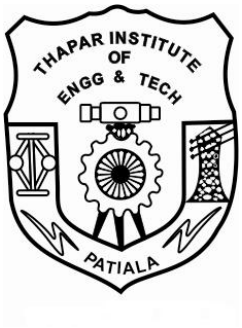
"To Design a Real Time Scheduler for Embedded Systems using Hardware and Software Co design approach"

A Thesis

Submitted in partial fulfillment of the

requirement for the award of degree of

Master of Technology
In
VLSI Design and CAD



By:
Raghu
(6040413)

Under the supervision of:

Miss Navjot Kaur
Lecturer, ECED

THAPAR INSTITUTE OF ENGINEERING AND TECHNOLOGY
(DEEMED UNIVERSITY)
PATIALA – 147004
JUNE 2006

Certificate

I hereby certify that the work, which is being presented in the thesis, entitled **"To Design a Real Time Scheduler for Embedded Systems using Hardware and Software Co design approach"** in partial fulfillment of the requirements for the award of degree of Master of Technology in VLSI Design and CAD at Electronics and Communication Engineering Department of Thapar Institute of Engineering and Technology (Deemed University), Patiala, is an authentic record of my own work carried out under the supervision of Miss Navjot Kaur.

I have not submitted the matter presented in the thesis for the award of any other degree of this or any other university.

(Raghu)

This is to certify hat the above statement made by the candidate is correct and true to best of my knowledge.

(Miss Navjot Kaur)
Supervisor
Lecturer
Electronics & Communication Engineering Department,
Thapar Institute of Engineering & Technology, PATIALA-147004

Countersigned by

(Dr. R.S. Kaler)
Professor & Head
Electronics and Communication Engineering Department
Thapar Institute of Engineering and Technology
PATIALA-147004

(Dr. T.P. Singh)
Dean of Academic Affairs
Thapar Institute of Engineering and Technology
PATIALA-147004

Acknowledgement

It is with the deepest sense of gratitude that I am reciprocating the magnanimity, which my guide Miss Navjot Kaur, Lecturer, Electronics and Communication Engineering Department has bestowed on me by providing individual guidance and support throughout the Thesis work.

I am also thankful to Dr. S.C. Chatterjee, P.G. Coordinator, Electronics and Communication Engineering Department for the motivation and inspiration that triggered me for my thesis work.

I would also like to thank all the staff members and my co-students who were always there at the need of the hour and provided with all the help and facilities, which I required for the completion of my thesis.

I am also thankful to the authors whose works I have consulted and quoted in this work.

Last but not the least I would like to thank God for not letting me down at the time of crisis and showing me the silver lining in the dark clouds.

Raghu

(6040413)

Abstract

Embedded systems can no longer depend on independent hardware or software solutions to real time problems due to cost, efficiency, flexibility, upgradeability, and development time. System designers are now turning to hardware/software co-design approaches that offer real time capabilities while maintaining flexibility to support increasing complex systems. Although long desired, reconfigurable technologies and supporting design tools are finally reaching a level of maturity that are allowing system designers to perform hardware/software co-design of operating system core functionality such as time management and task scheduling that allow the advantages of higher level program development while achieving the performance potentials offered by execution of these functions in parallel hardware circuits.

All event-scheduling functionality was migrated into hardware with a standard FPGA-based address mapped register set interface for the remainder of the operating system. This hardware-based event scheduling functionality liberated the CPU from performing the overhead processing associated with managing event queues, and provided microsecond resolution scheduling of events.

The scheduler component of the Operating Systems was implemented in hardware using VHDL language and simulations were done using Model SIM software and synthesis using Leonardo Spectrum software.

This work represented a critical first step towards achieving a full hardware/software co-design of key operating system functions into a hybrid system for embedded applications.

TABLE OF CONTENTS

List of Figures

List of Tables

CHAPTER 1

Introduction

1.1 The History of Highly Reliable Software

Programming as it is known today was effectively invented in the early 1950s, when the first generation of post-war computers was frustrating the first generation of experts responsible for making the machines complete their assigned tasks. The discovery by Grace Hopper of a moth embedded in the circuits heralded future programmers' frustration in trying to find errors in their programs which had no less obscure causes [22].

1.1.1 Programming vs. software engineering

Programming is simply the act of producing data (a program) designed to be executed by a computer. Software engineering is a wider ranging term. According to standard IEEE definition it's defined as:

The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software [5].

1.1.2 Historical failures

There have, of course, been numerous failures of software engineering. Some of them have been spectacular, such as the Ariane 5 flight control software numeric overflow that resulted in a hundred-million-pound firework display over French Guyana. Others have been hardly noticed by the public, but nevertheless expensive. Repeated efforts to develop next-generation air traffic control system for the United States have met with failure after expensive failure, and the current Standard Terminal Automation

Replacement System (STARS) has slipped by four years and incurred a 60% cost over-run so far. In the meantime, old software is operating far past its intended lifetime. The more serious failures involve human loss rather than financial loss. Remarkably, there are

relatively few fatalities directly attributable to software failure. One of the earliest, and worst, of such accidents was the Therac 25 incident described in. A number of radiotherapy patients received massive radiation overexposure as a result of a race condition within the Therac-25 radiotherapy machine software. Notably, the fault was also present within an earlier model of machine, but a hardware interlock there prevented its manifestation[21].

1.1.3 Where things go wrong

The most common point of project failure is, surprisingly, in the earliest phase: requirements gathering. 30% and 48% of IT projects fail due to requirements-related problems, even though the stage at which the projects fail is usually late in the development cycle. A significant fraction of safety-critical software projects start to go adrift less for technical reasons than for failures of process. The Ariane 5 explosion was traced back to a numeric overflow in the flight-control software, written in Ada. This was the cue for advocates of other languages and tools to leap in and say, “if only you had been using X you would have detected this possible overflow.” However, this misses the point. The relevant section of the software was taken from the Ariane 4 programme. It was not checked as it had been tested for Ariane 4, all known errors fixed, and had established a reliable track record. Ariane 5 flew a faster and tighter flight profile than Ariane 4, and so the numeric exception occurred where before the range of values was within the defined type range.

1.2 Modern Software Development

According to modern definition of software engineering it’s based on the following under given features

1. That system development time does not scale in an inverse-linear relation to team size,

and indeed that adding more manpower to a late project makes it later (the “mythical man-month”);

2. That there is no single development, in either technology or in the management, which promises an order of magnitude improvement within a decade in productivity, or in the reliability or simplicity (“no silver bullet”);
3. That after building one system successfully, the design and development of a follow on system is prone to balloon out with pointless features and an elephantine design (the “second system effect”); and
4. A small number of documents, in a sea of project documentation, become the critical pivots around which every project’s management revolves (“the documentary hypothesis”)[6].

1.3 Interfacing of hardware and software

The bane of a software engineer’s life is when his code is required to interact with actual physical hardware, that is, hardware external to the computer itself; “stepping outside the sandbox”, as it is sometimes called. It is not for nothing that the writing of device drivers for an operating system is regarded as something of a black art. Why is this?

1.3.1 The I/O problem

Taking the Universal Register Machine as the canonical computer, and ignoring for the moment the unlimited memory space that it provides, it might well be believed on first inspection that the machine is useless. It has a list of memory “slots”, each of which can hold an arbitrary natural number. It has an instruction counter, initially set to 1. It operates on a numbered list of instructions, each of which is one of the following:

Z (M) Zero the value in memory slot M

S (M) Increment the value in memory slot M by 1

T (M, N) Copy the value in slot M into slot N

J (M, I, J) if the value in slot M is zero, set the instruction counter to I; otherwise, set it to J.

For any of the first three instructions, once it is executed the machine will increment the instruction counter by 1. In any case, the next step of the machine will be to read and execute the instruction pointed to by the instruction counter. If this counter points beyond the end of the instruction list given, the machine stops. From a black box point of view, the machine does nothing and has no inputs or outputs defined. To give its actions meaning there should be a provision to inspect the memory locations, control the starting of the machine and possibly also feed in new programs. This must be accomplished outside the machine's normal operations. It is a similar situation with embedded systems. A well-established processor typically one of the ARM or PowerPC families – may be coupled via a bus and memory controller to a bank of RAM, and a program executed in the normal way. However, something must start program execution in some way after power-on, and the rest of the system under control (e.g. a water heating system) must be able to feed data to the processor and read control signals out of it. Without heavy customization of the processor, the simplest way is often memory mapped I/O. This technique uses the memory management unit of the system to flag certain locations in the processor's memory map as "special"; the values in those locations may either represent data read from external sensors, or be control values read by and used to control external actuators.

1.3.2 Why the interfacing is hard

The problems posed by such an apparently simple arrangement are many and subtle. The most obvious is a change in the way that anyone reason about program correctness. In the normal programming model any control path, which may write two values to a given variable in succession, without reading the first value back, is immediately suspected of being in error.

The second problem, more insidious, is the lack of synchronization between the software and hardware worlds. Events external to the processor may occur at any point, in any order. Inside the processor bounds can be placed on the number of computational steps between two events, but introducing dependencies on external events complicates the

problem of producing highly reliable software, which is correct with respect to a specification. These problems also occur in systems where there are multiple threads of control with a shared address space. Programming languages have had to develop features such as semaphores, monitors, protected objects and associated protocols to solve these problems[2] [7].

1.4 Problem domain

One of the constant challenges for real-time system designers is building a platform that can meet the timeliness requirements of the system. These requirements include time deadlines for scheduling tasks that cannot be missed. Additionally, the scheduling resolutions of these deadlines are many times at a finer granularity than what commercially available software-based schedulers are able to provide.

Many commercially available operating systems schedule events based on a periodic interrupt from a timer chip, known as the heartbeat of the system. Due to the unique timeliness requirements of real-time events, this heartbeat approach is insufficient in both frequency and resolution. Basing real time scheduling decisions on a periodic scheduling approach can yield unacceptable performance due to the overhead processing associated with context switching times and a periodic interrupt processing requirements. In fact, under certain conditions this heartbeat approach can introduce so much overhead, that the system not be able to achieve any useful computation.

Another problem results from the current approaches in use for reducing the service time of an interrupt service routine once acknowledged. In fact, there has been little research done on measuring and reducing the delay between the time an event is set to be scheduled and the time at which the event actually is executed. This is not a trivial problem as this delay is difficult to measure using software based monitoring approaches. To accurately capture this delay, sophisticated and expensive logic analyzers are required. Even with a sophisticated and expensive analyzer, it is still time consuming and difficult to capture this delay time.

Finally, real-time systems must track and maintain timing information for multiple events that should be serviced in the future. Typically this is done in software with the events stored in a priority queue. Maintaining and sorting the queue introduces time delays and jitter but is still necessary to ensure that the highest priority event will be serviced before its deadline. Sorting through the event priority queues is time-consuming and variable. Uni-processor solutions introduce overhead to manage the queue structure. Multi-processor solutions have the additional problem of communication overhead that can add even more unpredictability to the system.

1.5 Thesis motivation

This thesis introduces a new hardware/software co-design approach for real-time systems that require fine-grained scheduling support that may not be achievable using software-only solutions. The approach exploits the performance advantages of hardware/software co-design that integrates parallelization of independent functions in dedicated and custom hardware working in conjunction with software running on a processor.

CHAPTER 2

FPGA's and Hardware and Software Co Design

2.1 Background

2.1.1 Prior work in hybrid hardware/software co-design

Improvements in processor clock speeds and memory size have provided continual incremental performance enhancements to existing desktop applications. Additional incremental enhancements have resulted from new architectural techniques associated with out-of-order execution and instruction-level-parallelism, and deeper caching. Both technology and architectural advancements have resulted in increased throughput of multiple time sliced programs: the design objective of desktop systems. These techniques however, can have adverse effects on the worst-case execution times of individual programs. While not an issue for desktop systems with no time deadline constraints, guaranteed worse case execution time, or turnaround time, is of vital importance for real time and embedded systems. The idea of having to go down through multiple cache levels into a physical memory due to cache misses, or executing code out of order to achieve a higher aggregate combined throughput can result in missing scheduling deadlines and be catastrophic. Counter to the desktop architectures developed for general-purpose systems, different platform architectures are needed for real time systems that focus on minimizing the worst-case execution times to support real-time constraints [8].

2.1.2 Parallel systems

Parallel systems have emerged in response to meeting tight timeliness requirements. These include multi-processor systems, application-specific integrated circuits (ASIC), field programmable gate arrays (FPGA), systems with several chips on a die, system-on-a-chip (SoC), and dynamically reconfigurable systems. The commonality within all of these systems is to exploit the parallel capabilities of the added hardware functionality

ASIC's provide application specific specialized hardware, while FPGA's provide a programmable sea of gates that can be configured and modified for multiple purposes. Systems with parallel subsystems on a die can provide advantage of the processing power of other processors running in parallel, while a SoC usually has a single processor, which

takes advantage of the FPGA's flexibility by an interconnection between the processor and the FPGA. Finally, the dynamically reconfigurable systems can set themselves in different configurations for a particular situation.

