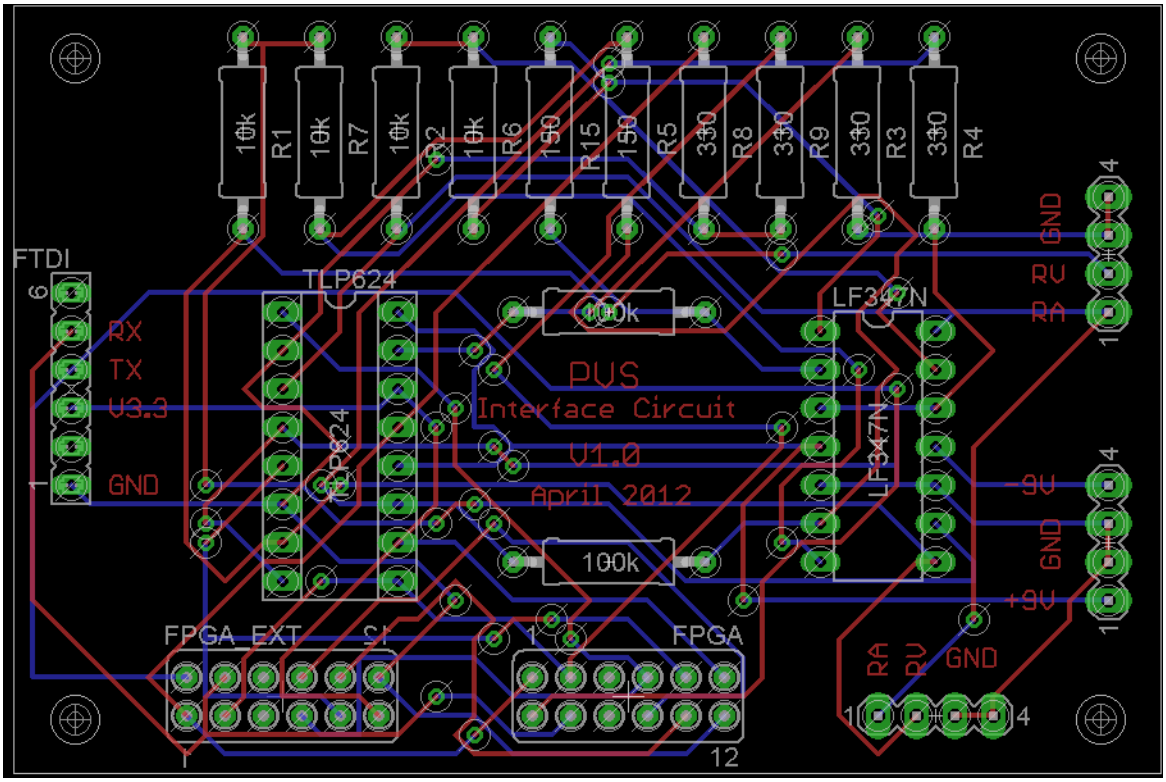


Figure 15: Pacemaker Interface Circuit Layout

Appendix 2: Software and Hardware Code

The Pacemaker Verification System contains two large code modules, one being the hardware description for the FPGA virtual heart model, and the other being the graphical user interface.

A2.1: Virtual Heart Model

The virtual heart model was implemented on a field programmable gate array (FPGA). Since an FPGA implements hardware, as explained in Section 3.3.1, this code is actually written in Verilog, a hardware description language (HDL) that describes how the hardware is laid out. There is HDL for each node and path automaton of the heart model, as well as for the full FPGA build harness (see Section 4.3). Shown below is the HDL representing one node automaton and one path automaton. These two pieces of HDL were automatically generated from the models using the principles of model-based development, as explained in Section 3.3.2.

