Embedded Computer System Design: A Framework

P. David Fisher, Michael Baladi
Michigan State University

Abstract

The area of embedded (computer) systems represents a very fertile framework for electrical and computer engineering students to acquire their *major design experience*. Analog, digital, and mixed-signal technologies continue to evolve at a very rapid pace, with a large gap existing between fundamental topics covered in introductory courses and the integrated knowledge and skills needed by practicing engineers to design embedded systems. Consequently, students involved with design projects that incorporate embedded (digital) computers have the opportunity to learn how to extend knowledge and skills acquired in introductory courses while participating on multidisciplinary teams to formulate realistic solutions to contemporary engineering design problems.

This paper is intended for both faculty and students actively involved in coursework associated with the major engineering design experience. It provides background information on embedded systems that builds upon topics typically covered in introductory electrical and computer engineering courses. It then identifies contemporary design methodologies and design constraints for components and systems that contain embedded computers to monitor and control processes. It also describes and illustrates how many of the standard educational program objectives can be fulfilled when students work in teams on projects involving embedded computers. These include the *major engineering design experience* itself, *multidisciplinary teaming*, *contemporary topics*, and *lifelong learning*.

The paper provides a basic model for embedded systems by first defining the *embedded computer* as a programmable state machine and an *embedded system* as a physical system that contains one or more embedded computers. Such systems often contain sensors, actuators, communication interfaces, user interfaces, and a human operator. The paper then identifies the generic design criteria and challenges that confront the embedded-system designer. These include: *real-time requirements*, *fault-tolerance requirements*, *testability requirements*, *time-to-market requirements*, and *product life-cycle requirements*. These design considerations—coupled with the more traditional design requirements associated with products that do not incorporate embedded computers—are realized by applying an embedded system *design methodology* that emphasizes a *hierarchical design process*, the judicious choice of a *system specification language*, the *reuse of intellectual property (IP)*, and the *co-design of hardware and software*.

I. Introduction

In 1997, Michigan State University introduced a new course *EE 482—Capstone: Computer System Design*[1]. This course was intended to serve as the cornerstone of the major engineering design experience for undergraduate students majoring in computer engineering. The course learning objectives are summarized as follows:

> *Students will learn about embedded systems—i.e., electrical systems that contain embedded computers to control processes. At the completion of this course, each student should have actively participated as a member of an engineering design team and made significant contributions to achieving the team's stated goal and objectives. Each design project should involve the collaborative development and evaluation of a "product" that contains and embedded computer.*

As of 2001, more than 300 students had enrolled in this course. Extremely positive feedback was received about this course from these students; from perspective employers of these students; from former students who had taken the course and graduated; and, finally from the 1998-99 ABET site-visit team[2, 3]. Because of this feedback, the faculty in the Department of Electrical and Computer Engineering voted in 2001 to drop the department's five separate capstone design courses and replace these with a single course that would serve students majoring in both electrical engineering and computer engineering. This new course is entitled *ECE 480—Electrical and Computer Engineering Capstone Course*[4]. It was modeled after the original EE 482 course, with some changes to reflect lessons learned while offering EE 482 each semester over a five-year period.

One of the most important lessons learned since EE 482 was first introduced in 1997 has been that the area of embedded (computer) systems represents a very fertile framework for electrical and computer engineering students to acquire their *major design experience*. Analog, digital, and mixed-signal technologies have continued to evolve at a very rapid pace, with a large gap existing between fundamental topics covered in introductory courses and the integrated knowledge and skills needed by engineers who design embedded systems. Consequently, students involved with design projects that have incorporated embedded (digital) computers have had the opportunity to learn how to extend knowledge and skills acquired in introductory courses while participating on multidisciplinary teams to formulate realistic solutions to contemporary engineering design problems.

This paper is intended for both faculty and students actively involved in coursework associated with the major engineering design experience. The next section of this paper defines key terminology associated with embedded computers and embedded systems. Section 3 provides an overview of the design criteria unique for embedded systems. Section 4 describes the key elements associated with the design methodology for embedded systems. The paper concludes by describing how the embedded-system design projects naturally address many of the *Educational Program Objectives* delineated in ABET's *Criteria for Evaluating Engineering Programs*[5].

II. Embedded Computer Systems

Overview of Embedded System Modeling and Design

Embedded computers and *embedded systems* are terms often encountered in today's world of the electrical and computer engineer. However, these terms are somewhat vague and often misunderstood. This is so because integrated-circuit technologies have evolved at a rapid rate during the last couple of decades. Increased chip complexity and chip functionality—coupled with new product innovations—have helped blur the definitions. In the modern era, nearly three billion CPU's are put to market every year[6]. And the number will only continue to rise at an impressive rate.

One of the foremost questions related to the design of embedded systems is the following: *What should be classified as an embedded computer?* As an example, consider the personal computer (a.k.a., PC), which immediately evokes the following secondary questions:

- Does the PC contain one or more *embedded computers*?
- If so, can the PC itself be viewed itself as an *embedded system*?
- Could the PC itself be viewed as an *embedded computer* if it is incorporated into a higher-level application—i.e., an automatic test system on a factory floor?
- Should something—such as a PC—be classified as an *embedded computer* based upon its deployment within a larger physical system?