2.1.3 Hybrid systems

The classic approaches for hardware acceleration generally fall into the following three non-disjoint categories: 1) Exploiting recursive patterns, 2) increasing quality of service, and 3) meeting real-time constraints. It is intuitive how parallel hardware can trade space for time to exploit independent recursive and iterative software loops. Iterative and parallel loops that repeat code segments on independent data sets require valuable clock cycles to implement the same operations on the different data sets. As an example DES encryption streams data through complicated bit shift and XOR operations with a key, 16 rounds per piece of data. By unrolling and pipelining the operations into parallel hardware, it is possible to perform a simple space-time tradeoff that results in a linear reduction of the encryption time. Improving the quality of service is also evident from the previous example. A well-designed hardware-based encryption system should be able to encrypt/decrypt different 64-bitdata every singe clock cycle, hence increasing the throughput of the system. An example of hardware being used for quality of service can be seen at the backend of network systems where the computation of intensive scheduling decision logic is often needed for controlling data movement in the form of packets. Finally, hardware has been used as an accelerator for time critical real time functions. In this scenario, computations are executed in parallel and controlled by the system software that ensures all timing constraints are met [10].

2.2 PLD's

PLD's were a development of the simple Programmable Logic Array (PLA), which has been available in electronics design since the early 1980s.The most common (and interesting) form of PLD in use is a Field Programmable Gate Array (FPGA). The key characteristics of an FPGA are as follows:

• "Field-Programmable" denotes their ability to have their program contents changed upon power-up, i.e. in the field;
• "Gate Array" indicates their structure of a regular array of logic gates;
• They provide a logic device of relatively low complexity;
• They compute some function of a set of digital inputs to produce a set of digital outputs;
• They have semi-permanent state in terms of programmed lookup tables, typically implemented as static random access memory (SRAM);
• They operate mainly in a highly parallel manner;
• They are programmed by the download of lookup table data from an external source;
• They differ from other programmable logic devices (PLAs, PROMs, CPLD's) by allowing a more complex flow of data through themselves; and
• They also differ from Application Specific Integrated Circuits (ASIC's) by trading speciality of design for speed of development and economy of small-scale production [15].

2.2.1 Introduction to FPGA's

FPGA's made their first appearance in 1984; manufactured by the company Xilinx.They are a compromise between a software implementation of their function (easier to program but somewhat slower) and a custom-made chip (faster and more reliable, but expensive and requiring more time to design and fabricate). A diagram of a "generic" FPGA is shown in *Figure 2.1*. The key components are the input and output pins, the array of look-up tables (LUTs), the routing logic, the external control and configuration loading, and the interfaces to external RAM and ROM blocks. As a result of this compromise,

FPGA's are typically used in building a prototype system in place of a custom ASIC. It is significantly cheaper and quicker to use such devices when the alternative is a minimum production run of 5000 ASIC's in a different company's fabrication plant ("fab").

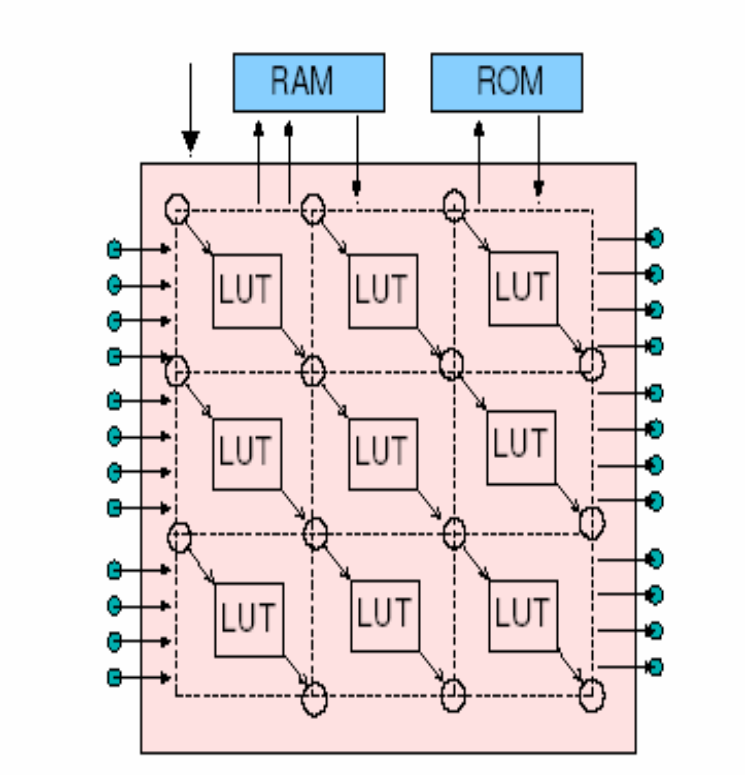


Figure 2.1 Architecture of a generic FPGA

There can be significant commercial gain in using FPGA's rather than ASIC's. Time to market is reduced, since there is not the delay in setting up and making the ASIC production run, and there is little overhead if an error is subsequently found in the device. There is also the potential for increased time-in-market, providing mid-life upgrades to the FPGA code without having to replace the hardware. FPGA's are also found in end-user products. Their ability to take processing load off the main system processor (e.g. as a bus interface) means that they provide a cheap way of increasing a system's speed without the complexity and expense added by an ASIC or second processor. Most PC sound, graphics and network cards will feature one or more FPGA's. For very simple combinatorial logic functions, FPGA's can be too complex a solution: devices such as Complex Programmable Logic Devices (CPLD's), or even PLAs may be appropriate. The majority of PLD's are usually programmed in VHDL or Verilog. These Hardware

Description Languages (HDLs) have substantial standard libraries, allowing a certain amount of code reuse. They model the PLD as interconnected blocks rather than providing higher-level functions such as one to operate on a data stream. Even if a higher-level language or design tool is used, it will normally compile its input into VHDL or Verilog. FPGA's can play a useful role in system development and be an effective component in end-user systems [13].

2.2.2 Description

An FPGA is characterized by a collection of cells, each of which has a number of single bit inputs and outputs. It typically uses a single clock for the whole device; multiple clocks are usually possible but seriously complicate programming. At each clock tick, the cell uses an internal lookup table to compute a function of its inputs, and possibly some internal state value, resulting in a defined output and possibly a change of state. The output is routed to other cells in a predefined manner, and new inputs are read in preparation for the next cycle. The FPGA's interface to the outside world occurs at a set of pins, each of which is a single-bit input or output. Since the pins are normally electrically identical, each pin's function will depend on the user-programmed routing inside the FPGA. These pins are linked to cell inputs or outputs respectively; the precise linkages will again depend on the user's routing scheme. The way that a user programs the FPGA will depend on the FPGA type. Some have SRAM cells that need to be reprogrammed whenever the device is powered up; others use Flash memory that retains data even when power to the device is removed. Both of these technologies may allow the user to reprogram the device mid-computation, with varying effects on the device's state. Some may use once-only programming (such as antifuse technology) which again retains data across power cycling but which requires a new device if the programming is to be changed. The reprogrammable aspect of an FPGA concerns the cell lookup tables, and also the routing tables in many FPGA's. Data for these tables are loaded using special control pins to supply a stream of bits to the FPGA. The FPGA will typically be configured in a period of tens of milliseconds. More advanced FPGA's may include small banks of random access memory (RAM) or other specialized devices such as DSP units that interface to cells. Such complications can be ignored since they do not affect the

fundamental functionality of FPGA's, and could be viewed as devices separate from the main FPGA circuitry; they just happen to be on the same piece of silicon [19].

2.3 Co-design Techniques

Though the co-design process applies to a wide spectrum of applications ranging from consumer electronics to plant control systems, the focus of this document is on its emphasis in embedded-SOC applications.

2.3.1 The Conventional Approach

Traditionally, system design has involved manual intervention at various stages of the design flow. This not only affects the efficiency of the process, but also the order in which different steps are performed [17].

A typical design process would involve the following

- **Specification**
- **Partitioning**
- **Synthesis**
- **Integration**
- **Co-Simulation**
- **Verification**

Figure 2.2 illustrates this widely used design flow in the conventional approach to co design. Each step is explained in detail in the sections that follow.

2.3.1.1 System Specification

The system design process starts with a specification of the system. The requirements of the system are typically specified in a non-formal language. However, there have been several attempts to formalize system specification to help in automation of the various steps that follow. System specification also incorporates Timing, Area and Power constraints.

2.3.1.2 System Partitioning

The system designer then uses the specification and his experience to make educated guesses on the performance of the system. Based on these guesses, he decides which part of the system will be implemented in hardware (as an ASIC) and which part in software. This step is called partitioning. It also involves writing the behavioral1 description for the different parts of the system. The hardware part, for example, might be described using VHDL or Verilog and the software model using the *C language*. In addition, the interfacing logic, including any handshaking or bus logic, is also decided at this time.

2.3.1.3 Synthesis

Synthesis is the process of generating the physical model from the behavioral model. This translation can be performed with the help of synthesis tools like design compilers and cross compilers. Hardware synthesis tools, for example, can read hardware descriptions written in VHDL or Verilog format and generate mask layouts. Similarly, cross compilers compile programs written in a high-level language into the native instruction set of an embedded processor.

2.3.1.4 Hardware/ Software Integration and Co-simulation

Co-simulation is a difficult task and is a topic of recent research. There are some commercially available co-simulation platforms that can simulate hardware and software synthesized models in an integrated environment. But typically, designers either simulate the synthesized models separately and interpret the results or generate prototypes that can be simulated. The latter method, however, tends to be expensive, more so because this process may have to be repeated several times with modifications to the original design.

2.3.1.5 Design Verification

The results of co-simulation are verified against functional requirements and design constraints from the specification. The performance of the system is also validated at this step. If the system does not meet the requirements, the entire process, starting with system partitioning, is repeated. Verification results may be used as hints to modify design decisions.

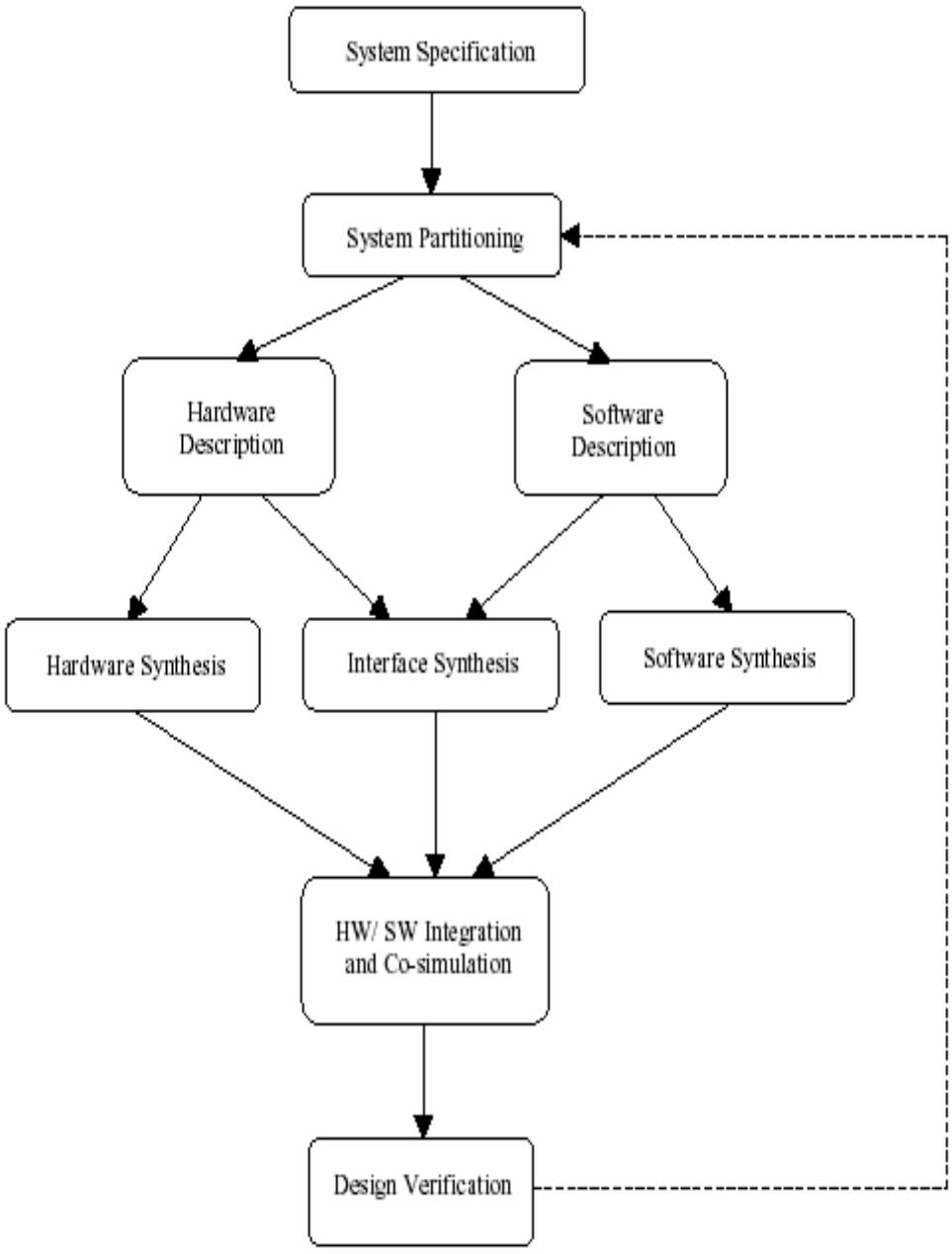


Figure 2.2 Conventional Approach to Hardware/ Software Co-design

2.3.2 Model Based Approach

In the conventional approach discussed above, system partitioning is done very early in the design process. This reduces the flexibility of the designer and thus the efficiency of

the final design. In this section, a more efficient scheme is proposed. Though the discussion is more general, it fits well to specific tools that were used as a part of this research. The process is illustrated in *Figure 2.3*. As in every design flow, System Specification is the first step. The introduction given in the previous section holds for this section as well [14].