Node Automaton HDL

```
/*********************************
Pacemaker Verification System (PVS)

Virtual Heart Model
Node Automaton (Sinoatrial Node)

This HDL implements the node automaton.  A similar
block of code exists for each node in the virtual
heart.


Authors:
Sriram Radhakrishnan
Varun Sampath
Shilpa Sarode

Version 1.0
April 15, 2012

*********************************/
`timescale 1 ns / 1 ns

module NA1_SA
          (
           clk,
           reset,
           enb,
           inActive,
           Trest_def,
           Active,
           State,
           path_timer
          );


  input   clk;
  input   reset;
  input   enb;
  input   inActive;
  input   [15:0] Trest_def;  // uint16
  output  Active;
  output  [7:0] State;  // uint8
  output  [15:0] path_timer;  // uint16


  reg  activeMem_out1;
  wire activeMem_out1_1;
  wire node_aut_out1;
  wire [7:0] node_aut_out2;  // uint8
  wire [15:0] node_aut_out3;  // uint16
  reg  inActiveMem_out1;
  wire activeMem_out1_2;
  reg [15:0] path_timer_1;  // uint16
```

```
  assign activeMem_out1_1 = activeMem_out1 | inActive;


  node_aut   u_node_aut   (.clk(clk),
                           .reset(reset),
                           .enb(enb),
                           .inActive(activeMem_out1_1),
                           .Trest_def(Trest_def),  // uint16
                           .Active(node_aut_out1),
                           .state(node_aut_out2),  // uint8
                           .path_timer(node_aut_out3)  // uint16
                           );

  always @(posedge clk or posedge reset)
    begin : activeMem_process
      if (reset == 1'b1) begin
        activeMem_out1 <= 1'b0;
      end
      else begin
        if (enb) begin
          activeMem_out1 <= node_aut_out1;
        end
      end
    end



  always @(posedge clk or posedge reset)
    begin : inActiveMem_process
      if (reset == 1'b1) begin
        inActiveMem_out1 <= 1'b0;
      end
      else begin
        if (enb) begin
          inActiveMem_out1 <= inActive;
        end
      end
    end



  assign activeMem_out1_2 = activeMem_out1 | inActiveMem_out1;



  assign Active = activeMem_out1_2;

  assign State = node_aut_out2;

  always @(posedge clk or posedge reset)
    begin : pathMem_process
      if (reset == 1'b1) begin
        path_timer_1 <= 0;
      end
      else begin
        if (enb) begin
          path_timer_1 <= node_aut_out3;
        end
      end
    end



  assign path_timer = path_timer_1;

endmodule  // NA1_SA
```

Path Automaton HDL

```
/*********************************
Pacemaker Verification System (PVS)

Virtual Heart Model
Path Automaton (SA-RV conduction)

This HDL implements the path automaton.  There
is a similar code block for each path in the virtual
heart.


Authors:
Sriram Radhakrishnan
Varun Sampath
Shilpa Sarode
```

```
Version 1.0
April 15, 2012

*********************************/
`timescale 1 ns / 1 ns

module PA2_2to3
          (
           clk,
           reset,
           enb,
           inActive1,
           inActive2,
           pathTimerEn,
           pathTimerEx,
           forw_param,
           outActive1,
           outActive2,
           state
          );


  input   clk;
  input   reset;
  input   enb;
  input   inActive1;
  input   inActive2;
  input   [15:0] pathTimerEn;  // uint16
  input   [15:0] pathTimerEx;  // uint16
  input   [15:0] forw_param;  // uint16
  output  outActive1;
  output  outActive2;
  output  [7:0] state;  // uint8


  wire path_aut_out1;
  wire path_aut_out2;
  wire [7:0] path_aut_out3;  // uint8


  path_aut_block   u_path_aut   (.clk(clk),
                                 .reset(reset),
                                 .enb(enb),
                                 .inActive1(inActive1),
                                 .inActive2(inActive2),
                                 .forw_param(forw_param),  // uint16
                                 .pathTimerEn(pathTimerEn),  // uint16
                                 .pathTimerEx(pathTimerEx),  // uint16
                                 .outActive1(path_aut_out1),
                                 .outActive2(path_aut_out2),
                                 .state(path_aut_out3)  // uint8
                                 );

  assign outActive1 = path_aut_out1;

  assign outActive2 = path_aut_out2;

  assign state = path_aut_out3;

endmodule  // PA2_2to3
```