These are important questions for today's practicing engineers who design—or specify—new application-specific computers[7, 8]. This is so because these engineers must make critical decisions that ultimately affect product development costs, manufacturing costs, overall product reliability, product life cycle, the creation and/or reuse of intellectual property, etc. We answer these questions in the sub-sections that follow by first providing a generic model for an *embedded computer*. We then use this model to define the concept of an *embedded system*.

Model of an Embedded Computer

From an abstract perspective, we view the embedded computer as a basic system building block (see Fig. 1). Our embedded-computer model incorporates the following important physical/logical attributes.

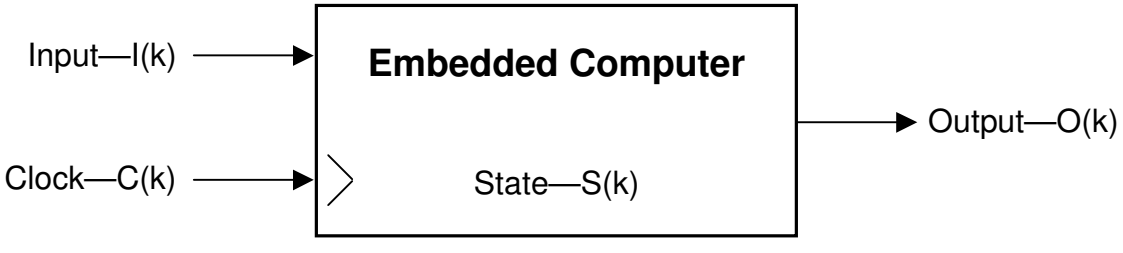

Figure 1. Block diagram illustrating the essential elements of an embedded computer.

Discrete time—*k*: Time advances discretely in our model of the embedded computer. However, discrete time can be mapped into continuous time, and vice versa. For example, one unit of discrete time may equate to a 25-ps interval of physical time. With respect to Fig. 1, the transition between the current and next discrete-time interval is marked by the positive transition of the *Clock—C(k)*. If the current time slot is *k*, then the previous time slot is defined as *k-1* and the next time slot is defined as *k+1*. Although the duration of individual time slots might be equal, this is not a requirement for our embedded computer. Faster is not necessarily better because power dissipation may be a major design consideration. Hence, the embedded computer might perform some of its activities at a relatively low speed or at a relatively high speed. Alternatively, the embedded computer might even be placed in a "pause/sleep mode" to further reduce overall power consumption. In summary, the clock—*C(k)*—may have a constant frequency, a variable frequency, and possibly even a pause mode whereby clock cycles cease.

Input—*I(k)*: The input is a digital data word—comprised of bits of information. The number of bits in *I(k)* depends upon the requirements of the embedded computer.

Output—*O(k)*: The output is also a digital data work—comprised of bits of information. And like *I(k)*, the number of bits in *O(k)* depends upon the requirements of the embedded computer.

State—S(k): The state of the embedded computer may also be modeled as a digital data word, with the number of bits being determined by the overall requirements of the embedded-computer's application.

State machine: If the "current time slot" is *k*, then the "current input," "current state" and "current output" are *I(k)*, *S(k)* and *O(k)*, respectively. If we neglect physical timing parameters—such as input setup time, input hold time, and propagation delay, the present state—*S(k)*—and next output—*O(k+1)*—can be defined mathematically as a Mealy Machine[9] as follows:

$$S(k) = f[S(k-1), I(k-1)] \tag{1}$$

and

$$O(k+1) = g[S(k), I(k)] \tag{2}$$

This set of equations defines the basic properties of a generic *state machine*. The embedded computer is a state machine; however, it has one additional important attribute.

Programmability: The state machine is user programmable. By this we mean that the set of allowed states—as well as the transition between specific allowed sates—can be modified to meet the specific needs of its intended application. This programmable state machine can be modeled as having both hardware and software and, hence, represents what is often referred to as a *computer*. However, we are very careful not to assign any more specific physical or logical characteristics to this computer, other than those illustrated in Fig. 1 and described above.

Embedded computer: If this programmable state machine is embedded within larger physical system—i.e., it is not an isolated entity but rather is an integral component of a larger system—we refer to it as an *embedded computer*.

In summary, *embedded computers* are user-programmable state machines that may be programmed to perform some useful task(s) within some larger physical system. We refer to this larger physical system as an *embedded system*[10] and present a generic model in the next section.

Generic Model of an Embedded System

The embedded system illustrated in Fig. 2 contains five distinct sets of components—i.e., *application*, *embedded computer*, *sensor*, actuator, and *human*[11].

Application: The embedded system has a purpose, and we refer to this purpose at the application. As integrated-circuit technology advances, new embedded-system applications become feasible. Only a partial list is provided below:

- To control an elevator in an office building;
- To control the breaking action of an automobile;
- To control the takeoff/landing of an airplane;
- To authenticate the identity of an individual who is in the process of making a banking transaction;
- To perform quality-control inspections on an assembly line;
- To monitor wear of the bearings in a large rotating machine.

The six applications listed above all related to two basic classes of embedded-computer usage—i.e., to process information and to control processes.