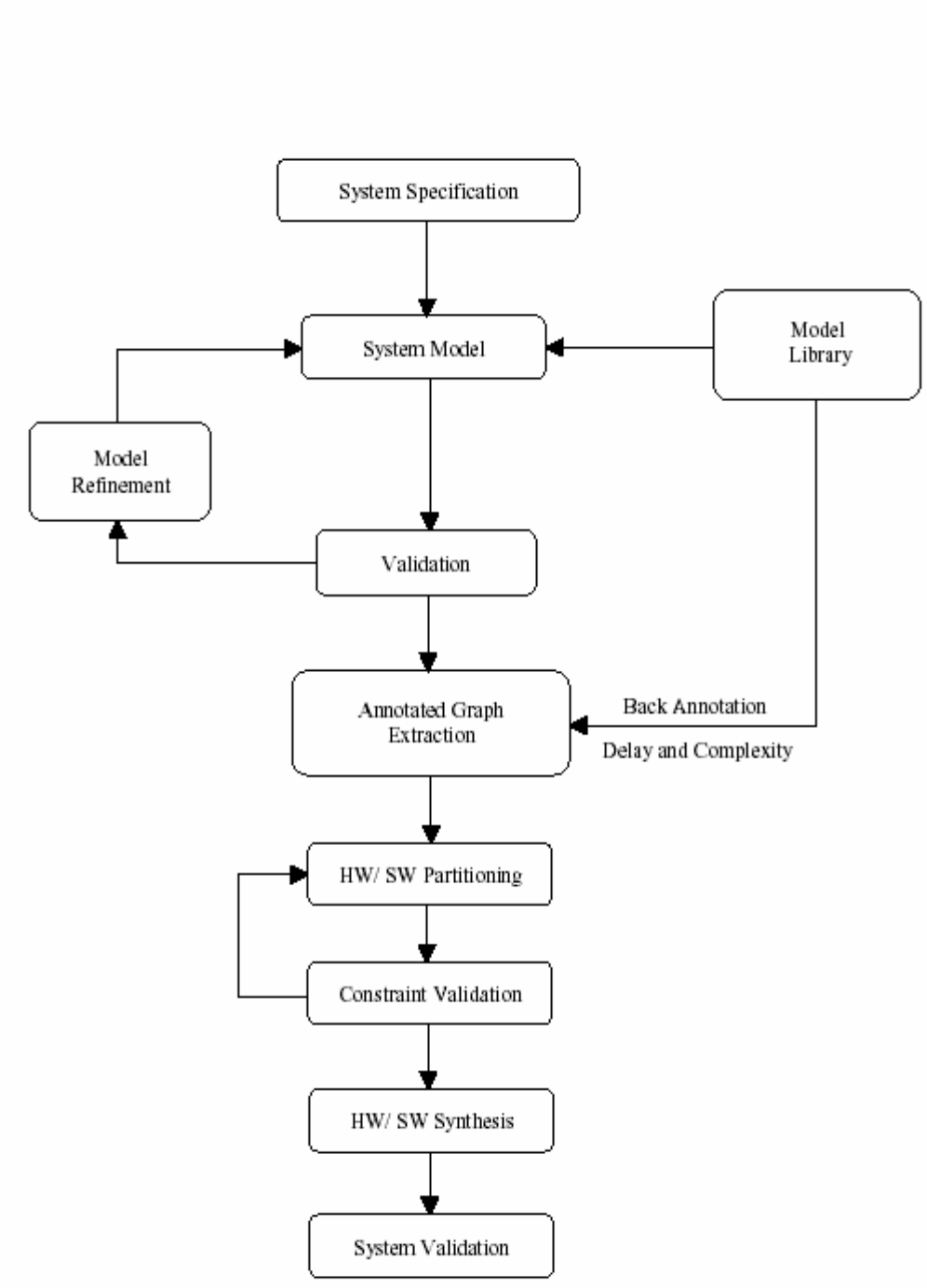


Figure 2.3 Model Based Approach to HW/ SW Co-design

2.3.2.1 System Model

The system model is a behavioral representation of the system written in a well-defined set of instructions. The amount of detail that a model abstracts is called the granularity of the model. There are different levels of granularity. At the lowest level, a system can be modeled as an *algorithm* that just implements a truth table. Higher granularity is achieved by writing *register-level* and *gate-level* descriptions. The Higher the granularity, the closer the model will be to the actual system. A good system model should have enough information to allow it to be simulated. The results of the simulation should be identical to the results expected from the final system. The set of instructions, or the language in other words, that describe a system model has been evolving over several years. An ideal SLML (System Level Modeling Language) describes the exact behavior of any part of the system, independent of whether it is implemented as hardware or software [1].

2.3.2.2 Model Library

The model can either be built from scratch or existing models can be reused. Models of various components of the system that have been thoroughly tested are added to a model library to facilitate reuse. As the library grows over time, design time is greatly reduced.

2.3.2.3 Validation and Model Refinement

Results of the simulation are used to validate the model. The validation here is largely functional verification and does not involve timing, power or area constraints. The output of the simulation is matched with the expected values. The validation results may also be used to refine the model. For example, redundant or unused states and interfaces may be removed from the system.

2.4 FPGA hardware/software co-design

The ability to fabricate smaller and smaller transistors has had significant effect on computer system design trends. Moore's law, which states that the processing speed of a

CPU doubles every three years has continued to hold true. Moore's law is also being followed by programmable devices. *Figure 2.4* below shows the change on design trends over two decades of embedded systems evolution.

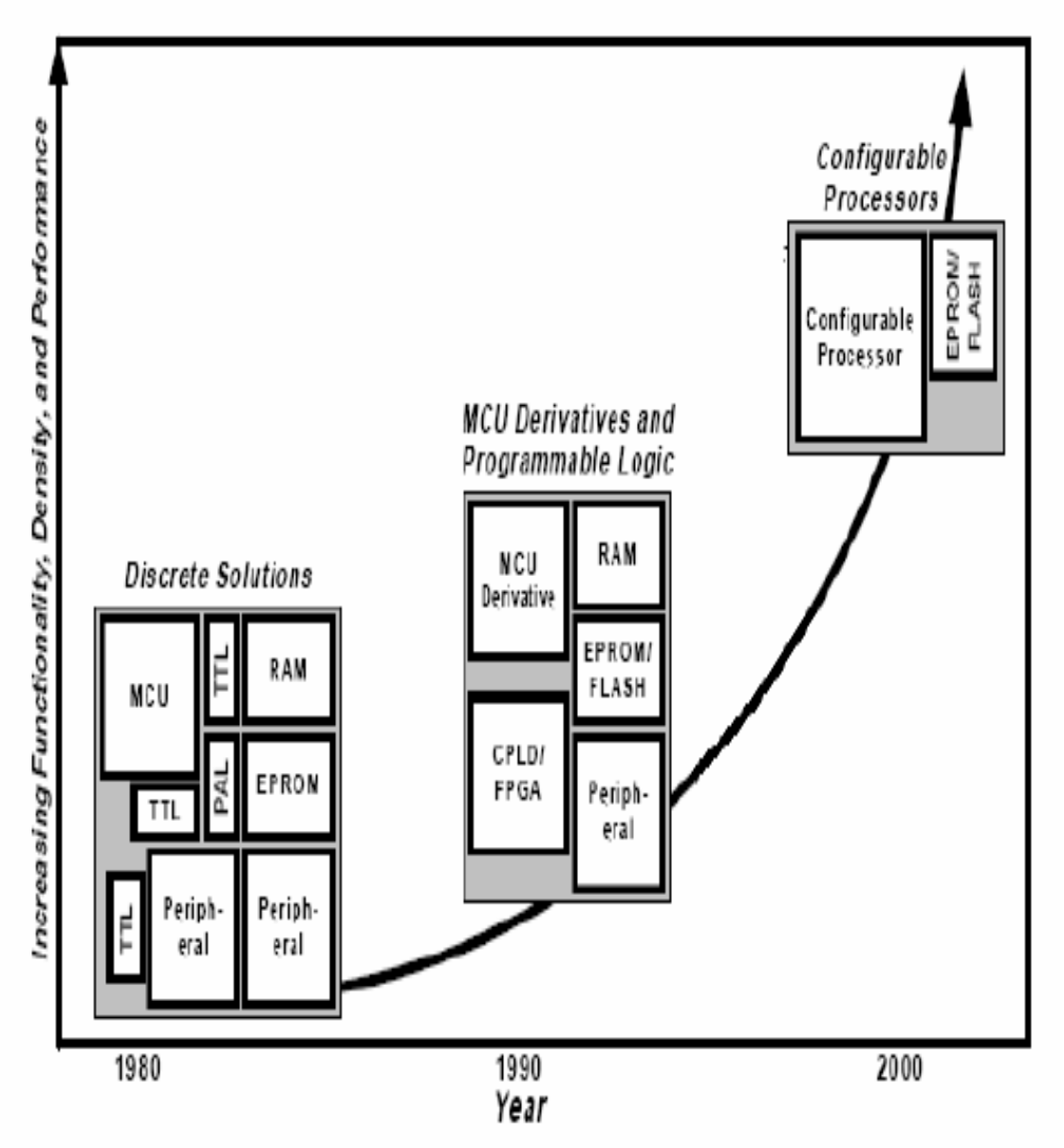


Figure 2.4 New programmable technologies of increasing density and performance

In the early 1980's embedded systems were built using discrete components. The Micro Controller Unit (MCU) was the central processing unit and Transistor-Transistor- Logic (TTL) devices or Programmable Logic Devices (PLD's) provided the interface between

the controller and its discrete peripherals .By the late 1980's and start of the mid 1990's, companies started using MCU derivatives like Complex Programmable Logic

Devices (CPLD's) and FPGA's, which provided control and some functionality for MCU peripherals. The MCU was integrated with additional system RAM and boot PROM to create integrated versions of the earlier MCU's. With the development of Intellectual Property cores now provided by companies such as Xilinx and Altera, and the increased capabilities of FPGA and CPLD devices, entire systems are now being built on a single silicon die (SoC's). These SoC's, which can be customized or configured for specific applications, reduce the economic disadvantage and inflexibility associated with ASIC customized designs, but still provide customization. The most current offerings for SoC's such as the Virtex II Pro and Excalibur now provide a dedicated processor and programmable logic on a single configurable chip. Commercially available IP cores such as UARTs and device controllers can be incorporated and even tailored into an existing SoC chip within the FPGA fabric. These platforms represent a robust environment for development of wide ranging and changing application requirements.

FPGA's provide flexibility in custom hardware circuit design. With these new hybrid chips, it now becomes viable to execute parallel processes running in both the software and in the FPGA hardware. In fact, an arbitrary number of parallel processes could be running in the FPGA along with concurrent processes on the CPU. Researchers are now exploring approaches that support the dynamic migration of parallel and concurrent processes fluidly across the CPU and FPGA components [9].

2.4.1 what are embedded systems?

A digital system may be providing a service as a self-contained unit, or as a part of a larger system. A traditional computer (with its peripherals) is an example of the first kind of systems. A digital control system for a manufacturing plant is an example of the latter case. Systems that fall in this second category are commonly referred to as embedded systems. The term embedded means being part of a larger unit and providing a dedicated service to that unit. Thus a personal computer can be made the embedded control system for manufacturing in an assembly line, by providing dedicated software programs and appropriate interfaces to the assembly line environment. Similarly, a microprocessor can be dedicated to a control function in a computer (e.g., keyboard/mouse input control) and

be viewed as an embedded controller. Digital systems can be classified according to their principal domain of application. Examples of self-contained (i.e., nonembedded) digital systems are information processing systems, ranging from laptop computers to supercomputers, as well as emulation and prototyping systems. Applications of embedded systems are ubiquitous in the manufacturing industry (e.g., plant and robot control), in consumer products (e.g., intelligent home devices), in vehicles (e.g., control and maintenance of cars, planes, ships), in telecommunication applications, and in territorial and environmental defense systems. An embedded IC with programmable DSP cores is shown in *Fig 2.5* below.