A2.2: Graphical User Interface

The graphical user interface was implemented using MATLAB, as explained in Section 4.6. The code shown below handles the real-time plotting functionality of the GUI. This code is called at a regular interval, and plots the heartbeat and pace signals when received from the FPGA virtual heart.

```
/*********************************
Pacemaker Verification System (PVS)

MATLAB GUI
Real-time Plotting Callback Function

This code gets called at a defined, regular
interval, and plots the heartbeat and pacing
signals when received from the FPGA virtual
heart.

Authors:
Sriram Radhakrishnan
Varun Sampath
Shilpa Sarode

Version 1.0
April 18, 2012
```

```
**********************************/

function [ ] = draw_new_point(obj, event, axisObj, serialObj,handles, timerObj)
%draw_new_point - Timer callback function that obtains & plots new point
%from serial
%   Data froms serialObj consists of a 1-byte header and a 4-byte
%   unsigned little endian body that holds a timer value. The header tells
%   what event that counter value is being transmitted for.
%
%   obj and event parameters are required for timer callbacks and simply
%   not used.
%
%   Header values:
%   0 - bad value
%   1 - SA node activation and no APace
%   2 - no SA node activation and APace
%   3 - SA node activation and APace
%   4 - RV node activation and no VPace
%   5 - no RV node activation and VPace
%   6 - RV node activation and VPace

global enableBeat;
global HBsound;
global ErrorSound;

persistent old_SA;
persistent old_AP;
persistent old_RV;
persistent old_VP;

persistent pulses;
persistent iterations;

persistent SA_diff;
persistent AP_diff;
persistent RV_diff;
persistent VP_diff;

persistent ecg_plot;
persistent sa_plot;
persistent rv_plot;
persistent a_plot;
persistent v_plot;

global flush;
persistent isRed;
persistent lastWave;
persistent isAlertMode;

%Axis Values
persistent x;
persistent tix;
persistent ecg_sig;
persistent sa_sig;
persistent rv_sig;
persistent apace_sig;
persistent vpace_sig;

persistent last_clock;


if isempty(SA_diff)
    SA_diff = 0;
end
if isempty(AP_diff)
    AP_diff = 0;
end
if isempty(RV_diff)
    RV_diff = 0;
end
if isempty(VP_diff)
    VP_diff = 0;
end
if isempty(enableBeat)
    enableBeat = 0;
end
if isempty(isRed)
    isRed = 0;
end
if isempty(lastWave)
    lastWave = 0;
end
if isempty(pulses)
    pulses = 0;
end


%% Constants
DEBUG = 1;
NUM_ITER_PER_FLASH = 4;
```

```
%% Position Constants
ylims = [0 15];
apzero = 5.85;
vpzero = 1;
pulselen = 2.5;

if enableBeat
    ecgzero = 11.75;
else
    ecgzero = 11;
end



%% Build PQRST Wave
beat = 3*ecg(400);
p_per = 0.205;
q_per_start = 0.295;
q_per_end = 0.815;

p = beat(1:round(p_per*length(beat)));
qrst = beat(round(q_per_start*length(beat)):(round(q_per_end*length(beat))));

origp = sgolayfilt(p,0,5);
origqrst = sgolayfilt(qrst,0,5);

p = origp + ecgzero;
qrst = origqrst + ecgzero;

%% Initializations
fpgaclk = 1.500; % clock rate is 1.5kHz
dt = 1/fpgaclk; %cache bucket size
c = clock;
c = c(6)*1000 + c(5)*60*1000 + c(4)*60*60*1000 + c(3)*24*60*60*1000;

cacheSize = 7000; %including extra
plotSize = 5000; %plot area

%get serial value if bytes are availble
if serialObj.BytesAvailable || flush
    header = fread(serialObj, 1, 'uint8');
    val = fread(serialObj, 1, 'uint32');
    millisec = val/fpgaclk; %input ms value
    serialRead = 1;
else
    serialRead = 0;
end

shift = 0;

x = 0:dt:cacheSize*dt-dt;

%% Axis Calculations
if flush
    flush = 0;
    tix = (val-(plotSize-1):(val+(cacheSize-plotSize)));
    ecg_sig = ecgzero*ones(1,cacheSize);
    sa_sig = ecgzero*ones(1,cacheSize);
    rv_sig = ecgzero*ones(1,cacheSize);
    apace_sig = apzero*ones(1,cacheSize);
    vpace_sig = vpzero*ones(1,cacheSize);
    isAlertMode = 0;
    iterations = 0;

    hold(axisObj(1), 'off');

    % Initialize plots
    try
    if enableBeat
        ecg_plot = plot(axisObj(1), x(1:plotSize), ecg_sig(1:plotSize),...
            'Color', [0 1 0],'LineWidth', 1);
    else
        sa_plot = plot(axisObj(1), x(1:plotSize), sa_sig(1:plotSize),...
            'Color', [256 170 191]/256,'LineWidth', 3);
        hold(axisObj(1), 'on');
        rv_plot = plot(axisObj(1), x(1:plotSize), rv_sig(1:plotSize),...
            'Color', [0 1 0], 'LineWidth', 3);
    end
    hold(axisObj(1), 'on');

    a_plot = plot(axisObj(1),x(1:plotSize),apace_sig(1:plotSize), ...
        'Color', 'y', 'LineWidth', 3);
    v_plot = plot(axisObj(1),x(1:plotSize),vpace_sig(1:plotSize), ...