Sensors: Sensors acquire information regarding the behavior—i.e., state—of the application and translate this information into data sets that can be processed by the embedded computer. The sensors may be simple transducers or themselves complex embedded systems. Sensors in embedded systems might be used:

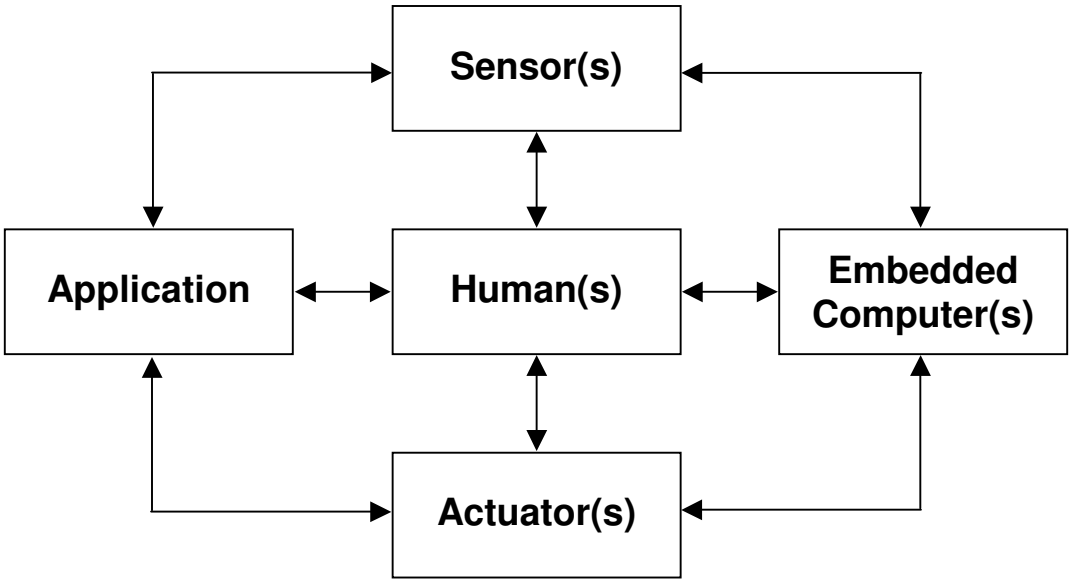

Figure 2. Model of an embedded system.

- To monitor the heartbeat of a person wearing a pacemaker;
- To monitor the acidity of a cleaning agent used in a manufacturing process;
- To monitor the temperature/humidity inside a greenhouse;
- To monitor the separation distance between two objects—e.g., a commercial airplane and the ground.

Actuators: Actuators—a.k.a., effectors[11]—perform the inverse function of sensors. They convert "…an electrical signal from the computer's output to a corresponding physical action that controls an application's function[11]." For example, actuators might be used:

- To increase or decrease the ambient light intensity in a room;
- To change the speed of a rotating machine;
- To lock a door in a bank;
- To power down an inactive PC to a standby mode to conserve power.

Humans: Humans play a central role in embedded systems and are, therefore, identified in Fig. 2 as the central element in our model of an embedded system. Humans have critical roles throughout the lifecycle of an embedded system.

- They design the embedded system.
- They implement it.
- They operate it or at least oversee its operation.
- They troubleshoot/maintain it.
- They modify/upgrade it.

Recognizing these diverse roles at the outset of the embedded-system design process enables the design engineers to build security, safety, testability, fault-tolerance, fault-avoidance, etc. into the designed system. For example, an "operator" of an "embedded-system application" on the factory floor may inadvertently change a parameter that might lead to defective parts being produced or catastrophic failure of the entire manufacturing process. Design engineers need to understand these possibilities and then design the hardware/software in such a way that the "operator" does not have the ability to change these critical parameters. As illustrated in Fig. 2, humans play a central role in monitoring the application; installing and maintaining the sensors and actuators; configuring and interacting with the embedded computer.

Embedded Computers: With the model of the embedded computer in hand (see Fig. 1) and its placement in the embedded system (see Fig. 2), we are now able to understand the relationships between the physical/logical characteristics of the embedded computer and the rest of the embedded system. The embedded computer has three basic purposes, as illustrated in Fig. 2.

- It monitors—through the sensors—the state of the application.
- It changes—through the actuators—the state of the application.
- It provides an interface for humans to monitor the application and to supervise all aspects of the operation and maintenance of the sensors, actuators, embedded computers and application.

III. Design Criteria for Embedded Computers and Systems

Although the generic models of an embedded computer and an embedded system appear to be quite simple, the design criteria and design methodology for implementing them is not. This is so because of the wide variability of embedded-system applications.

Based on the intended application there is a certain level of functionality that must be met in order for the embedded system to successfully complete it's given tasks. There will also be physical constraints to consider. For example, inside of an automobile, an embedded system will have a given shape and size that is mandated by other factors. Other physical requirements include environmental factors such as temperature, resistance to vibration, and weight. Some systems may have power concerns, especially portable applications. All of these constraints must then fit within a given cost that is pre determined. The usual relationship between these is that as size and performance go up, the cost also goes up. A balance must be reached that will satisfy the performance and physical requirements, while maintained an adequate cost.

Often times, simply meeting these often-cited design requirements will not provide an adequate product life for a newer and more advanced product to be developed and brought to market as a replacement. The product life cycle must include the end of one product with the introduction of a new product. This often includes the requirement that the process of generating a new product also leads to the development of new intellectual property (IP) and new markets for the company's core business.

These are many of the general issues that routinely design engineers. However, there are additional ones that relate more specifically to the design of application-specific embedded computers and embedded systems. These are identified and discussed in the sub-sections that follow.