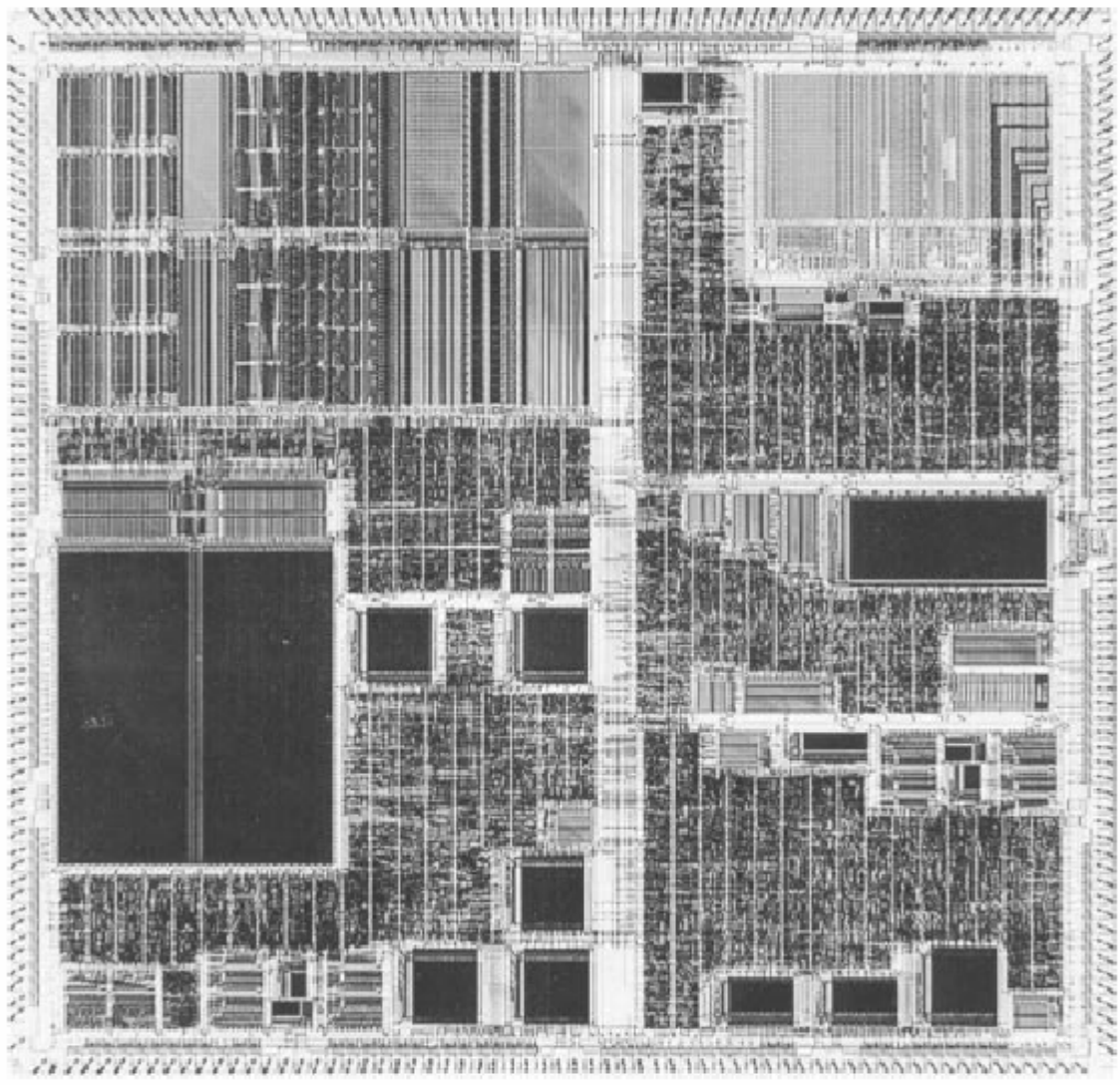


Fig 2.5 An embedded IC with programmable DSP cores

2.4.2 Desired embedded control system properties

Real time computers control larger embedded systems. Because of the close interaction with physical components and communication between systems, several properties are required for a real-time embedded control system. These are timeliness, concurrency, live ness, interfaces, heterogeneity, reliability, reactivity, and safety. Timeliness is the property that is concerned with total executions times. Therefore timeliness is one of the considerations that must account for not only the time for execution of instructions, but also all other system delays times such as communication times. This is essential if the system has concurrent processes running in parallel that must synchronize or react to asynchronous stimulus between processes. Live ness is the property that the system must never stop running, either by halting, suspending or terminating. This scenario would be deemed defective if implemented in hardware. Component interfaces must be well defined, not just for their static unification of component communication ports, but also for the dynamics of computations occurring between them and their timing. The heterogeneity of the system comes into play during the dynamic part of inter-component communication, as different components need to "talk" to each other in an understandable way. Hence, either software must accommodate to receive a constant stream of computation from a hardware process, or hardware must expect discrete results from a software procedure, or a tradeoff between the two must be achieved. The communication between parts of the system, and the level of correct system behavior, must be reliable. *Fig 2.6* shows essential parts of an embedded system

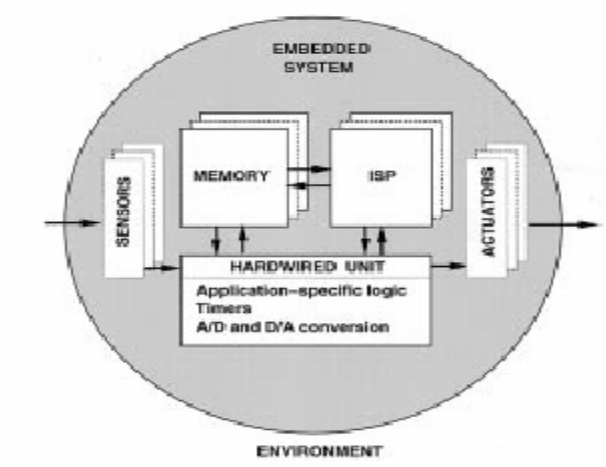


Fig 2.6 Scheme of essential parts of an embedded system

This is because the system will have high reactivity to the events happening around it in real-time. Finally, safety is always an issue in embedded and real time systems, since the system is probably in control of expensive or invaluable objects, and a failure could even result in a great loss of life [16].

2.4.3 Design considerations for HW/SW co-design

Solutions for problems with time consuming simple and constant repetitive processes, like network packet routing, data encryption, mathematical matrix operations, multimedia encoding and high speed signal processing are very suitable for hardware based implementations. Problems involving slower, more complex, variable computations, like software applications, GUI displays and end-user input processing are more appropriate for software. Hence it is observed that while different goals might require different parts, these are clearly not disjoint, and an effective system design would use both hardware and software to achieve a combined purpose complying with system design restrictions.

2.4.4 Average case / worst case

Real time systems have specific time constraints that must be considered during the initial design process. The average case delay of the system becomes of secondary importance against the worst-case scenario. Care must be taken in finding worst-case execution time paths in a real-time system to ensure its correctness. All variability in code execution and system processing times must be identified and accounted for to determine the worst-case execution time. The worst-case execution time is more easily achieved if the system has no influence from outside devices. However, a primary feature of most systems nowadays is the ability to communicate to peripherals running in different time scales, receiving and handling signals from sensors, controlling outside actuators, and receiving interrupts to conduct more

critical operations, etc. These functions can have very broad minimum and a maximum processing times. Not only is quantifying these time gaps important in determining worst-case time, but even if known can still degrade the overall utilization factor of the system.

2.4.5 System predictability

If the system is running a complex operating system, the number of states it can reach can be significant. However, if concentration on the hardware components is done, a certain degree of model state checking can be achieved. This is especially true for ASIC's, but FPGA's with a companion chip on a SoC should also be predictable. This checking can be guaranteed under normal conditions such as receiving events from the outside, be those an ASIC receiving a signal for a pushed button on a vending machine, or a CPU memory request to a hardware peripheral inside an FPGA.

2.4.6 System flexibility and performance

Another design consideration for a hybrid system is flexibility. FPGA's are configured at synthesis time, while dynamically reconfigurable hardware will eventually allow reconfiguration at run time. This flexibility can be used to adjust the design for die area, system speed, resource allocation, power consumption, and system performance. While designers want their constraints to be met and fulfilled, they also want the system to achieve useful computations, hence why a certain level of system performance is also required. In fact, designers will want the same or better level of performance of a non real-time operating system.

CHAPTER 3

Real time Systems

Real-Time Operating Systems (RTOS) are commonly used in the development, productizing, and deployment of embedded systems. Unlike the world of general purpose computing, real-time systems are usually developed for a limited number of tasks and have different requirements of their operating systems. This section first gives the requirements of real-time operating systems, then breaks down the internals of RTOS's and explains them in detail.

3.1 What are Real-time Systems?

Timeliness is the single most important aspect of a real-time system. These systems respond to a series of external inputs, which arrive in an unpredictable fashion. The real time systems process these inputs, take appropriate decisions and also generate output necessary to control the peripherals connected to them. As defined by Donald Gillis "A real-time system is one in which the correctness of the computations not only depends upon the logical correctness of the computation but also upon the time in which the result is produced. If the timing constraints are not met, system failure is said to have occurred." It is essential that the timing constraints of the system are guaranteed to be met. Guaranteeing timing behaviour requires that the system be predictable. The design of a real-time system must specify the timing requirements of the system and ensure that the system performance is both correct and timely. There are three types of time constraints:

§ **Hard**: A late response is incorrect and implies a system failure. An example of such a system is of medical equipment monitoring vital functions of a human body, where a late response would be considered as a failure. Another example of this is the communication mechanism from the cockpit of a commercial airliner to the embedded system controlling the wing flaps. If a pilot is coming in for landing, and pulls up on his flaps to slow his

descent, that communication must work for if it doesn't, the entire plane has the possibility of crashing.

§ **Soft**: Timeliness requirements are defined by using an average response time. If a single computation is late, it is not usually significant, although repeated late computation can result in system failures. An example of such a system includes airlines reservation systems. Another example of this is an Automated Teller Machine (ATM). If the software running upon the ATM takes a little longer to process a request, other than the costumer being slightly upset, the system will be able to perform its tasks, albeit late.

§ **Firm**: This is a combination of both hard and soft timeliness requirements. The computation has a shorter soft requirement and a longer hard requirement. For example, a patient ventilator must mechanically ventilate the patient a certain amount in a given time period. A few seconds' delay in the initiation of breath is allowed, but not more than that. One need to distinguish between on-line systems such as an airline reservation system, which operates in real-time but with much less severe timeliness constraints than, say, a missile control system or a telephone switch. An interactive system with better response time is not a real-time system. These types of systems are often referred to as soft real time systems.

In a soft real-time system (such as the airline reservation system) late data is still good data. However, for hard real-time systems, late data is bad data. Most real-time systems interface with and control hardware directly. The software for such systems is mostly custom-developed. Real-time Applications can be either embedded applications or non-embedded (desktop) applications.

Real-time systems often do not have standard peripherals associated with a desktop computer, namely the keyboard, mouse or conventional display monitors. In most instances, real-time systems have a customized version of these devices. The following table compares some of the key features of real-time software systems with other conventional software systems [4] [3].

A comparative study of real time systems and other programming systems is made in *Table 3.1*.

Feature	Sequential Programming	Concurrent Programming	Real-time Programming
Execution	Predetermined order	Multiple sequential programs executing in parallel	Usually composed of concurrent programs
Numeric Results	Independent of program execution speed	Generally dependent on program execution speed	Dependent on program execution speed
Examples	Accounting, payroll	UNIX operating system	Air flight Controller

Table 3.1 Comparative study between real time and other programming systems

3.2 Real-Time Operating Systems: The Requirements

A good RTOS not only offers efficient mechanisms and services to carry out real-time scheduling and resource management but also keeps its own time and resource consumption predictable and accountable. A RTOS is responsible for offering the following facilities to the user programs that will run on top of it. The first responsibility is that of scheduling: a RTOS needs to offer the user a method to schedule his tasks. The second responsibility is that of timing maintenance: the RTOS needs to be responsible in both providing and maintaining an accurate timing method. The third responsibility is to offer user tasks the ability to perform system calls: the RTOS offers facilities to perform certain tasks that the user would normally have to program himself, but the RTOS has them included in its library, and these system calls have been optimized for the hardware system that the RTOS is running on. The last thing that the RTOS needs to provide is a method of dealing with interrupts: the RTOS needs to offer a mechanism for handling interrupts efficiently, in a timely manner, and with an upper bound on the time it takes to service those interrupts.

There are several concepts that need to be defined in any discussion of RTOSs. The first concept is that of preemption. Real-time operating systems are either preemptive or non-

preemptive. If a real-time operating system is preemptive, it means that a task currently being run by the RTOS can be interrupted by another task with a higher priority or an external interrupt. The interrupted task's state is saved, and this state will be restored when it is run again, allowing it to continue along from the same point that it was interrupted. RTOSs that are nonpreemptive cannot be interrupted. If a task is currently running when a second task needs to run, that second task must wait for the first task to finish running before it can begin to run. Another important concept is that of hard real-time versus soft real-time. Hard real-time means that a task must always be completed by a specific time. The integrity of the system designed with hard real-time tasks will be compromised if such a deadline is missed.