        'Color', 'c', 'LineWidth', 3);

    catch err
        disp('failed in plot init');
        flush = 1;
    end

    set(axisObj(1), 'Color', 'k','YLim', ylims, 'XGrid', 'on', ...
        'YTick', [], 'XTick', [0:300:dt*plotSize-dt], ...
```

```
        'XColor', [0.7 0.7 0.7]);

    xlabel(axisObj(1), 'Time (ms)');

else

    % Compute Shift Amount
    shift = ceil((c - last_clock)*1.5);

    tix = [tix((shift+1):end) (tix(end)+1):(tix(end)+shift)];
    ecg_sig = [ecg_sig(shift+1:cacheSize) ecgzero*ones(1,shift)];
    sa_sig = [sa_sig(shift+1:cacheSize) ecgzero*ones(1,shift)];
    rv_sig = [rv_sig(shift+1:cacheSize) ecgzero*ones(1,shift)];
    apace_sig = [apace_sig(shift+1:cacheSize) apzero*ones(1,shift)];
    vpace_sig = [vpace_sig(shift+1:cacheSize) vpzero*ones(1,shift)];
end

last_clock = c;

if serialRead

    %% SA pulse handling
    if header == 1 || header == 3
        SA_diff = val - old_SA;
        old_SA = val;

        if DEBUG
            print_string = sprintf('SA pulse at %d, diff: %d and shift: %d\n', val, SA_diff, shift);
            disp(print_string);
        end

        try
        sa_sig(tix == val) = ecgzero + pulselen;

        if enableBeat
            ecg_sig(find(tix==val):find(tix==val)+length(p)-1) = ...
                ecg_sig(find(tix==val):find(tix==val)+length(p)-1) + origp;

        end
        catch err
            disp('failed in SA');
            flush = 1;
        end
    end

    %% AP handling - plot orange lines on plot 2
    if header == 2 || header == 3
        AP_diff = val - old_AP;
        old_AP = val;


        if DEBUG
            print_string = sprintf('APace at %d, diff: %d\n', val, AP_diff);
            disp(print_string);
        end

        try
        apace_sig(tix == val) = apzero + pulselen;
        catch err
            disp('failed in AP');
            flush = 1;
        end
    end

    %% RV pulse handling - plot blue lines on plot 3
    if header == 4 || header == 6
        RV_diff = val - old_RV;
        old_RV = val;

        if DEBUG
            print_string = sprintf('RV pulse at %d, diff: %d, shift: %d\n', val, RV_diff, shift);
            disp(print_string);
        end

        try
        rv_sig(tix == val) = ecgzero + pulselen;

        if enableBeat
            ecg_sig(find(tix==val):find(tix==val)+length(qrst)-1) = ...
                ecg_sig(find(tix==val):find(tix==val)+length(qrst)-1) + origqrst;
        end
        catch err
            disp('failed in rv plot');
            flush = 1;
        end

        % border color routine
        if pulses >= 4
            BPM = round(60000*fpgaclk/RV_diff);
            set(handles.BPM, 'String', num2str(BPM));
            if (BPM <= handles.cThresh(1) || BPM >= handles.cThresh(2))
                isAlertMode = 1;
```

```
            else
                isAlertMode = 0;
                wavplay(HBsound, 44100, 'async');
            end
        end

    end

    %% VP handling - plot green lines on plot 4
    if header == 5 || header == 6
        VP_diff = val - old_VP;
        old_VP = val;

        if DEBUG
            print_string = sprintf('VPace at %d, diff: %d\n', val, VP_diff);
            disp(print_string);
        end
        try
        vpace_sig(tix == val) = vpzero + pulselen;
        catch err
            disp('error in vp plot');
            flush = 1;
        end
    end

    if header > 6 || header == 0
        print_string = sprintf('Header should not be %d. val: %d\n',header, val);
        disp(print_string);
        return;
    end

    pulses = pulses + 1; %increment for each pulse

end


%% Border Color Alert
if isAlertMode
    iterations = iterations + 1;
    if (iterations >= NUM_ITER_PER_FLASH)
        if ~isRed
            set(handles.figure1,'Color',[216 41 0]./256);
            isRed = 1;
        else
            set(handles.figure1,'Color',[11 131 199]./256);
            isRed = 0;
        end
        wavplay(ErrorSound,44100,'async');
        wavplay(HBsound,44100,'async');
        iterations = 0;
    end
else
    set(handles.figure1,'Color',[11 131 199]./256);
end

try
if enableBeat
    set(ecg_plot,'YData',ecg_sig,'XData',x);
else
    set(sa_plot,'YData',sa_sig,'XData',x);
    set(rv_plot,'YData',rv_sig,'XData',x);
end


set(a_plot,'YData',apace_sig,'XData',x);
set(v_plot,'YData',vpace_sig,'XData',x);
catch err
    disp('failed in X/Y data set');
    flush = 1;
end

xlim(axisObj(1),[x(1) x(plotSize)]);
drawnow;
end
```