Real-time Criteria

Real-time systems distinguish themselves from other systems by the involvement of time and system correctness[13]. Embedded systems are used in a wide variety of applications. Some of these are very trivial, such as the system in a desktop digital clock. If for some reason the chip inside a digital clock controlling the LCD display didn't meet its timing requirements, the worst-case scenario is that the digits flip to the new time reading a little late. This normally would have no diverse affect on anything outside the system. Other embedded systems however are in very critical applications. Some examples of this are power plant controls, embedded tactical systems, flight mission control systems, and traffic control systems.

The difference between these types of real-time systems is how strict the requirements are for meeting the application-imposed timing criteria. A hard real-time system is one in which meeting the deadline is absolutely necessary[13]. A nuclear plant system or avionics controller would be an example of this. On the other side, a soft real time system won't necessarily have a failure or critical problem arise from missing a timing deadline. It may even be possible that a missed deadline simply causes the system to wait a little longer for the results, in essence rescheduling operations to accommodate for the delay. Under the right circumstances however missed deadlines do still have the potential to cause system instability. An example of a soft real time system would be a video conferencing system. Missing a deadline could cause the video to have

distortion or go dead temporarily, however the service may still continue. The importance of missed deadlines in this scenario is more a factor of what quality of service is desired[13]. The most important design concern in a real time system is predictability.

A hard real-time system must be able to guarantee predictability in regards to meeting its timing deadlines. A deadline is a timing constraint that is typically associated with an event. More important in regards to meeting timing deadlines is consideration of the worst case. Predicting worst-case timing events is critical in system modeling. Some other key issues are data access policies with regard to shared resources. Other systems may be utilizing shared resources, which could prevent a system from meeting a timing deadline. Priority assignment to appropriate threads running on a processor is also important. These are many of the concerns existing today in real time systems.

All of these must be accounted for when an embedded computer is modeled in software. Testing software and routines should be able to identify and investigate issues with worst-case timing and system predictability. Moving beyond the ability to meet a timing constraint, a system must also be able to deal with a potential failure. If a timing deadline is missed in a hard real time system then the system must be able to continue operation and recover from this miss. To stay stable, the system must identify the problem and either correct it or ignore it so that the system does not crash. A lot of the methodology behind this is derived from fault tolerance.

Fault-tolerance Criteria

There are many reasons why fault tolerance is needed in embedded computer systems. Real-time systems need to be fault tolerant so that in the event of a missed deadline the system can remain stable and recover without having any down time. Timing problems however aren't the only reason for a need for fault tolerance. Embedded computer systems such as those in automobiles need to be fault tolerant so that users can be notified of problems before they cause a critical failure that may cause the loss of life. In manufacturing, the increasing use of robots for mass production has increased the need for reliability and fault tolerance[14]. In some sense the addition of fault tolerance however works against itself, as more hardware must be added which itself can also fail. Reliable embedded computers include many mechanisms for system stability including self-checks, and self-diagnosis[15].

The ultimate goal of a fault tolerant system is to maintain stability through a fault until a repair can be made. If a faulty component is detected, a system may activate a backup unit, or simply deactivate the unit if it is non-critical to system operation. Fault tolerance in today's applications however is difficult to implement because they carry high performance penalties with them[16]. Large complicated embedded computers may simply have too much hardware to be able to monitor everything. The easiest systems to monitor for error detection are data paths. These can be monitored using parity or more advanced error detection techniques. On detection of an error a system may ignore the data, or try to process the data again.

Control paths are more difficult to monitor. Today, it is not plausible to create fault-tolerant, single-cycle control paths. Error detection and correction must be done on the next state, which could provide an impact to real time systems that have timing constraints. In the design stages, engineers must work with management teams to determine what level of fault tolerance a system will need. Fault trees and Markov models can be used to evaluate the cost effectiveness of fault

tolerance in designs. These models aren't completely accurate, however, since it is difficult to model the probabilities of failure in different environments and system configurations. For this reason, there is a great deal of research being put into modeling fault tolerant systems. One approach is the development of fuzzy Markov models. These models attempt to create probabilities for probabilities to estimate possible potentials for failure. Fault tolerance offers a very rich field of work in developing hardware and software solutions, as well as the ability to model error detection routines and solutions.

Testability Criteria

With the increasing use of embedded computer systems in every day items it is expected that they are more and more reliable and cost efficient. A typical user expects that a product purchased will function correctly and reliably. For this reason manufacturers are beginning to include testability features, which in the past were seen only in high-end expensive systems. Testing techniques vary depending on the type of circuit in question, however there are many common threads. In the field of Programmable Logic Devices (PLD's) the most typical circuit is the Field Programmable Gate Array (FPGA)[17].

FPGA's are used as logic circuits and to store memory. As with most circuits however, the advances in technology that makes newer FPGA's attractive for design solutions make them less reliable at the same time. Radiation affects smaller feature sizes more and larger die sizes are more susceptible to interference radiation. Common faults in FPGA circuits are stuck on faults, in which a cell may always output a 1 or 0, and design errors, which can cause erratic behavior under certain input conditions. Testing can be accomplished through internal resources or external hardware. External hardware must take the FPGA offline, however, to perform its test and functionality is temporarily lost. Internal resources are capable of online testing by slowly roving over the chip testing units one at a time when they are not in use. The FPGA as a whole, however, remains online and ready for use.