There are several different types of task scheduling for today's real-time operating systems to choose from. There is the endless loop scheduler, that is basically a while (1) loop that continuously runs a piece of code. Activities within the loop are executed in sequence and as many times as possible. The next level of task scheduling is that of the basic cyclic executive scheduler. In a basic cyclic scheduling algorithm, the idea of the endless loop is extended in that designers can separate the code to be executed into separate tasks. These tasks execute in a standard sequence in an infinitely repeating loop. This type of scheduling is often called round robin scheduling. Like the endless loop, all of the tasks run as often as possible. Time driven cyclic scheduling, the next level of task scheduling, differs from basic cyclic in that instead of running each one of these tasks as often as possible, it introduces the idea of a time interrupt. In this scheduler, one hardware timer is used to wake up all tasks. This timer wakes up the first task in line, and as soon as that first task is finished, the next task runs. All of the tasks in line must finish before the next timer interrupt. Following the time driven cyclic scheduler is the multi-rate cyclic executive scheduler. This is an expansion of the time driven cyclic scheduler in that it allows multiple periods, so long as higher frequency tasks are a multiple of the

base task's frequency. This is done by inserting a task more than one time into the chain or into multiple chains. The multi-rate executive for periodic tasks scheduler adds the ability to have multiple periods by instituting a timer that is the lowest common multiple

of all of the periods of all of the tasks. At each tick of this timer, tasks can be made to execute. All of the above scheduling algorithms usually deal with interrupts by inserting tasks that poll for them, and all of the above scheduling algorithms are nonpreemptive. A multi-rate executive with interrupts allows external interrupts to break into current execution and be serviced. The task interrupted is then restarted when the interrupt is done. Finally, the priority based preemptive executive scheduler is the same as the multi-rate executive with interrupts except that it allows not only interrupts to break into the current program, but tasks with higher priority as well. Scheduling algorithms are either static or dynamic. Static scheduling is performed when the execution times of all tasks to be scheduled by the scheduler are determined before any execution has taken place. Static scheduling is done when the deadlines for all of the tasks are known, and the time that it takes to execute those tasks is also known. All of the scheduling is done offline, before the execution of any tasks has begun, and is fixed. Dynamic scheduling is performed when the execution time of the tasks to be run is not fixed, is variable, and scheduling orders and priorities must be done dynamically during execution. This is done when task priority, execution time, or deadlines either change during execution, or are unknown before execution begins. The order in which tasks are scheduled and executed is decided upon during runtime, and is variable [11].

3.3 User Tasks and Threads

In RTOSs, user tasks are implemented in the form of threads. Each thread implements a computation job and is the basic unit of work handled by the scheduler. When the kernel creates a thread, it allocates memory to that thread and brings in the user code to be executed by that thread. The two different types of threads are periodic and aperiodic. Obviously, aperiodic threads run only once while periodic thread runs continuously at a given frequency. Their are five major states of threads. The first is sleeping: this is when a task is set to sleep for a certain amount of time before it is to be woken up and run. The

second state is ready: this is when the thread is ready to run and is simply waiting for the resources to do so. The third state is that of executing: this is when a thread is currently

running on the operating system. The fourth state is that of suspended, or also known as blocked: this is when the task cannot proceed for some reason, such as waiting for a new event to occur, or for some value to be brought in. The final state is terminated: this is when a thread has run, and is not to be run again.

3.4 The Kernel

The kernel in any RTOS, as mentioned in the introduction to this section, is responsible for four things. They are scheduling, system calls, timing maintenance, and handling interrupts. The RTOS is responsible for maintaining a schedule for all of the tasks running on it, and one of the above scheduling techniques is usually chosen. A system call is any function that the kernel might do at the request of a user thread. To perform a system call, the user task places the name or ID of the function that it wishes to run in a preset location and then traps to the kernel. After the context switch has taken place, the kernel looks up the function that it has been asked to complete, completes it, and puts the result of that function, if there is one, in a second preset location, and then returns control over to the user process. It is also possible for the user process to make a system call and continue working while the kernel is performing this system call. The kernel is also responsible for maintaining the timer. Every time that a timer interrupt is handled, the kernel must update the time as well as wake up tasks that need to be woken up and put on the ready queue. The last thing that a RTOS is responsible for is the handling of interrupts. Upon a interrupt, the hardware starts the RTOSs exception handler software. The RTOS is then responsible for saving the current state on the stack, determining the type of interrupt that has interrupted normal processing, and to know where that interrupt's service routine is. It then turns over control to that interrupt's service routine. After that routine has finished, the kernel is also responsible for transferring control back to the user process.

3.5 Synchronization and Communication

In addition to all of the above requirements, RTOSs are also responsible to provide better methods of synchronization and communication between tasks. Mechanisms such as semaphores, mutexes, and condition variables add the ability for tasks to synchronize amongst themselves. To allow communication between the tasks, mechanisms such as message queues, mailboxes, and shared memory can be provided by the RTOS [13].

3.6 The Emulator's Benefit to Real-Time Operating Systems

One of the biggest decisions in choosing a RTOS for an embedded system is not which RTOS to choose, but whether or not to use a RTOS. Unlike the world of general purpose computing, embedded systems are usually developed for a limited number of tasks. Any facilities that these tasks might need are often built directly into the code, so many designers believe that a real-time operating system would just add unnecessary overhead. What is needed, and what, is a method to test both commercially available Real-Time Operating Systems and in-house creations on the target architecture to verify which would give the best performance, without having to run the RTOS on the actual hardware, saving both time and money [12] [6].

3.7 Evaluation of Real-Time Systems

From complex mathematical theories to full system hardware simulation, there are many different ways to evaluate real-time systems. The evaluation of these systems, like research in many fields, usually falls into two parties; theoretical and experimental. While many argue for one over the other, these two fields should not be at odds against each other. They are in fact complementary, and any evaluation cannot really be said to be complete without both having been performed. This section provides both the methods and metrics used to evaluate those real-time systems, provides examples of current research that is being done in the evaluation of real-time systems, and concludes with the benefits that the emulator that was developed for this report can give to the evaluation of Real time systems.

3.7.1 Methods of Evaluation

There are varying levels of real-time systems evaluation. The most prevalent ones are the use of analytical models, the simulation of scheduling algorithms, and simulation of the hardware.

Analytical models are mathematical theorems and proofs that model the worst time performance of one or more of the aspects of real-time systems, and by changing certain inputs to these theorems; an optimum performance can be proven. Simulation takes the analytical models one step further in creating a simulation using scheduling theory to experiment with behavior of real-time systems. Finally, hardware tests take the theorems that were postulated by the analytical model and have been simulated through the use of scheduling algorithms, and run tests on the actual hardware to discover any behavior that was not determined through either of the other two methods.

3.7.2 Metrics of Characterization

Two of the most common metrics used to characterize real-time systems are jitter and response time. Jitter represents the minimum and maximum time separating successive iterations of periodic tasks. If this inter-arrival time is greater than the period of the task, it means that the task is running late, and this will show up as a positive jitter value. If that inter-arrival time is less than the period of the task, that means that the task is running early, and this will show up as a negative jitter value. Response time is the time that it takes for a real-time system to respond to an external interrupt and represents the reaction time of the system to an unscheduled event while under load.

3.8 Current Studies

Simulation and hardware execution of real-time software has been used in many different projects: from validating the accuracy of schedulers and analytical models to measuring Worst-case execution time of functional blocks in dataflow graphs, to measuring the effects of pipelined and super scalar processors on timer analysis, and to validating the performance of real-time databases. However, while some simulations are accurate down to cycle behavior, most experiments model systems by using dataflow graphs to represent real-time system behavior [7].

3.9 The Emulator's Benefit to the Evaluation of Real-Time Systems

The analysis of these scheduling algorithms should be accompanied with experimental evaluation on the actual hardware. Unfortunately, this sometimes presents a problem when the hardware is not available, or there is a question of money or time. However, with the help of an emulator it is possible to run tests on an emulation of that hardware, saving both time and money.

CHAPTER 4

Real Time schedulers

4.1 Scheduling

As specified in the section attractive uses for hybrid software/hardware real-time operating systems include interrupt handling, task scheduling, memory management, resource allocation, and data routing. Of particular interest are the cases for interrupt handling and task scheduling.

4.1.1 Scheduler operation

When there are multiple levels of processes working "simultaneously" on the same system, its needed to schedule certain shared resources (CPU, memory) in a fair manner. A scheduler takes care of managing the processes/entities, the resources they are requesting, and running a scheduling decision/algorithm to assign resources to requesters for a specified/unspecified amount of time. However, such a decision does not come for free, and oftentimes, even efficient resource allocation scheduling algorithms end up performing worse than a random scheduler. This can result from the decision function itself, as it requires resources from the system in order to run. The scheduler functionality is usually dependent on *how* it's allocating *which* resources *to whom* and for *how long*. The most common task for a scheduler is allocating CPU time to different programs, threads or events, for an organized amount of time to ensure fairness and freedom from starvation, guaranteeing every program will get a CPU share. This allows for Uni-processor multitasking models that allow different programs to run at different times in a form that simulates concurrency. As the number of processors a scheduler must manage increases, so does its internal logic and communication costs. A scheduler for real-time systems must handle real-

time events, including stimulus from the outside world collected through sensors, or a high priority event. In all cases, these events must be given special importance, and the execution of the event be met within its time constraint. The adopted method to handle these situations is to create an interrupt for the CPU, and run the interrupt service routine. This is a costly process, particularly if a system has to support multiple real-time interrupts or handle fine granularity events scheduled to run within a small period of time. Handling these events has the unfortunate cost of CPU cycles, the very resource that is in demand. As previously indicated, if the scheduling decision is complex, the larger the cost will be during the scheduler runtime. Such an overhead to an already overloaded real-time system is not desirable [12] [3].

4.1.2 Scheduler components

A scheduler in general has five main components, which vary slightly from implementation to implementation. The first component of the scheduler defines what the event type is: an interrupt, a program thread, an object, etc. However, what is needed only is to know what the event is, not what the event does. Handling the event is typically left for a different software component within the system. The second component is the queue structure for the events. This component takes care of storing event information for each event and allowing events to be added or removed. The next component is the scheduling algorithm, which is the decision function to choose the next event to be scheduled. The most common algorithms are Earliest Deadline First (EDF), Rate Monotonic, and Priority Based. The way in which the scheduling algorithm is implemented inside the event queue is considered the fourth non-trivial component. This is because sorting operations can take a large amount of computational time, which is in itself crucial to real-time systems, and lead to hardware supported sorting designs. The final component of a scheduler is the interface back to whatever requested a scheduler service, in general a CPU. This can be achieved through interrupt controllers, message passing or shared memory.

4.1.3 Hardware scheduler research

A fair amount of research has been done in the area of migrating schedulers and time critical application programs into hardware for real time systems. The most straightforward approach is to implement hardware support for real time applications that cannot execute fast enough in software. Here, the scheduler will assign certain special priority real-time tasks to dedicated hardware in the system. The next approach is to implement the scheduler in hardware, through an FPGA. The scheduler is usually static, but some systems allow dynamic scheduling algorithms, which can be changed during run-time. Allowing reconfiguration in the FPGA's also brought forth different scheduling techniques for non-static schedulers, and with the arrival of multiprocessor machines, new schedulers sprung forth to maximize resource utilization, with the main function of being hardware accelerators. Most of these approaches have undesirable limitations in their implementation. One vital but often overlooked property of most systems is the way the processing of interrupts are handled. While most of these systems implement optimizations for handling interrupts once an interrupt is detected, they fail to notice that there is indeed a non-trivial amount of time in which the interrupt flag is set and waiting to be noticed by the CPU, which most likely won't happen until the next scheduling decision function iteration. However, providing a finer granularity for the scheduler means paying a higher overhead of computation on the system. The reconfigurable solutions can provide better performance at the cost of hardware reconfiguration overhead, which might not be possible in a real time system. And lastly, multiprocessor solutions that handle scheduling are not ideal, since there would be certain unreliability in the system due to communication and synchronization costs. These tradeoffs in the designs are limiting factors for possible deployment of real-time systems.

4.2 Various types of Scheduling and their algorithms

The scheduling problem has many facets. Scheduling algorithms have been developed in both the operation research and computer science community, with different models and objectives. The techniques that are applicable today to the design of hardware and software systems draw ideas from both communities. Generally speaking, hardware and

software scheduling problems differ not just in the formulation but also in their overall goals. Nevertheless, some hardware scheduling algorithms are based on techniques used in the software domain, and some recent system-level process scheduling methods have leveraged ideas in hardware sequencing. Scheduling can be loosely defined as assigning an execution start time to each task in a set, where tasks are linked by some relations (e.g., dependencies, priorities,). The tasks can be elementary (like hardware operations or computer instructions) or can be an ensemble of elementary operations (like software programs). When confusion may arise, tasks are referred to as operations in the former case, and to processes in the latter. The tasks can be periodic or aperiodic, and task execution may be subject to real time constraints or not. Scheduling under timing constraints is common for hardware circuits, and for software applications in embedded control systems. Tasks execution requires the use of resources, which can be limited in number, thus causing the serialization of some task execution. Most scheduling problems are computationally intractable, and thus their solutions are often based on heuristic techniques. Next section describes scheduling algorithms as applied to the design of hardware, compilers, and operating systems [18].

4.2.1 Operation Scheduling in Hardware

Now the major approaches to hardware scheduling is considered . These techniques have been implemented (to different extent) in CAD tools for the design of ASIC's and DSP's, which are modeled with a behavioral-level HDL (e.g., VHDL, Verilog HDL, and DFL). The behavioral model can be abstracted as a set of operations and dependencies. The hardware implementation is assumed to be synchronous, with a given cycle-time. Operations are assumed to take a known, integer number of cycles to execute. (a consideration is done to remove this assumption later). The result of scheduling, i.e., the set of start times of the operations, is just a set of integers. The usual goal is to minimize the overall execution latency, i.e., the time required to execute all operations. Constraints on scheduling usually relate to the number of resources available to implement each operation and to upper/lower bounds on the time distance between start times of operation pairs. Usually, the presence of resource constraints makes the problem intractable. The scheduling problem can be cast as an integer linear program, where

binary-valued variables determine the assignment of a start time to each operation. Linear constraints require each operation to start once, to satisfy the precedence and the resource constraints. Latency can also be expressed as a linear combination of the decision variables. The scheduling problem has a dual formulation, where latency is bounded from above and the objective function relates to minimizing the resource usage, which can also be expressed as a linear function. Timing and other constraints can be easily incorporated in the ILP model. The appeal of using the ILP model is due to the uniform formulation even in presence of different constraints and to the possibility of using standard solution packages. Its limitation is due to the prohibitive computational cost for medium-large cases. This relegates the ILP formulation to specific cases, where an exact solution is required and where the problem size makes the ILP solution viable. Most practical implementations of hardware schedulers rely on list scheduling, which is a heuristic approach that yields good (but not necessarily optimal) schedules in linear (or over linear) time. A list scheduler considers the time slots one at a time, and schedules to each slot those operations whose predecessors have been scheduled, if enough resources are available. Otherwise the operation execution is deferred. Ties are broken using a priority list, hence the name. Another heuristic for scheduling is force-directed scheduling, which addresses the latency-constrained scheduling problem. Here, operations are scheduled into the time slots one at a time, subject to time-window constraints induced by precedence and latency constraints. Ties among different time slots for each operation are broken using a heuristic based on the concept of force, which measures the tendency of the operation to be in a given slot, to minimize overall concurrency. The computational cost of force-directed scheduling is quadratic in the number of operations. When resource constraints are relaxed, the scheduling problem can sometimes be solved in polynomial time. For example, scheduling with timing constraints on operation time separation can be cast as a longest-path problem. On the other hand, scheduling under release times and deadlines is intractable, unless the operations take a single cycle to execute. There are several generalizations of the scheduling problem. In some cases, operations are not restricted to take an integral number of cycles to execute, and more than one operation can be chained into a single time slot. Pipelined circuits require specific constraints on data rates, and additional resource conflicts have to be taken into account due to the

concurrent execution of operations in different pipe stages. Periodic operation subsets, e.g., iteration construct bodies, may be advantageously scheduled using loop pipelining techniques, which is an example of a method borrowed from software compilers. Chaining and pipelining can be incorporated in ILP, list, and forcedirected schedulers. The synchronization of two (or more) operations or processes is an important issue related to scheduling. Synchronization is needed when some delay is unknown in the model. Relative scheduling is an extended scheduling method to cope with operations with unbounded delays called anchors. In this case, a static schedule cannot be determined. Nevertheless, in relative scheduling the operations are scheduled with respect to their anchor ancestors. A finite-state machine can be derived that executes the 360 operations in an appropriate sequence, on the basis of the relative schedules and the anchor completion signals. The relative scheduling formulation supports the analysis of timing constraints, and when these are consistent with the model, the resulting schedule satisfies the constraint for any anchor delay value. Scheduling with templates is a similar approach, where operations are partitioned into templates that can be seen as single scheduling units. Thus templates are useful for hierarchical scheduling and scheduling multicycle resources (e.g., pipelined multipliers).

4.2.2 Instruction Scheduling in Compilers

Compilers are complex software tools, consisting of a front-end, a suite of optimization routines operating on an intermediate form, and a back-end (called also code generation) which generates the machine code for the target architecture. In the context of compilation, instruction scheduling on a uniprocessor is the task of obtaining a linear order of the instructions. Thus it differs from hardware scheduling because the resource constraints typically refer to storage elements (e.g., registers) and the hardware functional resource is usually one ALU. In the more general case, scheduling can be viewed as the process of organizing instructions into streams. Instruction scheduling is related to the choice of instructions, each performing a fragment of the computation, and to register allocation. When considering compilation for general-purpose microprocessors, instruction selection and register allocation are often achieved by dynamic programming

algorithms, which also generate the order of the instructions. When considering retargetable compilers for ASIP's, the compiler back-end is often more complex, because of irregular structures such as inhomogeneous register sets and connections. As a result, instruction selection, register allocation and scheduling are tightly coupled phases of code generation. In both cases, scheduling objectives are reducing the code size (which correlates with the latency of execution time) and minimizing spills, i.e., overflows of the register file, which require memory access. Optimizing compiler algorithms for ASIP's and general purpose DSP's has been a subject of recent research activities. Instruction selection, instruction scheduling, and register spilling problems for ASIP's are solved using a *branch-and-bound algorithm*. Scheduling has been modeled by resource and instruction set conflicts and solved by *bipartite matching algorithms*. Some researches consider code generation for basic blocks in heterogeneous memory-register DSP processors and used register-transfer paths to convert basic block graphs into expression trees, which are used in code generation. The co-design of deeply pipelined microprocessors can leverage the coupling between instruction scheduling and hardware organization. Pipeline hazard avoidance can be achieved by hardware means or by software means (e.g., instruction reorder and NOP insertion). Recent research has addressed the problem of the concurrent synthesis of the pipeline control hardware and the determination of an appropriate instruction reorder that the corresponding back-end compiler should use to avoid hazards.

4.2.3 Process Scheduling in Different Operating Systems

Process scheduling is the problem of determining when processes execute and includes handling synchronization and mutual exclusion problems. Algorithms for process scheduling are important constituents of operating systems and run-time schedulers. The model of the scheduling problem is more general than the one previously considered. Processes have a coarser granularity and their overall execution time may not be known. Processes may maintain a separate context through local storage and associated control information. Scheduling objectives may also vary. In a multitasking operating system, scheduling primarily addresses increasing processor utilization and reducing response time. On the other hand, scheduling in real-time operating systems (RTOS) primarily

addresses the satisfaction of timing constraints. Scheduling without real-time constraints is considered first. The scheduling objective involves usually a variety of goals, such as maximizing CPU utilization and throughput as well as minimizing response time. Scheduling algorithms may be complex, but they are often rooted on simple procedures such as:

4.2.3.1 Shortest job first (SJF)

The SJF is a priority-based algorithm that schedules processes according to their priorities, where the shorter the process length (or, more precisely, its CPU burst length) the higher the priority. This algorithm would give the minimum average time for a given set of processes, if their (CPU-burst) lengths were known exactly. In practice, predictive formulas are used. Processes in a SJF may be allowed to preempt other processes to avoid starvation.

4.2.3.2 Round robin (RR)

The round-robin scheduling algorithm uses a circular queue and it schedules the processes around the queue for a time interval up to a predefined quantum. The queue is implemented as a first-in/first-out (FIFO) queue and new processes are added at the tail of the queue. The scheduler pops the queue and sets a timer. If the popped process terminates before the timer, the scheduler pops the queue again. Otherwise it performs a context switch by interrupting the process, saving the state, and starting the next process on the FIFO.

4.2.3.3 Static and Dynamic algorithms

Process scheduling in real time operating system is characterized by different goals and algorithms. Schedules may or may not exist that satisfy the given timing constraints. In general, the primary goal is to schedule the tasks such that all deadlines are met: in case of success (failure) a secondary goal is maximizing earliness (minimizing tardiness) of task completion. An important issue is predictability of the scheduler, i.e., the level of confidence that the scheduler meets the constraints. The different paradigms for process scheduling in RTOS can be grouped as static or dynamic .In the former case, a

schedulability analysis is performed before run time, even though task execution can be determined at run time based on priorities. In the latter case, feasibility is checked at run time. In either case, processes may be considered periodic or aperiodic. Most algorithms assume periodic tasks and tasks are converted into periodic tasks when they are not originally so.

4.2.3.3.1 Rate monotonic (RM) analysis

Its one of the most celebrated algorithms for scheduling periodic processes on a single processor. RM is a priority-driven preemptive algorithm. Processes are statically scheduled with priorities that are higher for processes with higher invocation rate, hence the name. This schedule is optimum in the sense that no other fixed priority scheduler can schedule a set of processes, which cannot be scheduled by RM. The optimality of RM is valid under some restrictive assumptions, e.g., neglecting context-switch time. Nevertheless, RM analysis has been the basis for more elaborate scheduling algorithms. Let us consider now hardware/software system implementations obtained by partitioning a system-level specification. The implementation consists of a set of software fragments executing on a processor in parallel with the execution of other tasks in dedicated hardware. A relevant problem is to determine the execution windows for both the hardware and software tasks. Since the partition depends on the specific application and design objectives, a run-time scheduler for the system is required that fits the hardware/software partition. Conversely, a given partition may be chosen because a runtime scheduler can assign schedule tasks while satisfying given deadline and rate constraints. Software tasks are represented by threads, each thread being a set of operations with known execution, except possibly the head of the thread. Operations within threads are statically scheduled (with respect to the head of the thread), so that timing constraints are marginally satisfied, i.e., within the limits of the lack of knowledge of the delay of the thread head operation. Threads execution is then dynamically determined by a nonpreemptive run-time scheduler whose task is to synchronize the execution of hardware and software. Thread-based scheduling can be seen as an application and extension of relative scheduling to the hardware/software domain, thus showing the cross fertilization of the hardware and software fields [19] [20].

CHAPTER 5

Different Entities in Hardware Scheduler

The scheduler was implemented in hardware using VHDL language which is the acronym for Very High Speed Integrated circuit Hardware description language. The entire architecture may be viewed as a group of subsystems each having its own functionality and the signals used for their proper working and synchronization between them
There are a total of four different entities that were developed and a complete description of the these entities along with their test simulations is provided below

1. Time slice unit
2. Interrupt control unit
3. Address Generator unit
4. Control unit

5.1 Time Slice Unit

Time slice unit is one of the most imp entities in the whole architecture. Time slice is the duration of the time for which a processor is allocated to a particular process. The time slice approach is used in order to avoid starvation of some process from acquiring the resources of system here the processor. The scheduling algorithm is implemented using this unit and here Round Robin Scheduling algorithm has been used. The unit is implemented using a decrement counter that is when the processor is allocated to a particular process the time slicer starts operating and decrement counter comes into picture. The decrement counter decrements for the period of the time slice allotted to a particular process and keeps on decrementing till it reaches zero value. As soon as it is decremented to zero the time slicer stops signifying that that time slice for a particular process has expired and the processor can be allotted to a new process.
This entity is the most important entity in the whole system as it creates a performance constraint. The better the algorithm and the bigger the time slice the better the performance of the scheduler.
The input signals to the time slice units are a clock signal (clk), a write signal (wr), an eight bit data vector (data _in out) and a stop signal and the output signal is a change signal.

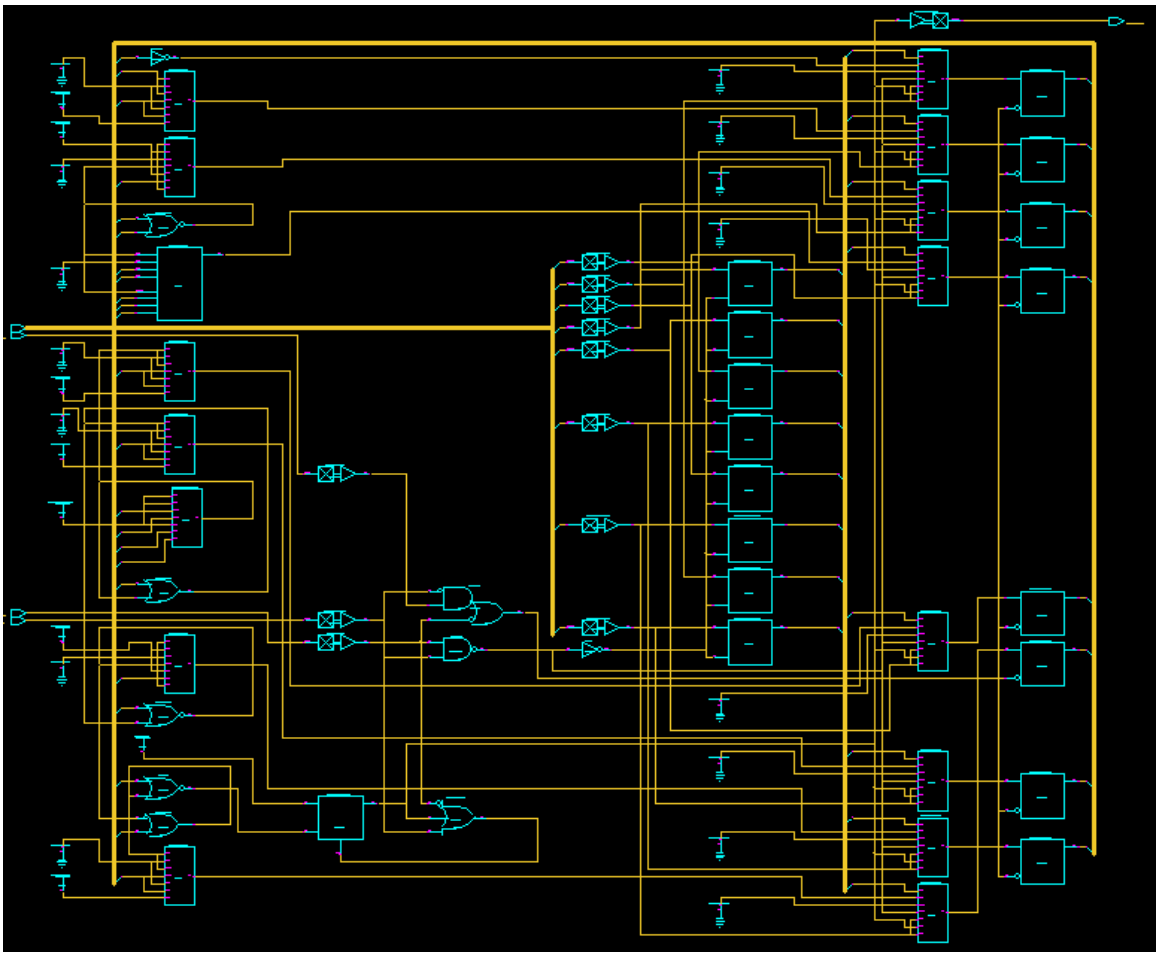


Fig 5.1 Layout of Time slice unit

5.2 Interrupt control unit

Interrupt control unit is the second entity used in the architecture. This entity generate signals that interrupt the normal functioning of the processor and the time slice unit stops working as soon as an interrupt is generated by this unit. The processor stops executing the process and it can now be allocated to a newer process till the interrupt is satisfied. The most basis type of interrupts is I/O device interrupts or a DMA (Direct Memory Access) request or a supervisor call etc.