Memory faults are categorized into the following groups. Parametric faults are related to design errors and may be output levels being too low or inadequate fan out capability. Functional faults include pattern sensitive, stuck at, and coupling faults. Pattern sensitive faults only emerge when certain inputs are applied. Stuck at faults are cells in which the output is always a 1 or 0. Coupling faults occur when certain conditions inside the circuit cause a bit to incorrectly flip. Looking away from FPGA's to other circuits, Built in Self Test (BIST) is a common term used for internal hardware testing. This capability can be included on many systems including microprocessors. The types of faults that BIST units look for are permanent faults, intermittent faults, and transient faults[18]. Different methods however are needed to detect these different types of faults. Permanent faults remain in existence all the time. These faults can be detected through the use of concurrent testing over time. Concurrent testing happens while the chip is in operation. It is very good at detecting errors however can take a lot of time to isolate the fault causing the error. Intermittent faults happen some times and disappear at other times. Sometimes they are influenced by environmental conditions however the effects of the fault are highly correlated. Transient faults are very similar however there may be no correlation between the detected error and the fault. These types of errors are best detected by non-concurrent testing. Non-concurrent testing is either initiated by events, or on a timetable. An event may be system startup or shutdown. In order to isolate an intermittent fault it is essential to perform periodic testing.

The level of BIST put on an embedded system must be determined from a system design and marketability perspective. Extra hardware for system testing can increase the cost of the system. There are other affects of BIST however that impact system performance. As the error coverage of BIST hardware increases, the error and fault latency, space and time redundancy also increase. Error and fault latency is the amount of time required to detect an error or fault after its onset. Space and Time redundancy refers to the extra hardware and time needed for online testing. Self-testing requires the use of resources within the chip. The more time it takes a chip to perform a self-test the more clock cycles are lost. In an ideal situation, a BIST process would have 100 percent error coverage, 1 clock cycle of latency, and no space or time redundancy. The AMD-K6 is an example of a chip that uses many advanced BIST techniques to lower design costs[19]. The K6 is a fully scannable microprocessor with support for boundary scanning of I/O cells, and JTAG IEEE 1149.1 compatibility. Inside the K6, every functional block contains a BIST routine. AMD is not the only company leading the way of implementing BIST. Embedded computing covers a wide range of fields such as automotive or household items. As these systems all become more complicated they are all receiving more and more advanced BIST techniques to lower overall costs of design and in field implementation. Research in techniques to more effectively test circuits while saving space and time is constantly ongoing.

IV. Design Methodology

With some of the concerns regarding design criteria addressed some of the methodologies that exist in the design process can be investigated. Similar to most of the design criteria, design methodology is an evolving science. For every process that already exists, there is another in development to replace it. Even with all the progress that has been made in embedded computing technology, there are still a lot of deficiencies in the design process. Methodologies are needed to simplify the design process. With established methods for design, companies save time and money, which is essential to the making of a profit and the turnover of good product. Time to market is constantly getting more and more difficult to meet as competition in the embedded computer realm increases with more demand.

Hierarchical Design Process

The design process of an embedded computer system will ultimately make or break a design. For example, the design process will affect the time-to-market, the reliability of the design, and the ability to re-use portions product (IP) previously designed and tested. There are several components necessary to a successful design cycle. Organization of the design steps comes first. It should be easily understandable from many other life experiences that a successful design is one that is planned well. When designing a system of any kind, one should first develop a clear and concise specification of what is to be designed. Included in this specification are all requirements of the system including but not limited to physical, performance, cost, and environmental constraints. When writing this specification, it must be usable by any design team. For this reason we need a system specification language.

There are many different types of specification languages on the market for different needs. VHDL and Verilog are hardware languages[20]. They can be used to create simulations of hardware and describe behavioral models. Assembly and C are example of software languages. As a specification of the system is developed, the pieces of hardware in hardware specification language can be packaged as reusable IP blocks[21]. Different manufacturers can reuse these

blocks in different designs. Sometimes IP blocks are called virtual components or cores[22]. One challenge when working with cores is interfacing between them when the cores are written in different specification languages. This demonstrates further the need for organization in the design process to minimize communication errors between software packages.

To further aid in organization of the design process are hardware-software co-design packages. These help in the management of features between hardware and software. Through the use of specification languages, IP cores, and co-design packages it is possible to minimize design cycle times and get products to the market faster. Competitiveness however keeps the development of new techniques for organized hierarchical design critical to keeping a competitive edge.

System Specification Language

In the design process things must be simulated and coded at some point. It is important that standards for this process be developed that can adequately handle the variety of different components of embedded computer system design. In current system specification language theory there are different ways to model systems[23, 24]. State Oriented models can be finite state machines, Petri-nets, or hierarchical concurrent finite state machines. These models use a set of states and show the transitions between them. They are mainly used in modeling systems that react to external input or events. Other methods of modeling include activity-oriented models such as data flow graphs and flow charts. Often used in digital signal processing, they show a set of activities and their relation to data and control dependencies. Structure oriented models such as component connectivity diagrams usually show the physical modules themselves and the interconnections between them. Entity-relationship diagrams are a form of data-oriented models, which show the relationships between data that is processed by the system. Finally there are heterogeneous models such as control data flow graphs and program-state machines. These models in essence combine many of the features of all the different models already mentioned to form a large system model.