Here the input signals to the interrupt control unit are eight interrupt signals that consists of seven interrupts (int 0 to int 7) and one interrupt for time slice, a clock signal, a don't signal and a show signal.

The output signals are an 8 bit vector signifying which interrupt is high and a number pin that tells the number of the interrupt.

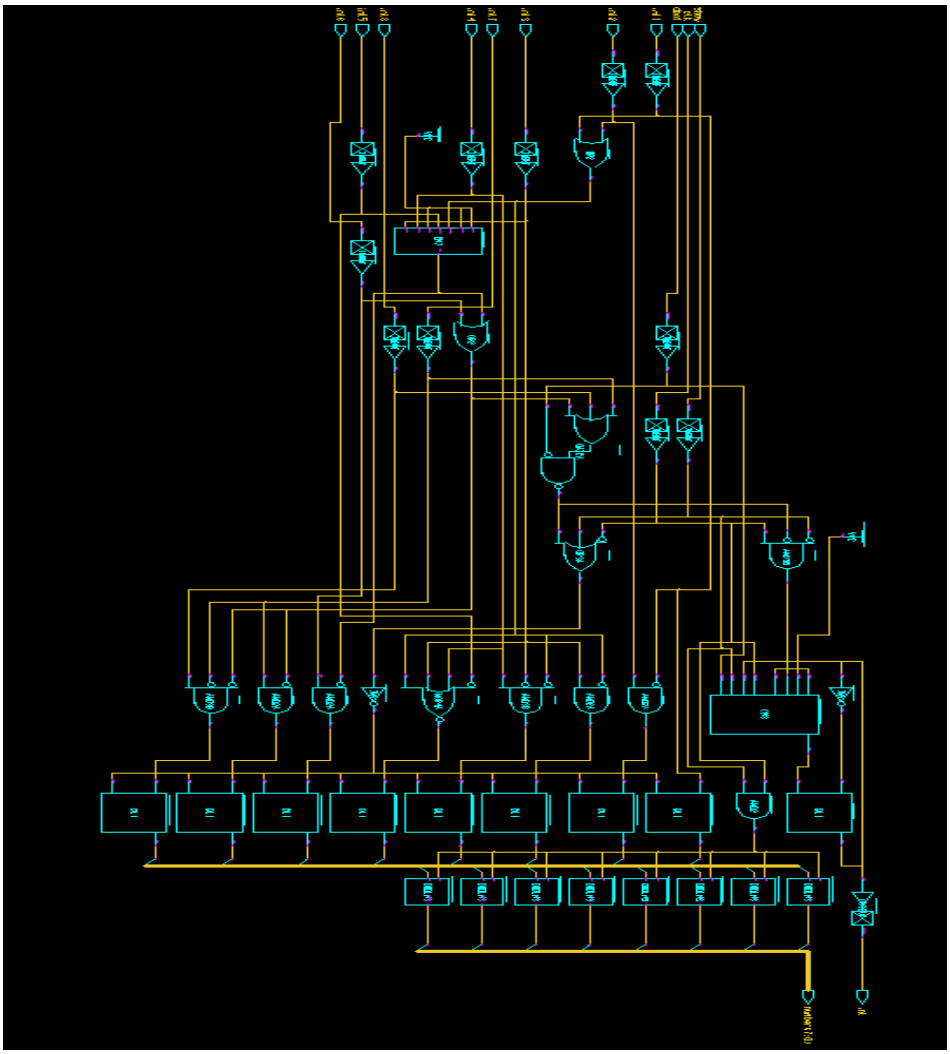


Fig 5.2 Layout of interrupt control unit

5.3 Address Generator unit

The address generator unit is the third entity in the system and it generates the starting address of the next Process control Block (PCB) in memory. A PCB stores all the relevant information related to a particular process including the PID (Process identification number), the registers that are being modified by the process, and the addresses where data and related information is encapsulated. The PCB size supported by the architecture is 20 bytes.

Here the address generator has input signals consisting of an eight bit data vector, a clock signal, and a two bit input vector (mode). In one of the modes the address of PCB is

generated by the unit as two eight bits data, first 8 bits for LSB and next 8 bits for MSB. The address is generated in two clock cycles.
There is one more input signal (inr) that increments the address by one bit so as to generate the next PCB address stored in memory.
The output signals from the unit are a 16 bit vector (address) that is the address of the next PCB generated by the unit.

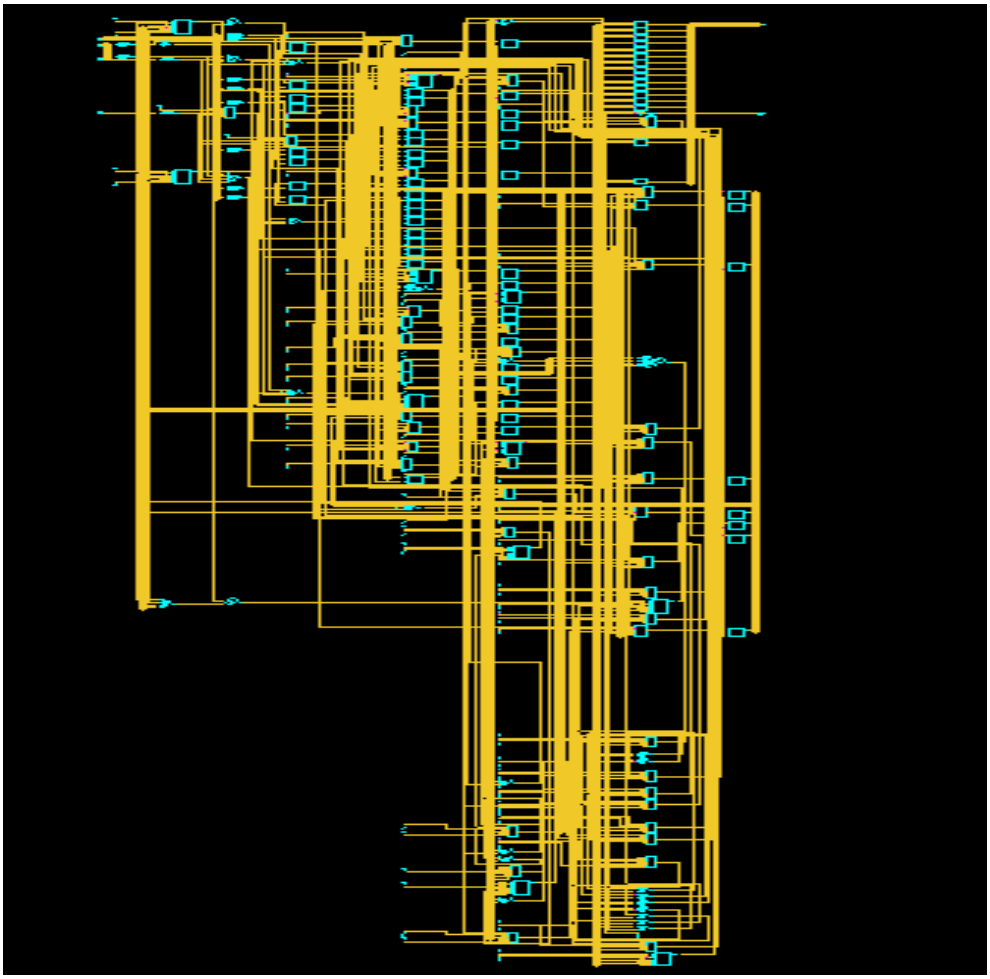


Fig 5.3 Layout of Address generator unit

5.4 Control Unit

Control unit is the very heart of the whole system. The main functionality of this unit is the execution of a particular process. This unit keeps on executing a particular process till the time slice of a particular process expires or some other interrupt is generated by the interrupt unit.
The input signals to the control unit are a 16 bit vector (address) that is the address of the next PCB generated by the Address Generator unit, a clock signal, a 2 bit vector (mode) signifying the mode in which scheduler is working, a write signal (wr), a change signal, a

number signal signifying the number and type of interrupt, and an acknowledgement signal (ack) signifying that the Control unit can now start executing the next process.
The output signals from the unit are an 8 bit data vector, a 16 bit address vector, a show signal, a stop signal, an interrupt signal and an increment signal.

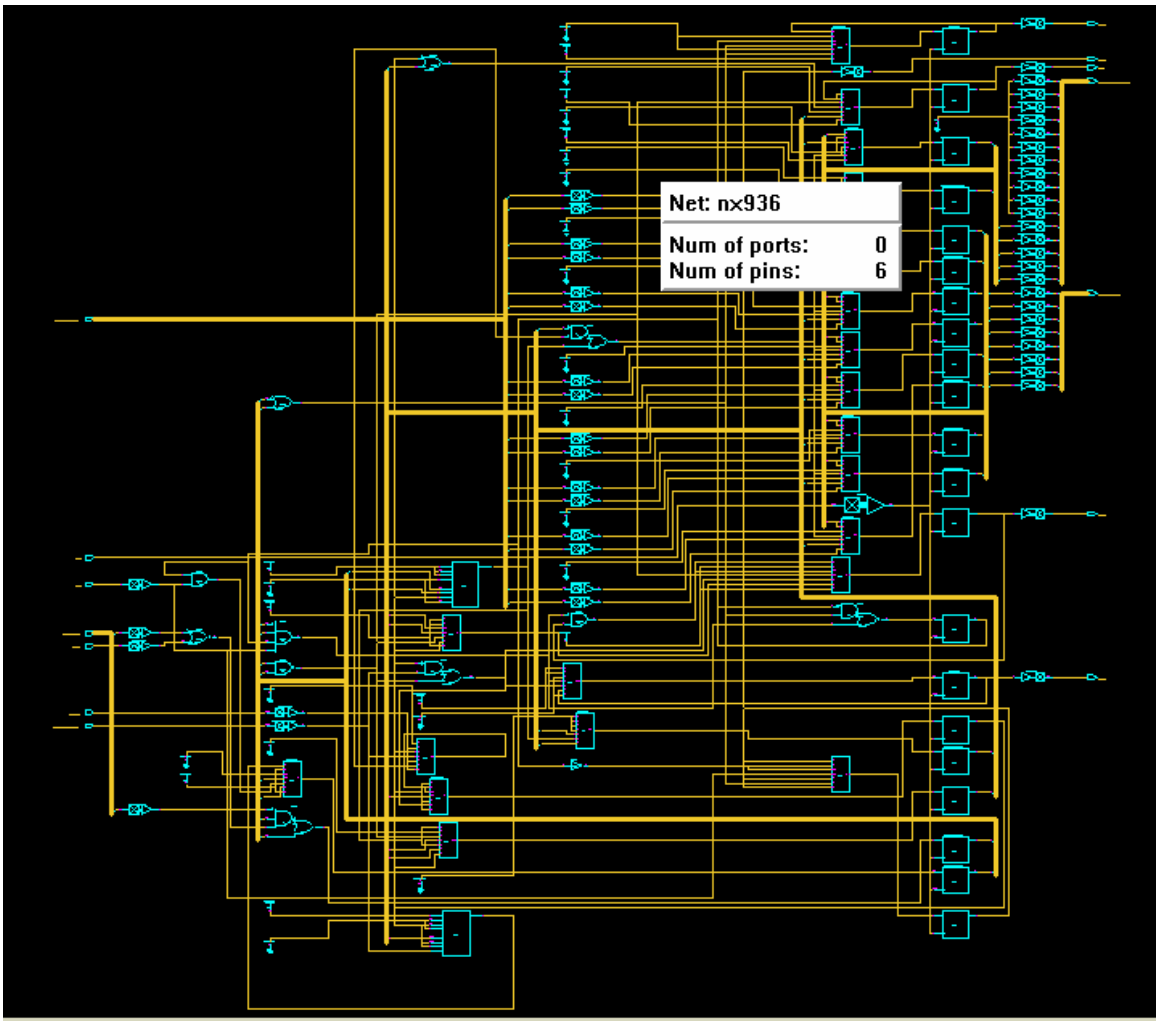


Fig 5.4 Layout of control unit

Chapter 6

Working of Hardware Scheduler

6.1 Working of scheduler in various modes

The scheduler implemented in hardware works in three modes. There is one mode signal whose value defines in which mode scheduler is going to operate.

The three modes are given below

1: Mode “00” ---- Normal Processing

2: Mode “01” ---- Enter total number of processes in memory.

3: Mode “10” ---- Enter starting address of PCB in memory.