> *The overall goal of a System Specification Language is to describe non-ambiguously the desired functionality of a system.*[24]

System specification languages include telecommunication, real-time system, hardware description, programming, parallel programming, and data-flow languages. They are used in the modeling and description of nearly all types of embedded systems and computers. There are many languages that have already been developed however many still have shortcomings that need to be developed. Lotos, SDL, and Estelle are telecommunication languages. Esterel and StateCharts are both real-time system languages. Hardware description languages include VHDL, Verilog, SpecCharts and HardwareC. C^X and SpecC are programming languages and CSP and Occam are parallel programming languages. Lastly, Silage is a data-flow language. The reasons for having a fully developed and functional specification language are so that standards are present for the development and modeling of embedded computer systems. This ties in closely with other design methodologies such as Hardware-Software Co-design and Intellectual property reuse. With standards in place, this allows exchangeability of modeling code with other companies and organizations. Work continues in the developing of better specification methods and modeling techniques. There is a lot of opportunity available in this field.

Hardware-software Co-design

The goal of hardware-software co-design is to manage the integration of software and hardware to make a functional product. Embedded computers are in nearly all cases mixed hardware/software systems. A co-design process defines the hardware and software partition and their interaction to make a final product. In the design process both hardware and software can be used to achieve the project goals. Hardware is usually faster and better at meeting timing constraints. Some cons of hardware however are redesigns becoming very costly, and user updates being difficult and expensive. With software, features are easier to add, the design cycle is less expensive and shorter, and in field updates are possible. Cons of software are the ability to meet timing requirements. The traditional design process involves software and hardware teams working independently to design their corresponding components[25]. A hardware team will work on designing a microprocessor, while the software team may be programming the code to run on the microprocessor. During the design phase, little interaction may happen, and no testing is possible until finished product becomes available.

With a co-design package, some of these challenges are diminished or eliminated. The software package will now handle the partition of what's in hardware and what's in software. Design teams will know exactly what features need to be coded and what are designed for in hardware. As design changes are made and different tasks are handed off the co-design package will track these changes and ensure that all features continue to be accounted for. Hardware-software handoffs are also tracked. This insures that during the design phase, fewer errors are made in that the software is designed for the hardware and vice versa[26]. With fewer total errors during the design process the cost of development is lowered significantly. System debug and test is also easier. The debug stage normally can't begin until finished products start to return from fabrication. With a co-design package working inside a system specification language the debug of a system can begin without physical hardware being present.

Through the use of behavioral models, software can be simulated on the developing hardware and testing can be completed sooner, speeding up the time to market. These behavioral models will then become part of the intellectual property package that will be developed and perhaps sold to other companies for use in their products. These models can be used in other design processes to verify that a developed piece of property can be used in another design[27]. Advantages of hardware-software co-design extend into the investigation of product upgrade possibilities and in-field debug of current products. Software and hardware behavioral models can be simulated together to investigate problems and try solutions before actual changes are made.

Existing co-design packages include SystemLab[28], The Chinook Project[29], POLIS[30], and Cadence[31]. Each of these packages has strengths and weaknesses. A common weakness is nearly all packages are accurate simulation of software running on hardware. Other challenges in current software are the ability to easily represent design models in a way that's easily comprehensible. The opportunity for development and growth in co-design is large with these companies. Cadence has cited a 50% increase in digital-analog system design in the coming years. Digital-analog design is a new and developing aspect of hardware-software co-design. As previously mentioned the design of behavioral models in the co-design methodology leads into the use of intellectual property.

Reuse of Intellectual Property

Intellectual Property (IP) has grown to be of great value in today's market. Without IP reuse the rate of development for new chips would be impossible to maintain. The typical designer at best can code one thousand lines of code per month to generate up to ten thousand logic gates. The average semi conductor has well over 5 million gates, with some pushing far more then this[32]. If designers generate half of the 5 million gates then 20 engineers would be needed to work on the integrated circuit and complete it in a one year design cycle. This means that IP reuse is needed to complete a circuit within a tight design cycle. Many large companies have large libraries of IP that are reused in circuits to complete design objectives. When designing a circuit with five, ten, or even more transistors, it's going to have to be done in bigger blocks, not individual gates[33].

Most IP can be categorized into two sections—i.e., System IP and Component IP. System IP is involved with the ability for system integration, such as partitioning and architecture. Component IP is a stand-alone unit that completes a given function, such as a memory design. Despite the importance of reusing IP, it is also important to develop new IP. The lively hood of many companies exists in the creation and marketability of IP. A large company may sell parts of its libraries to other design teams. Some IP manufacturers may develop IP strictly for use in other company's designs and not for their own use. The Virtual Socket Interface Alliance (VSIA) made a prediction in the past that over a 10-year period, the average rate of IP reuse in design would go from 10% to 90% in any given chip. Along with this increase in reuse, the number of transistors used on any one chip would increase by a factor of 50. An important decision for many companies will be how much reuse? Should additional libraries be purchased, or new circuits researched? When developing libraries, three important factors to any given component must be fulfilled. These are reuse documentation, the library component, and verification support[34]. Necessary to the component's architecture is to account for the potential for reuse. This involves anticipating components incorporation into a design. Making it as easy as possible to hook up and test circuits functionality within another design is not to be taken lightly.

There are many IP reuse tools that are already developed for today's use. Again, however, there are many shortcomings and items still in development. Coware N2C Design System has the ability to capture both hardware and software architectures[35]. Using C or C++ as a system language allows for hardware and software tradeoffs to be made easily. Mentor Graphics, another large IP company, provides a large array of reusable cores[36]. All their soft cores are available in Verilog or VHDL for incorporation into designs. Development of IP reuse is essential to the progression of complex designs and keeping down design costs and cycles. The faster a product can be brought to market the more profit that can be made from it.

V. Discussion

Student projects that require the use of embedded computers represents a very fertile area for electrical and computer engineering students to acquire their *major design experience*. Students have the opportunity to extend knowledge and skills acquired in a wide variety of technical and non-technical introductory courses while participating on multidisciplinary teams to formulate realistic solutions to contemporary engineering design problems. Students have the opportunity to integrate knowledge and skills needed by practicing engineers to design embedded systems.

Major Design Experience: Projects that require the use of embedded-computer technology naturally fulfills many of the requirements of the major engineering design experience. It logically builds upon material contained in introductory courses and requires the integration of this knowledge and skills to design a useful product or system. Physics, chemistry, mathematics, statistics, computer science, electric circuits, electronics, communications, signal analysis and system analysis can be brought together in an integrated fashion to solve the design problem.

Multidisciplinary Teaming: A single student cannot realistically undertake the embedded-system design project. A diversified set of students would collectively possess more knowledge and skills needed to meet the overall design objectives. This would lead to communication amongst team members that would identify best strategies for meeting the design objectives. In addition, the project could not be completed on time if worked on by a single individual.

Lifelong Learning: Students working on contemporary embedded-systems projects will quickly discover that many of the questions they formulate cannot be answered by looking back into material encountered in previous courses. Technological change and the application of this technology advances at a much more rapid pace than does the content of textbooks. Consequently, students need to learn how to acquire new knowledge and skills. Coming to grips with the fact that all answers cannot be found in textbooks is an important step to realizing that problem solving and professional growth requires lifelong learning throughout ones career.

Contemporary Topics: If undergraduate engineering student design projects focus on the use of embedded computer to design a new product or improve on an existing product, students naturally have the opportunity to be exposed to a variety of important contemporary technical and non-technical topics. For example, we discussed in earlier sections of this paper contemporary topics that bridge both technical and non-technical fields—e.g., fault tolerance, safety, security, and the reuse of intellectual property. These contemporary topics can also be addressed through the application itself—e.g., homeland security, improving the quality of life for people with disabilities, improving highway vehicle safety, and improving the health and quality of people.

VI. Summary

We have had more than five years of experience supervising students on senior-level design projects that involve embedded computer systems. An undergraduate engineering program—such as electrical engineering, computer engineering, mechanical engineering or chemical engineering—could become distinctive by virtue of the types of applications students focus on as they seek to fulfill the requirements for their major engineering design experience and other educational program objectives. For example, faculty responsible for a specific academic program might collectively decide to focus student projects in areas related to bioengineering. Students, faculty, alumni, employers of graduates, and long-term support for design projects could use the theme of "bioengineering/biotechnology" as a lightning rod to attract external recognition to the program. Embedded computer systems would be the enabler to help make this happen. Of course, bioengineering is only held out as one possible focus. It would be up to the faculty to decide how they would like to distinguish their academic programs.

VII. Acknowledgments

This work was supported in part by the General Electric Fund through a grant entitled "Reforming the Early Undergraduate Engineering Learning Experience: Phase II" and by NSF through a grant entitled "Enhancing the Bioengineering Opportunities for Engineering Majors."

Bibliography

[1] Rover, D.T., and Fisher, P.D., "Cross-functional teaming in a capstone engineering design course," *Proc. of IEEE/ASEE 1997 Frontiers in Education Conference,* November 1997.

[2] Rover, D.T. and Fisher, P.D., "Student self-assessment in upper level engineering courses," *Proc. of 1998 IEEE/ASEE Frontiers in Education Conference*, November 1998.

[3] Rover, D.T., "Perspectives on Learning in a Capstone Design Course," *Proc. of 2000 IEEE/ASEE Frontiers in Education Conference*, November 2000.

[4] *ECE 480 Course Web Site*, Michigan State University, http://www.egr.msu.edu/classes/.

[5] *Criteria for Evaluating Engineering Programs*, http://www.abet.org/criteria.html.

[6] Koopman, P. "Embedded system design issues (the rest of the story)," IEEE International Conference on Computer Design: VLSI in Computers and Processors, 1996. ICCD '96. Austin, TX. Pp. 310-317.

[7] Aylor, J. & Camposano, R. & Schueete, M. & Wolf, W. & Woo, N. "The future of embedded system design," IEEE 1992 International Conference on Computer Design: VLSI in Computers and Processors, 1992. *ICCD '92. Proceedings*, Cambridge, MA, USA pp. 144-146.

[8] Napper, S., "Embedded-system design plays catch-up," *Computer*, vol. 31, no. 8, 1998, pp. 120-118-19.

[9] Tanenbaum, A. Structured Computer Organization. New Jersey: Prentice Hall. 1999

[10] Wolf, W. Computers as Components. California: Morgan Kaufmann Publishers

[11] Dunn, W.R., "Designing safety-critical computer systems, *Computer*, vol. 36, no. 11, 2003, pp. 40-46.

[12] National Institute of Technology (2003, September). Available: Lichen, Z., Peijiang, Y., "An integrated approach for real-time system design," in *Communications, Computers and Signal Processing, 1999 IEEE Pacific Rim Conference on*, Victoria, BC, 1999, pp. 63-66.