During the first clock cycle, with the write signal “HIGH” the value of time slice is entered for which the Control unit executes the process.
During the next clock cycle write signal is made “LOW”, mode is set to value “01” and using the eight Bit data vector the number of PCB’s in the memory is entered.
During the third and fourth clock cycle the mode is set to value “10” and the 16-bit address of first PCB is given with 8 bits of LSB and MSB each entered using the same eight bit data vector.
Now mode is again set to value “00” and the Control Unit starts executing the process till the time slice of the event expires or an interrupt is being generated by the Interrupt Control Unit. A change signal becomes high signifying that new process can be loaded from memory. Inr signal becomes “HIGH” signifying the number of interrupt. Now int pin that sends interrupt to the time slicer unit to stop its working. As soon as time slicer stops a Stop signal becomes “HIGH”. Now an acknowledgement signal is sent to the Control Unit that it can start executing the next process. The PCB address is incremented by 1 to get the next PCB address and the address is generated in two clock cycles. Now the Control Unit starts executing the next process till the change signal is activated signifying another interrupt or expiration of time slice.
The timing diagram for the Hardware Scheduler as generated by Model SIM software is shown in *Fig 6.1* and the architectural layout of the scheduler as generated by Leonardo Spectrum is shown in *Fig 6.2*.

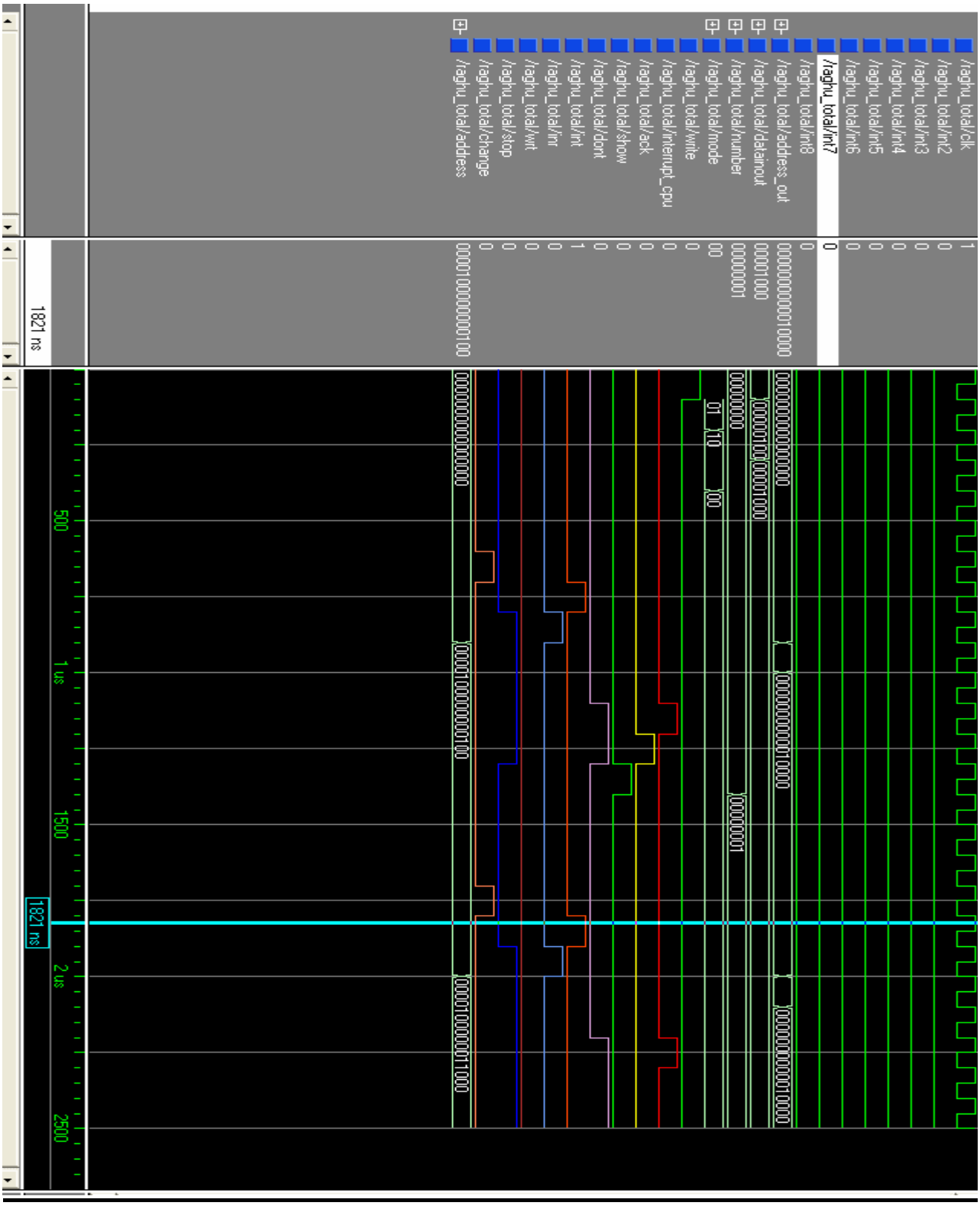


Fig 6.1 Timing Diagram of Hardware Scheduler

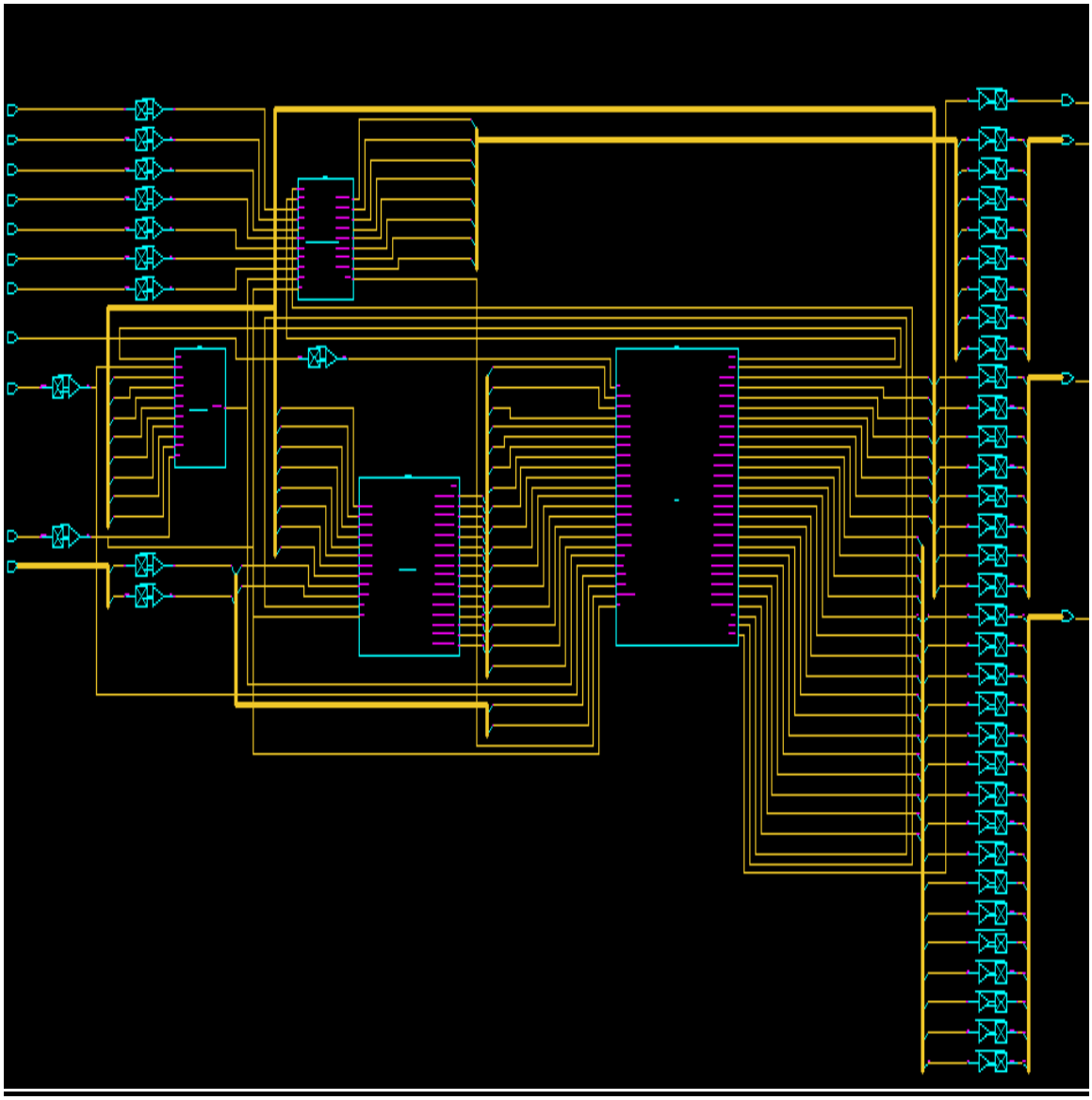


Fig 6.2 Architectural layout of hardware scheduler

Conclusions and Future Scope

Real time operating systems rely heavily on both the hardware platform the system runs on, the real time services that this platform is able to provide, and the operating system running on the software side. For proper hardware/software co-design of the system, hardware must be utilized to provide timely execution for real time services, while the operating system must grant access to these services with minimum overhead penalties, all while still complying with the constraints of the system. For embedded systems, the increase in complexity of software applications and the real-time services in dedicated hardware are usually mutually exclusive. A more flexible solution was found by combining reconfigurable logic in hardware and a real-time version of a commercially available operating system in software. To reduce the overhead caused by operating system core operations such as time management and event scheduling, the timer and the event scheduler were moved into hardware, which would account for reduced overhead time spent in context switches.

FPGA's provided us with enough flexibility to not only design and implement hardware-based services for real time operating systems, but also to exploit portability by using IP cores when implementing them. The correctness of the main functions of the event scheduler were tested and evaluated.

Round Robin algorithm was implemented that avoided the starvation of processes for acquiring of CPU resources.

It is also concluded from the results that the throughput of the scheduler increases with increase of time slice but upto a certain limit and after that it starts to decrease again.

The frequency with which the chip operates is 30 MHz and the total number of gates used is 1500.

While no major performance gain was expected (or achieved) by migrating the event scheduler to hardware, it set a strong base for further research in hardware schedulers. Current research on hardware/software co-design includes creating hardware-based threads. Eventually, these hardware threads will be running in parallel with threads in software on the same system. The scheduler scheduling for these threads will be hardware-based, thus allowing the CPU to be interrupted to make scheduling decisions only when its completely necessary, yielding higher system, processor and resource utilization. At this point, it will be likely that better decision functions will be added to the scheduler, according to system specifications (group scheduling, priority, etc).

The performance of scheduler can be enhanced by implementing the scheduling algorithm in a better way.

References

[1] L. Abeni and G. Buttazzo, "Integrating multimedia applications into hard real-time systems", In Proc. *IEEE Real-Time Systems Symposium* (RTSS), 1998.

[2] J. H. Anderson, et al, "Efficient object sharing in quantum-based real-time systems", In Proc. *IEEE Real-Time Systems Symposium* (RTSS), 1998.

[3] T. Anderson, "System-on-chip design with virtual components", Circuit Cellar, No. 109, pp. 12-19, August 1999.

[4] M. J. Bach, "The Design of the Unix Operating System. Prentice-Hall", Englewood Cliffs, NJ, 1999.

[5] S. R. Ball, "Embedded Microprocessor Systems: Real-World Design", Newnes, Boston MA, 1999.

[7] A. Bestavros and S. Nagy, "Value-cognizant admission control for RTDB systems", In Proc. *IEEE Real-Time Systems Symposium* (RTSS), 1999.

[8] M. Brockmeyer, et al, "A flexible, extensible simulation environment for testing real-time specifications", In Proc. *IEEE Real-Time Systems Symposium* (RTSS), 2000.

[9] D&T Roundtable, "Hardware-Software co design", *IEEE Design and Test of Computers*, Vol. 14, No. 1, pp. 75-83, 2000.

[11] R. Enrst, "Co design of Embedded Systems: Status and Trends", IEEE Design and Test of Computers, Vol. 15, No. 2, pp. 45-54, 2001.

[12] D. D. Gajski and F. Vahid, "Specification and Design of Embedded Hardware-Software Systems", *IEEE Design and Test of Computers*, Vol. 12, No. 1, pp. 53-67, 2001.

[14] J. G. Ganssle, "An OS in a can", Embedded Systems Programming, January 2001.

[15] J. G. Ganssle, "The challenges of real-time programming", Embedded Systems Programming, Vol. 11, No. 7, pp. 20-26, July 2002.

[16] L. Garber and D. Sims, "In Pursuit of Hardware-Software Co design", *IEEE Computer*, Vol. 31, No. 6, pp. 12-14, 2002.

[17] R. K. Gupta, "A framework for interactive analysis of timing constraints in embedded systems", In Workshop on Hardware-Software Co-Design (CODES), 2002.

[19] R. K. Gupta and G. D. Micheli, "Specification and analysis of timing constraints for embedded systems", IEEE Transactions on Computer- Aided Design, Vol. 16, No. 3, pp. 240–256, March 2003.

[21] J. R. Haritsa, et al, "Earliest deadline scheduling for real-time database systems", In Proc. *IEEE Real-Time Systems Symposium* (RTSS), 2003.

[22] M. G. Harmon, et al, "A retargetable technique for predicting execution time", In Proc. *IEEE Real-Time Systems Symposium* (RTSS), 2003.