[13] Vasan, S., Rajkumar, R., "A methodology for predictable real-time system design," in *Digital Avionics Systems Conference, 1998. Proceedings, 17th DASC. The AIAA/IEEE/SAE*, Bellevue, WA, vol. 1, 1998. pp. C26/1-C26-8.

[14] Leuschen, M.L., Walker, I.D., Cavallaro, J.R., "Robot reliability through fuzzy Markov models," in *Reliability and Maintainability Symposium*, *Proceedings*, 1998.

[15] Pflanz, M., Vierhaus, H.T., "Generating reliable embedded processors," in *Micro, IEEE*, vol. 18, no. 5, 1998, pp. 33-41.

[16] Avresky, D.R., Lmbardi, F., "Fault-tolerant embedded systems," in *Micro, IEEE*, vol. 21, no. 5, 2001, pp. 12-15.

[17] Shnidman, N.R., Mangione-Smith, W.H., Potkonjak, M., "On-line fault detection for bus-based field programmable gate arrays," in *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 6, no. 4, 1998, pp. 656-666.
[18] Al-Asaad, H., Murray, B.T., Hayes, J.P., "Online BIST for embedded systems," in *Design & Test of Computer, IEEE*, vol. 15, no. 4, 1998, pp. 17-24
[19] Fetherson, R.S, Shak, I.P, Ma, S.C. "Testability features of the AMD-K6 microprocessor," in *Design & Test of Computers, IEEE*, vol. 15, no. 3, 1998, pp. 64-69.
[20] S. Edwards, *Design Languages for Embedded Systems*, Available: October, 2003 http://www1.cs.columbia.edu/~sedwards/papers/edwards2001design.pdf
[21] R. Passerone, J.A. Rowson, A. Sangiovanni-Vincentelli, "Automatic synthesis of interfaces between incompatible protocols," in *Design Autmation Conference*, Berkeley, CA, 1998, Proceedings of the IEEE 1998, pp. 8-13.
[22] P. Coussy, A. Baganne, E. Martin, "A design methodology for integrating IP into SOC systems," in *Custom Integrated Circuits Conference*, LESTER, Univ. de Bretagne Sud, Lorient, France, 2002, Proceedings of the IEEE 2002, pp. 307-310
[23] Gajski, D.D., Vahid, F., Narayan, S. and Gong J. *Specification and Design of Embedded Systems*. Pearson Education POD, 1994
[24] "Embedded System Specification,". Available: October, 2003 http://www.informatik.uni-stuttgart.de/ipvr/ise/Skripte/Prof_B_Folien/HSCD_Folien/HSCD_F03.pdf.
[25] G. Walters, "System hardware/software co-design ensures first pass success," in *Wescon/98*, Anaheim, 1998, pp. 29-35.
[26] M. Lajolo, M. Rebaudengo, M. Sonza Reorda, M. Violante, L. Lavagno "System-level test bench generation in a co-design framework," in *Proc. European Test Workshop*, Cascais, 2000, pp. 25-30.
[27] B. Tabbara, M. Sgroi, A. Sangiovanni-Vincentelli, E. Filippi, L. Lavagno, "Fast hardware-software co-simulation using VHDL models," in *Proc. Design, Automation and Test in Europe Conference and Exhibition*, Munic, 1999, pp. 309-316.
[28] G. Fitzhugh, R. Comfort, "Test and validation of electronic systems," in *Proc. 18th Digital Avionics Systems Conference*, St. Louis, 1999, pp. 10.A.1-1 – 10.A.1-7 vol.. 2.
[29] "The Chinook Project,". Available: October, 2003 http://www.cs.washington.edu/research/projects/lis/www/chinook/.
[30] "Ptolemy,". Available: October, 2003 http://ptolemy.eecs.berkeley.edu/.
[31] "Cadence,". Available: October, 2003 http://www.cadence.com/.
[32] Bricaud, P.J. "IP reuse creation for system-on-a-chip design," in *Custom Integrated Circuits*, 1999. Proceedings of the IEEE 1999, pp. 395-401.
[33] Jacome, M.F.; Peixoto, H.P. "A survey of digital design reuse," in *Design & Test of Computers*, IEEE, vol. 18, no. 3, May 2001, pp. 98-107.
[34] Thomas, T. "Technology for IP Reuse and Portability," in *Design & Test of Computers*, IEEE, vol. 16, no. 4, Oct-Dec 1999, pp. 7-13.
[35] CoWare N2C design system [Online Datasheet] Available: October, 2003 https://www.coware.com/portal/page?_pageid=167,105531&_dad=cust_portal&_schema=STAGE.
[36] Mentor Graphics Inventra [Online Source] Available: October, 2003 http://www.mentor.com/inventra/.

Biography

P. DAVID FISHER is a Professor Emeritus of Electrical and Computer Engineering at Michigan State University. He serves at Project Director and Principal Investigator for the GE Fund-sponsored project: “Reforming the Early Undergraduate Learning Experience.” Dr. Fisher is a registered Professional Engineer in the State of Michigan and is an ABET-IEEE Program Evaluator for EC2000 computer engineering and electrical engineering programs.

MICHAEL BALADI is a Masters degree candidate at Michigan State University. He graduated in 2001 with his Bachelors of Science in Computer Engineering at Michigan State University. Michael Baladi is working under Dr. P David Fisher while completing his degree and has studied Embedded Computing both in class and with Dr. Fisher.